The Most Elegant Bump Number Algorithm

Gara Pruesse

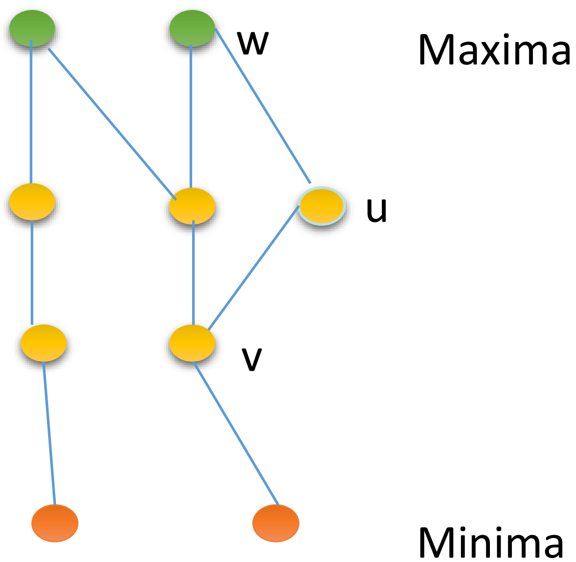
Vancouver Island University

Derek Corneil

Lalla Mouatadid

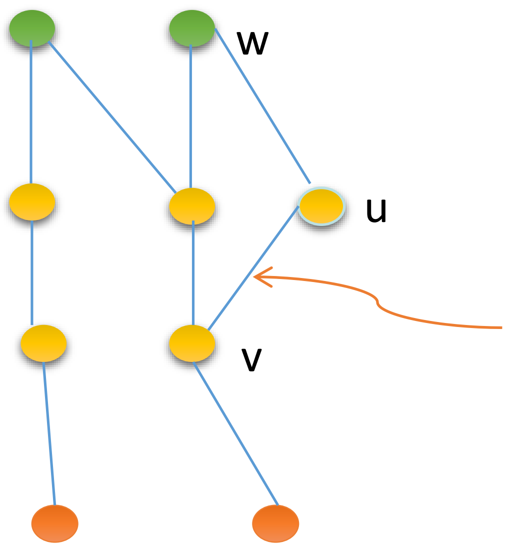
University of Toronto

Posets = partially ordered sets



Hasse Diagram

Posets = partially ordered sets



w > v

v < w

v and w are *transitively related*

u *covers* v

u is an *upper cover* of v

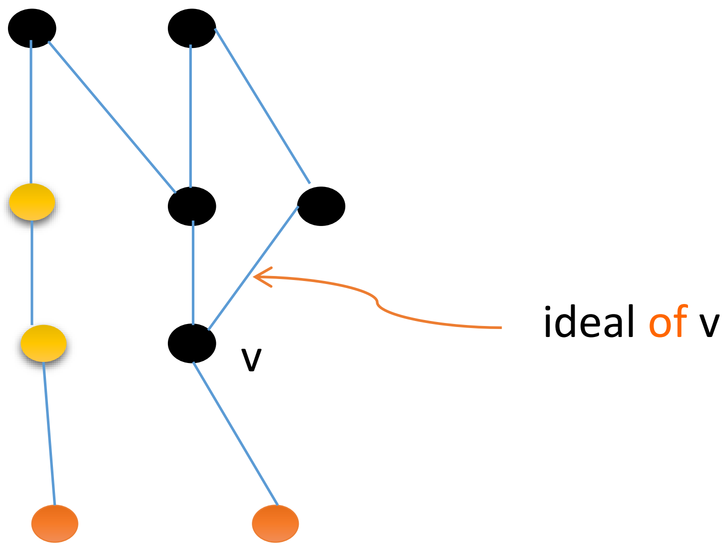
v is a *lower cover* of u

$u \lessdot v$

Hasse Diagram

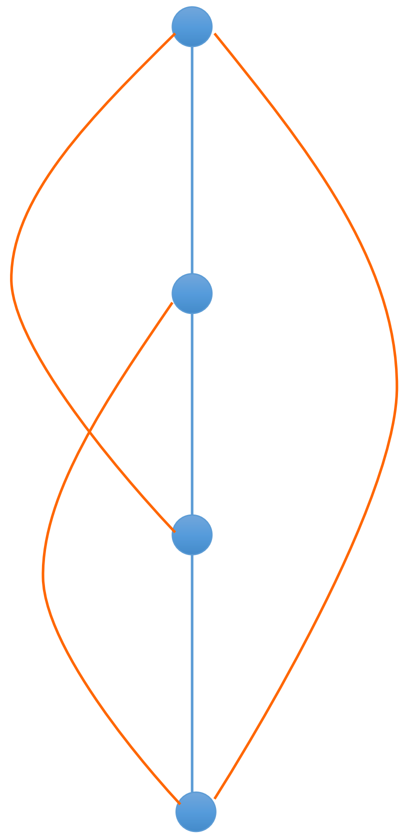
- A compact representation of a set of relations
- i.e. can be $O(n)$ representation of $O(n^2)$ relations

Posets = partially ordered sets



Hasse Diagram

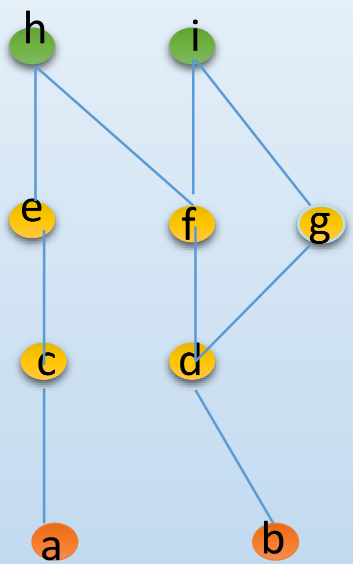
- A compact representation of a set of relations
- i.e. can be $O(n)$ representation of $O(n^2)$ relations



Hasse Diagram
O(n) for chain

Comparability Graph
O(n^2) for chain

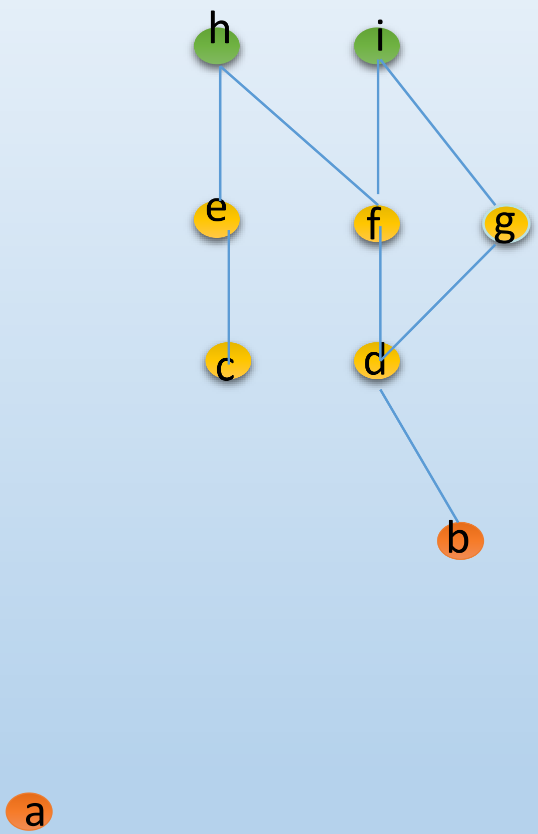
Posets = *partially* ordered sets



Can always be extended (add relations) until the ordering is *linear*

… equivalent to shelling the poset

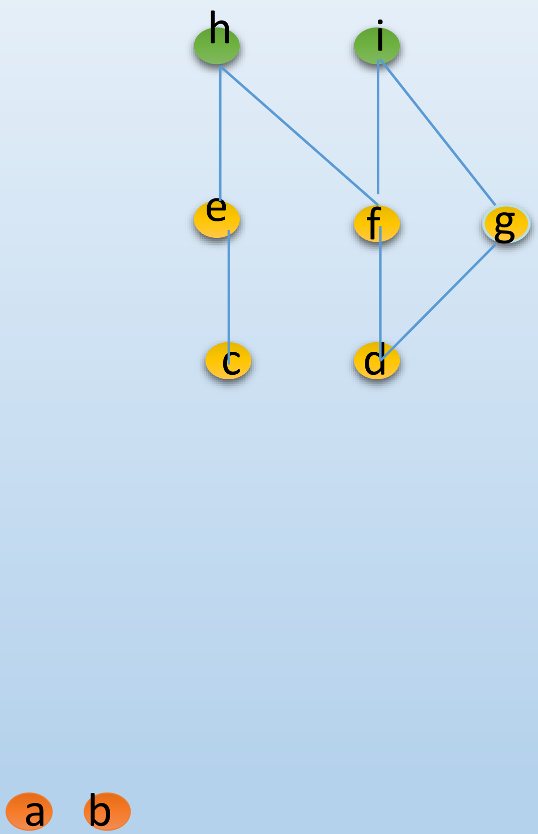
Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

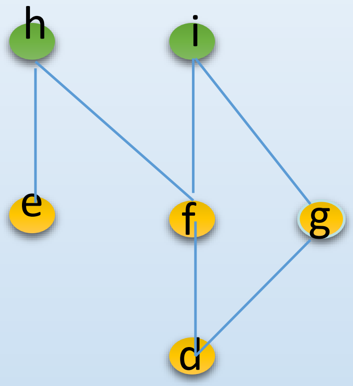
Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

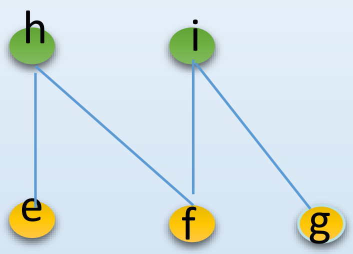
Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

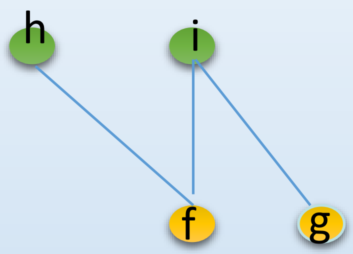
Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

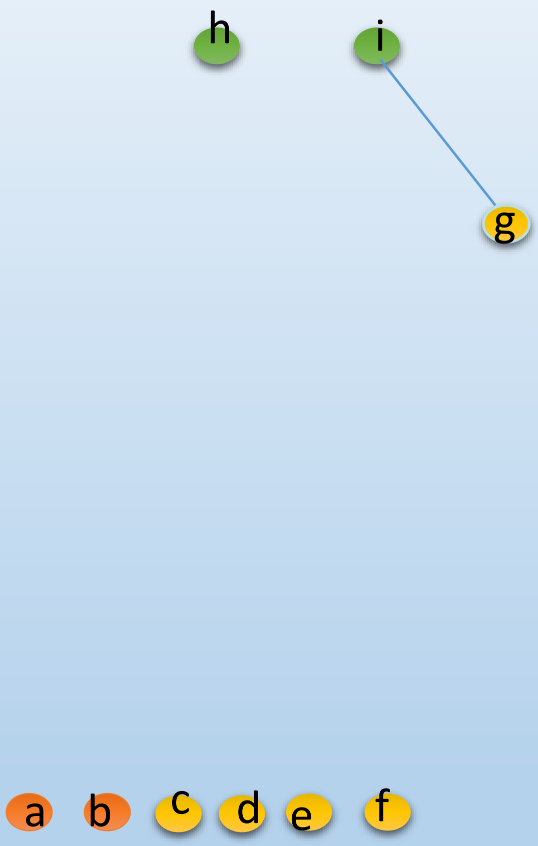
Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

Posets = *partially* ordered sets



Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

Posets = *partially* ordered sets

Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

Posets = *partially* ordered sets

Can always be extended (add relations)
Until the ordering is *linear*

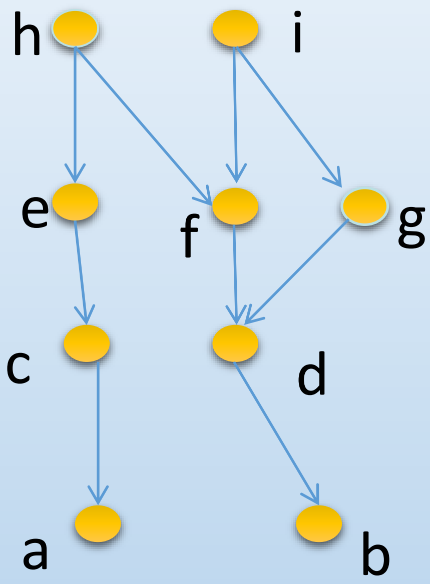
… equivalent to shelling the poset

Posets = *partially* ordered sets

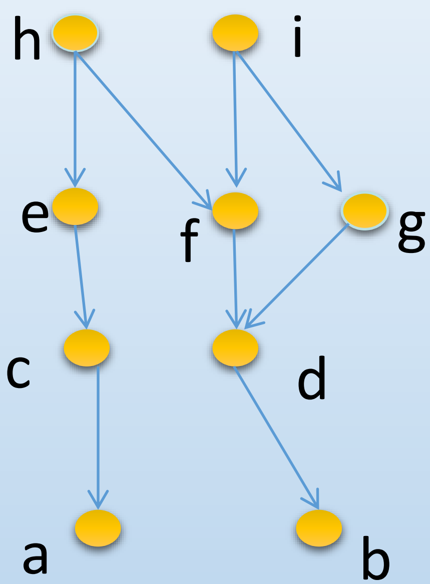
Can always be extended (add relations)
Until the ordering is *linear*

… equivalent to shelling the poset

Bumps in linear extensions

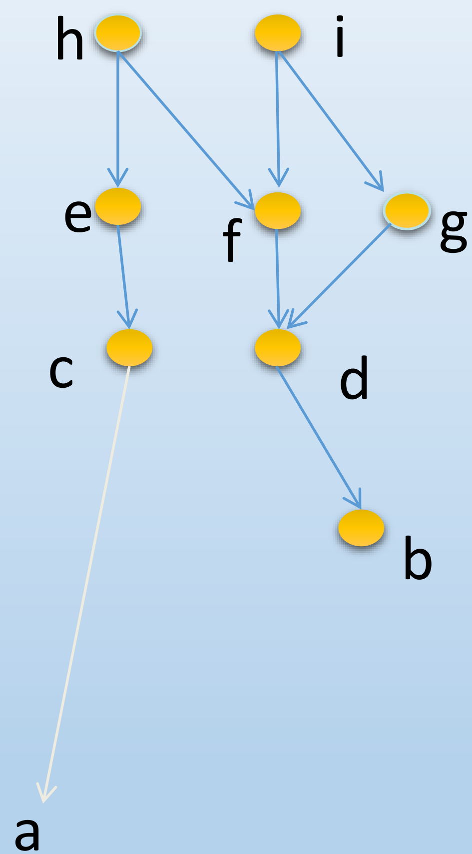


Bumps in linear extensions



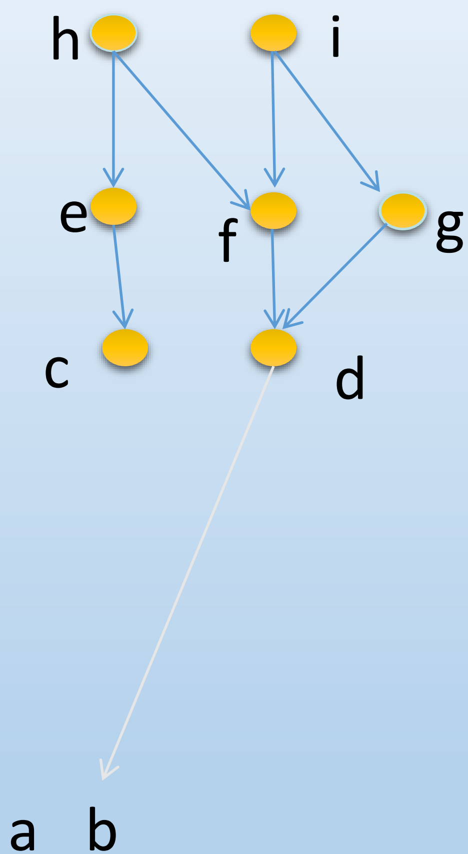
Linear extension (showing bumps)

Bumps in linear extensions



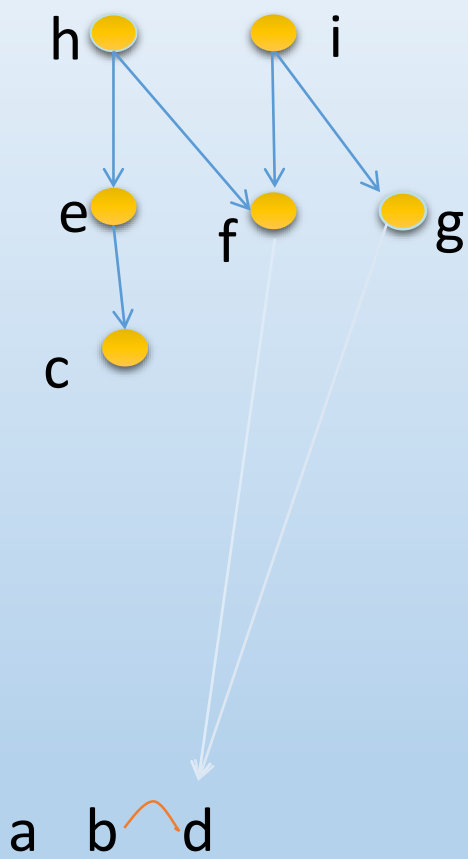
Linear extension (showing bumps)

Bumps in linear extensions



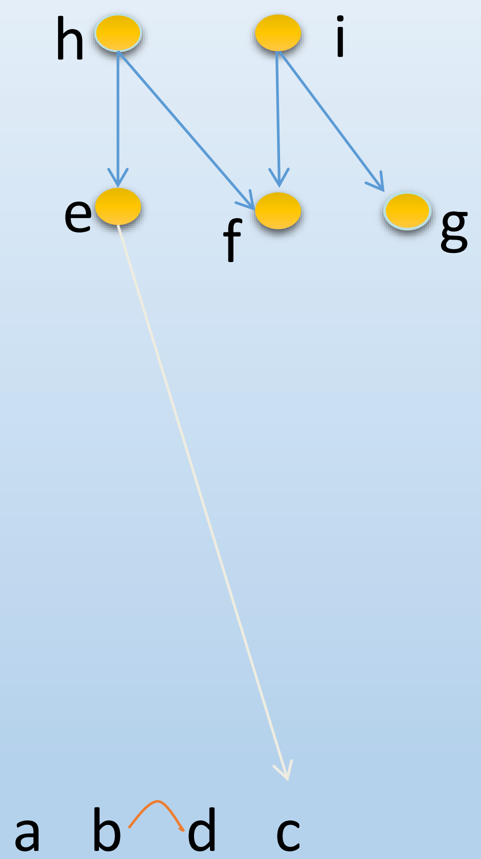
Linear extension (showing bumps)

Bumps in linear extensions



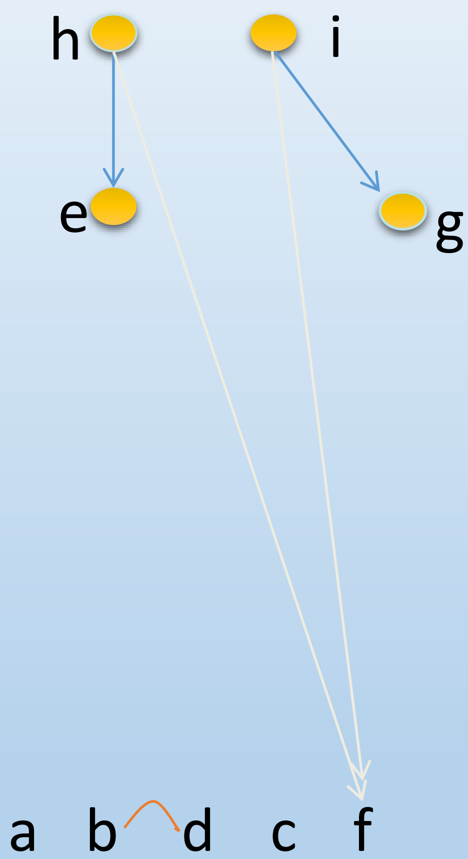
Linear extension (showing bumps)

Bumps in linear extensions



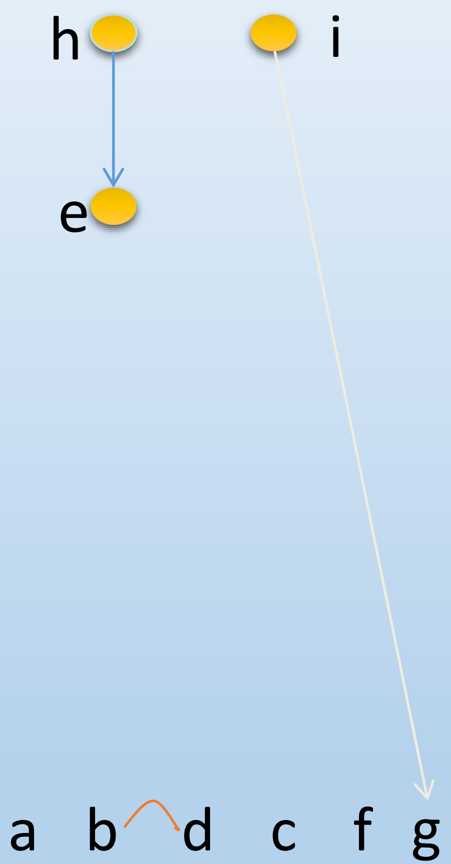
Linear extension (showing bumps)

Bumps in linear extensions



Linear extension (showing bumps)

Bumps in linear extensions



Linear extension (showing bumps)

Bumps in linear extensions

Linear extension (showing bumps)

Bumps in linear extensions

h

Linear extension (showing bumps)

a b d c f g e i

Bumps in linear extensions

Linear extension (showing bumps)

a b d c f g e i h

Bump Number Problem

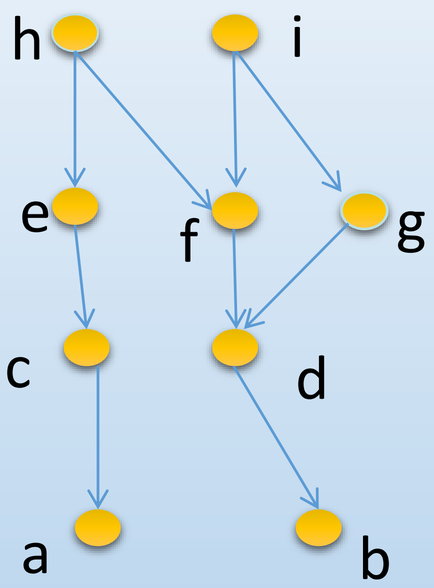
Given poset P, what is the least number of bumps realized by a linear extension of P?

b(P)= bump# of P

Find an algorithm to compute b(P) and construct a linear extension with fewest bumps

a b d c f g e i h

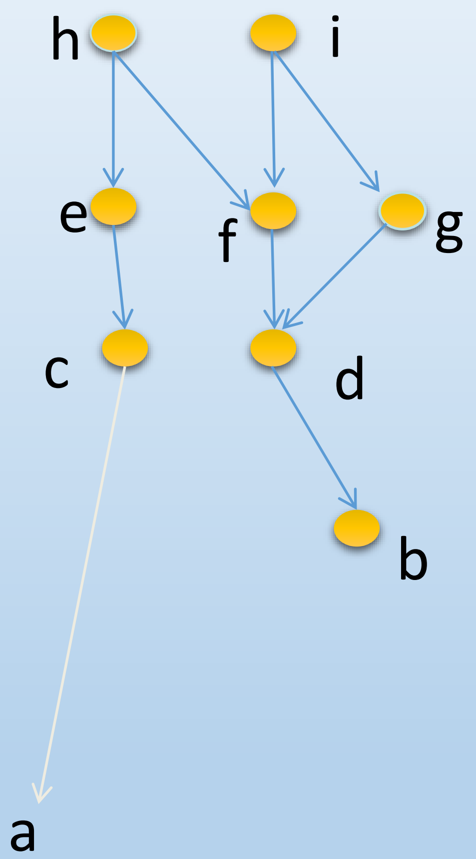
Greedily seeking min-bump l.e.



Linear extension (showing bumps)

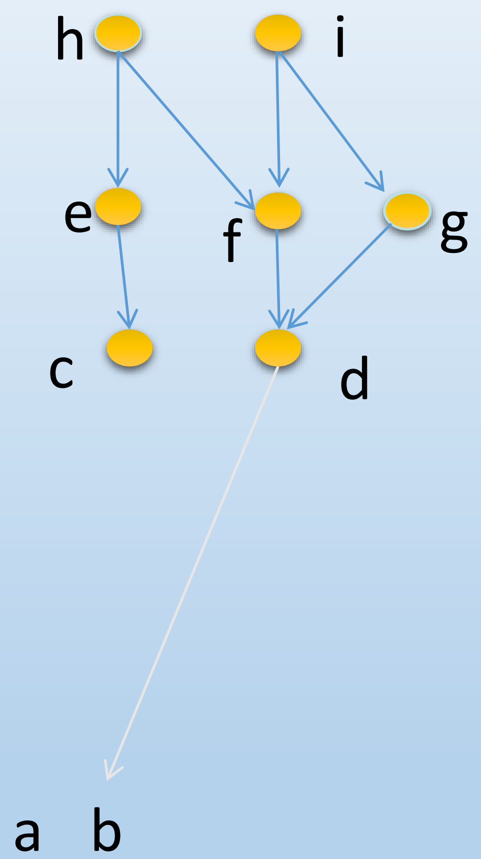
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



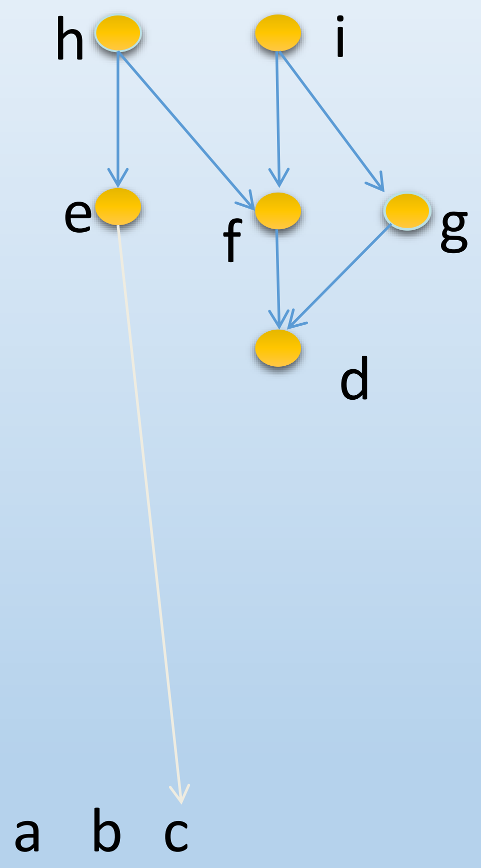
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



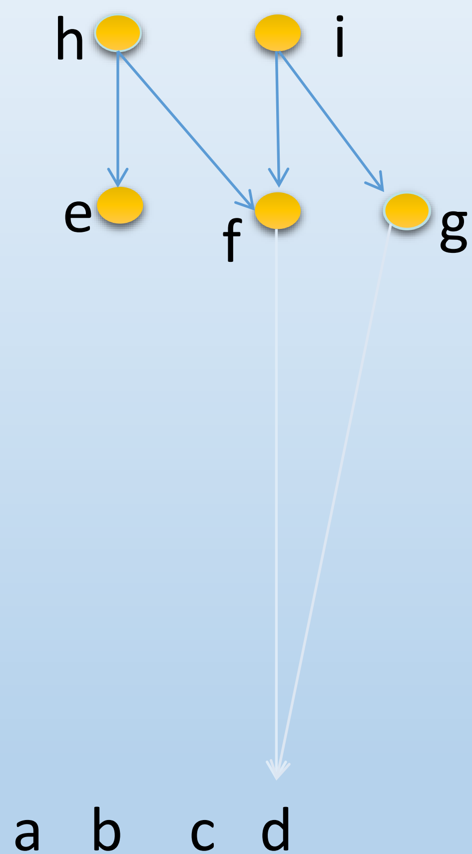
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



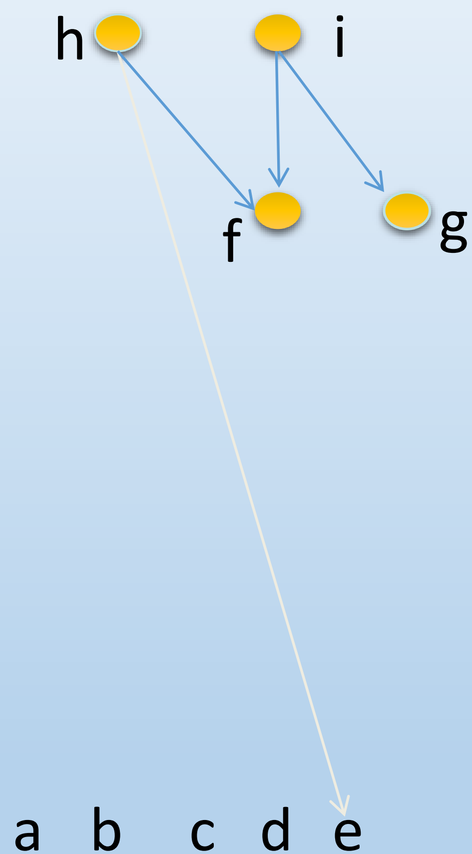
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



Linear extension (showing bumps)

Greedily selecting to avoid bumps

Greedily seeking min-bump l.e

Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e

Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e

i

Linear extension (showing bumps)
Greedily selecting to avoid bumps

a b c d e f g h

Greedily seeking min-bump l.e

Linear extension (showing bumps)
Greedily selecting to avoid bumps

a b c d e f g h i

Greedily seeking min-bump l.e

There is always some greedy l.e. that achieves minimum bump (Fishburn & Gehrlein, ‘86).

For which posets does greedy always work?

Greedily seeking min-bump l.e

There is always some greedy l.e. that achieves minimum bump (Fishburn & Gehrlein, ‘86).

For which posets does greedy always work?

Greedy + ? works for all posets?

Greedily seeking min-bump l.e

There is always some greedy l.e. that achieves minimum bump (Fishburn & Gehrlein, ‘86).

For which posets does greedy always work? F&G’86

Greedy + ? works for all posets?

Greedily seeking min-bump l.e

There is always some greedy l.e. that achieves minimum bump (Fishburn & Gehrlein, '86).

For which posets does greedy always work? F&G'86

Greedy + ? works for all posets? This talk

Bump Number

- polynomial algorithms for interval order posets and for partial semiorder posets – both are based on the greedy shelling algorithms
  Fishburn and Gehrlein 1986

- polynomial algorithm for width=2 posets – not based on greedy shelling
  Zaguia 1987

- polynomial algorithm for any poset – not based on shelling
  Habib, Möhring, Steiner 1988

- linear time algorithm – based on Gabow's linear time 2-proc scheduling algorithm
  Schäffer & Simons 1988

Linear Time Bump Number

Schäffer & Simons 1988

relies on Gabow and Tarjan's special case Union-Find algorithm: union and find operations known in advance

O(n+m) in the cover graph

… relies on hybrid linked-list / array data structure ... Switch to array representation of tree for subtrees that are small enough…

Algorithm, proof of correctness, and analysis

- Spread across several papers
- Proofs long and case-ridden
- Analysis complex

Question:

a simple algorithm

$\exists$ with a short proof

that can be made efficient (linear time) without recourse to Special Case of Union-Find?

Algorithm, proof of correctness, and analysis

- Spread across several papers
- Proofs long and case-ridden
- Analysis complex

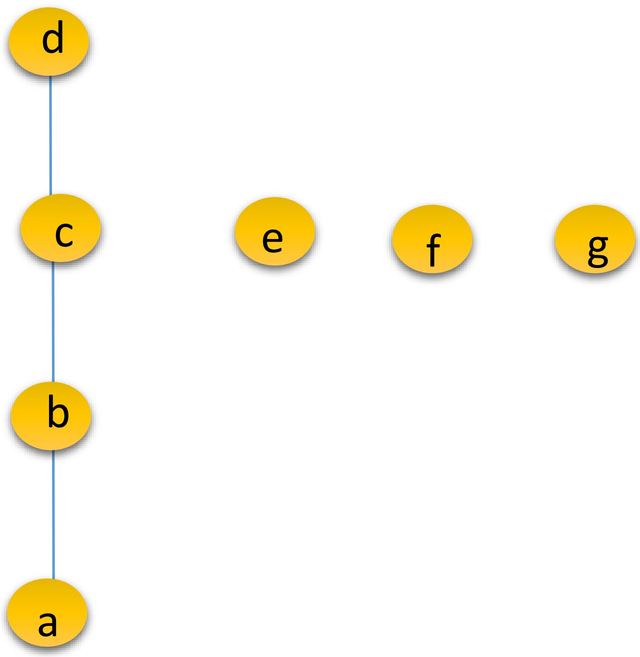
Question:

a simple algorithm YES

$\exists$ with a short proof YES

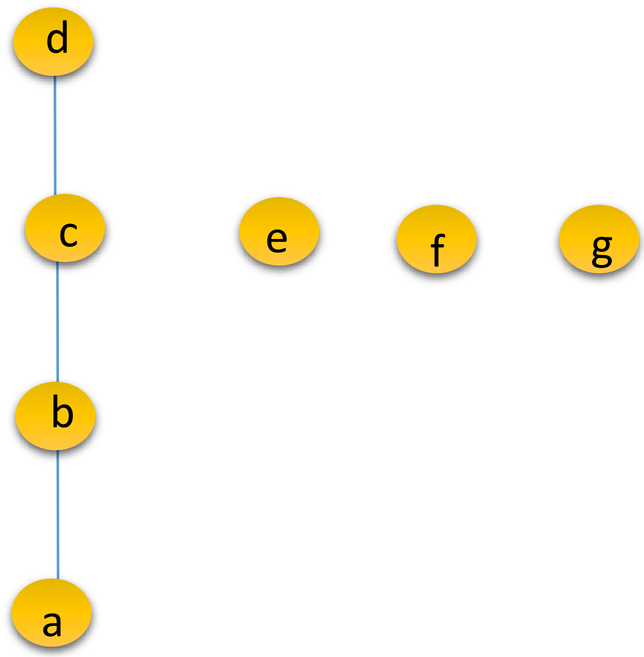
that can be made efficient (linear time) without recourse to Special Case of Union-Find? Open.

When Greedy is not enough

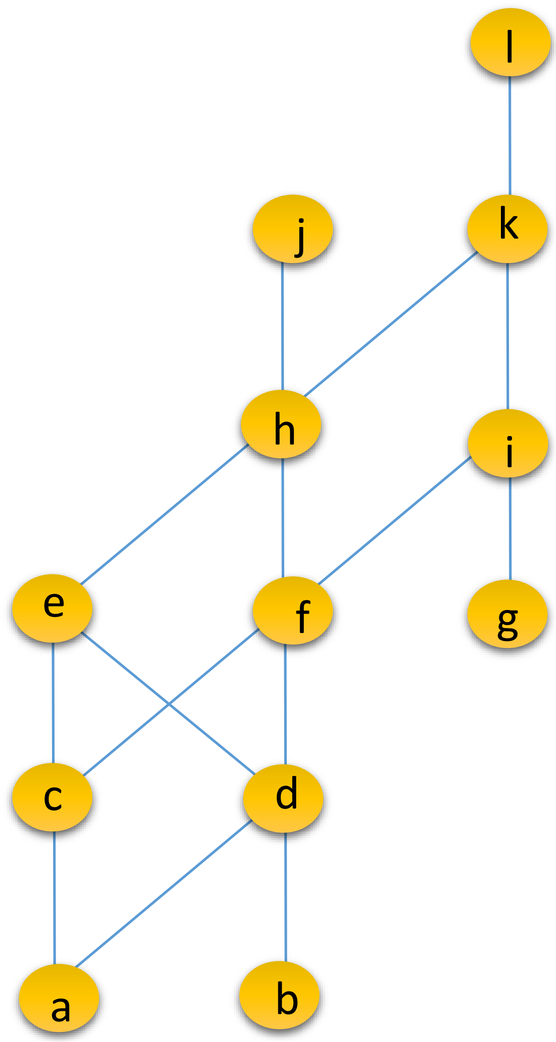


When Greedy is not enough

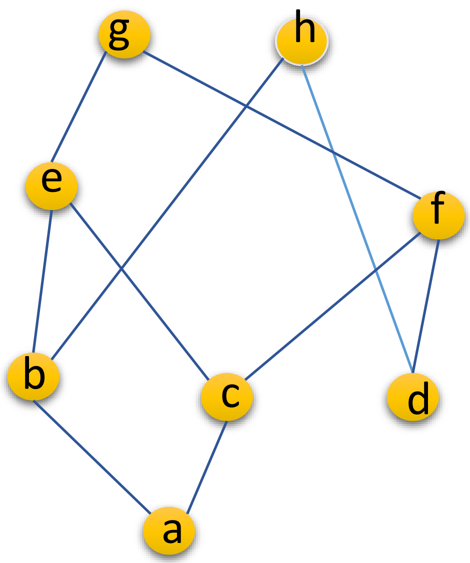
e a f b g c d



When Greedy is not enough

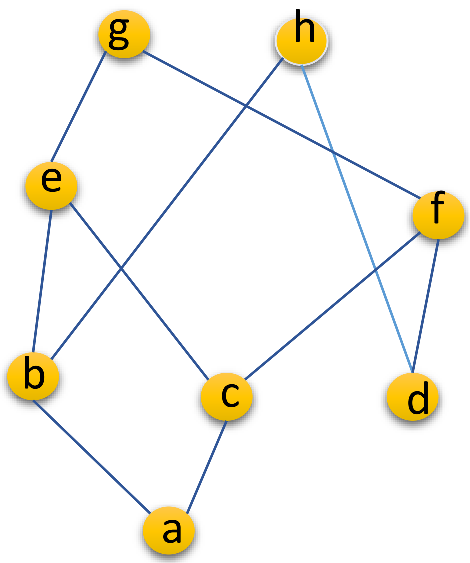


Greedy bump#



Greedy Approach

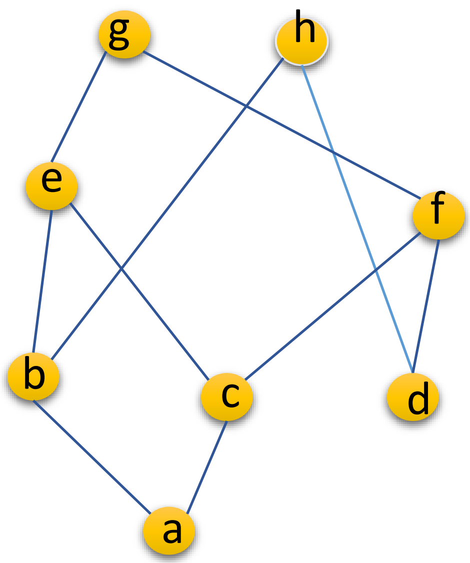
Greedy bump#



Greedy Approach

d a ... oops

Greedy bump#

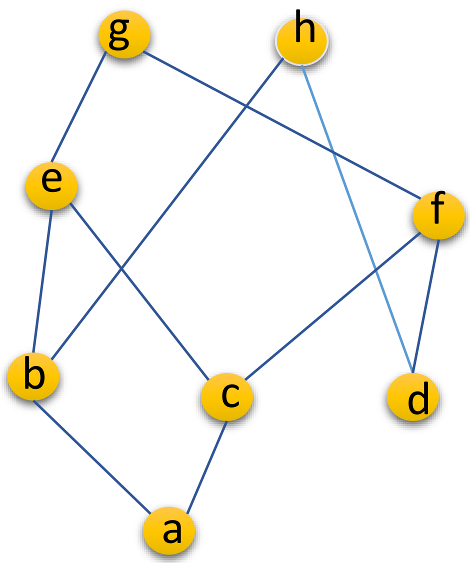


Greedy Approach

d a ... oops

a d b c h e f ...oops

Greedy bump#



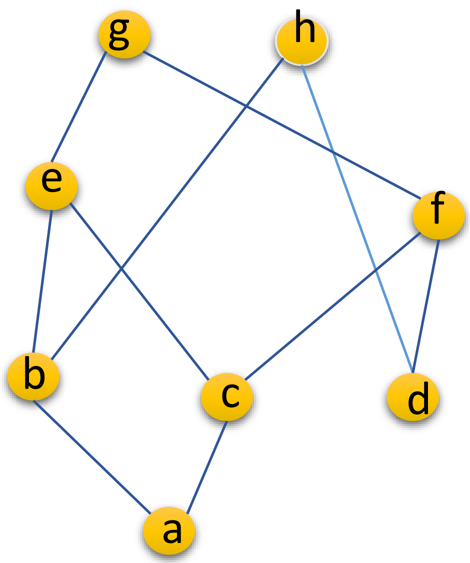
Greedy Approach

d a ... oops

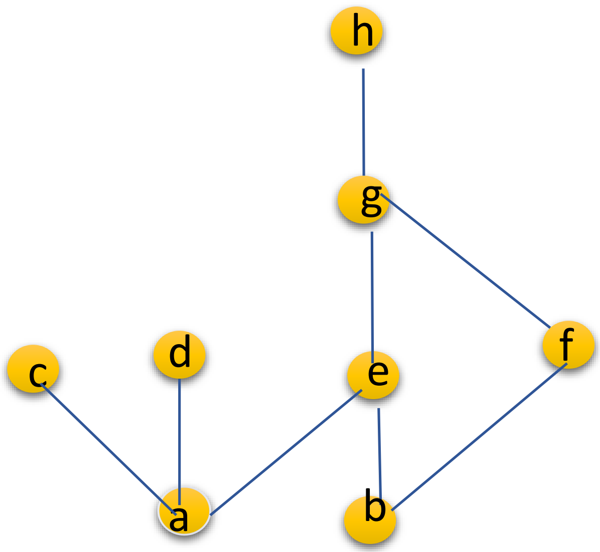
a d b c h e f ...oops

a d c b f e h g

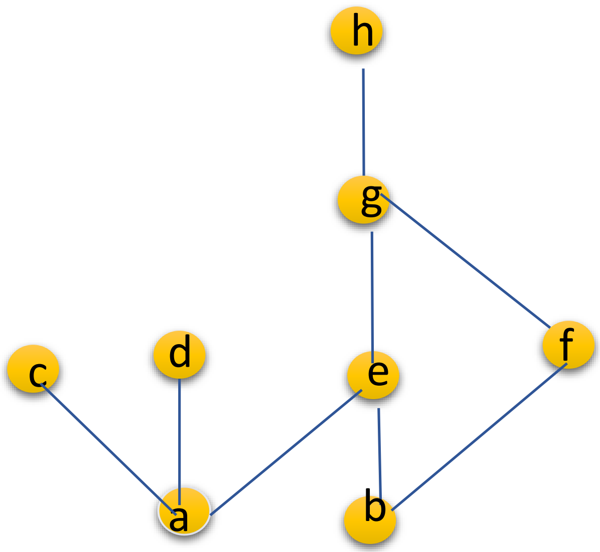
Take the min at bottom of longest chain?
Take the min with largest upper ideal?



Take the min at bottom of longest chain?
Take the min with largest upper ideal?

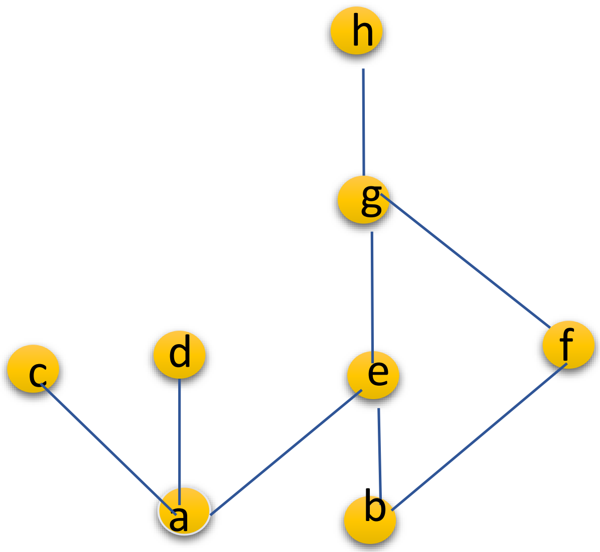


Take the min at bottom of longest chain?
Take the min with largest upper ideal?



a b c e f d g h

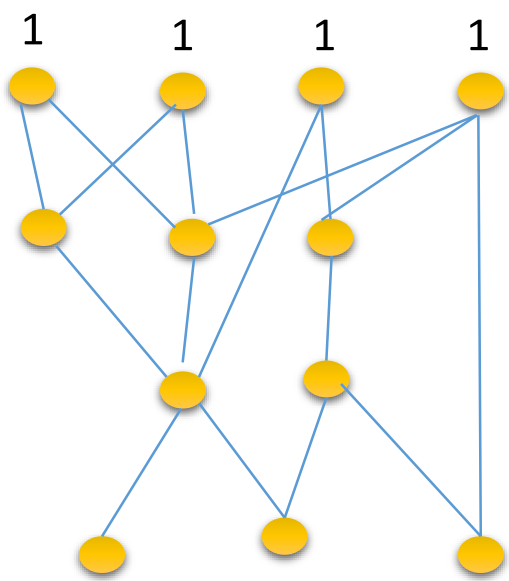
Take the min at bottom of longest chain?
Take the min with largest upper ideal?



a b c e f d g h

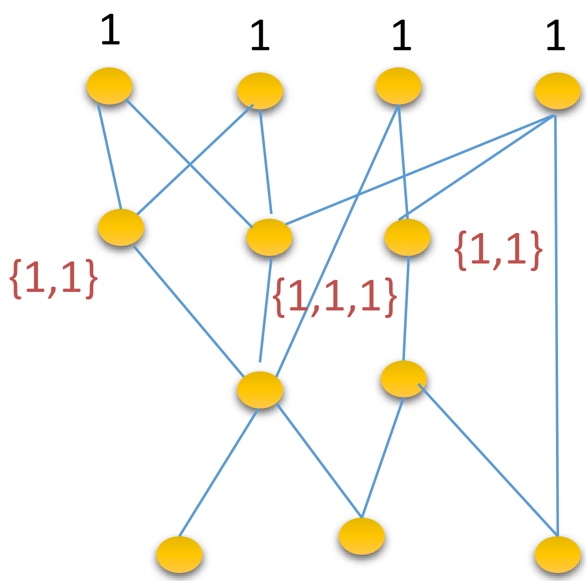
b a f e c g d h

Lexicographic Labelling



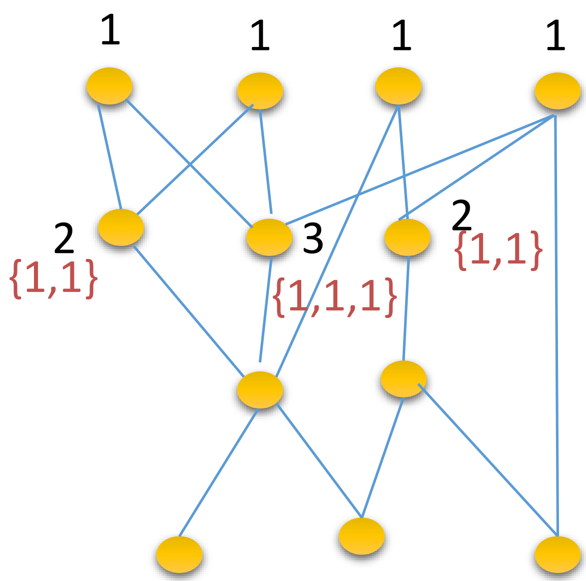
- Give minima arbitrary lex#

Lexicographic Labelling



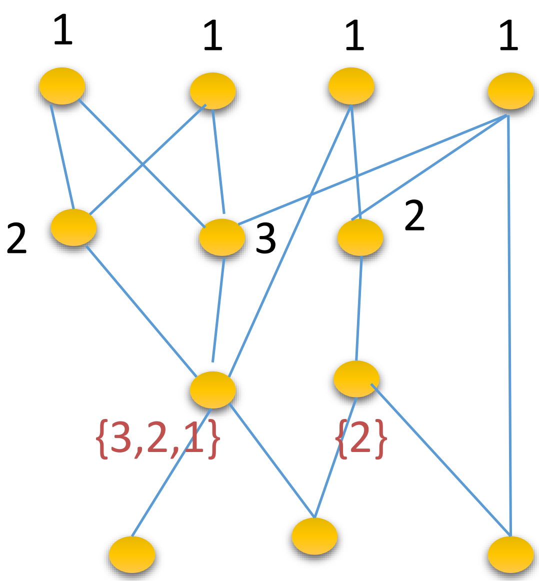
- Give maxima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



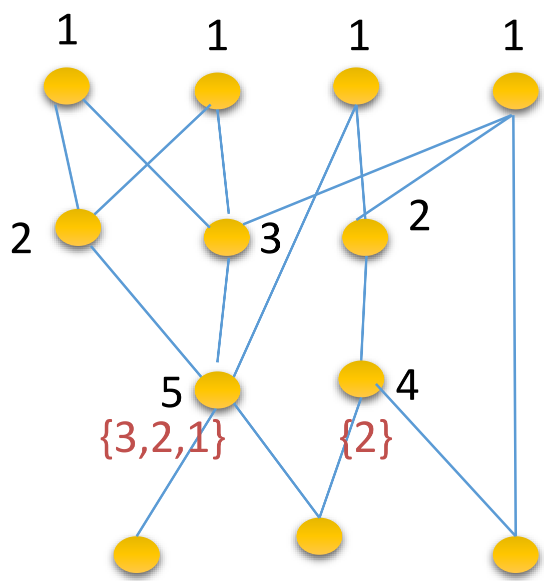
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



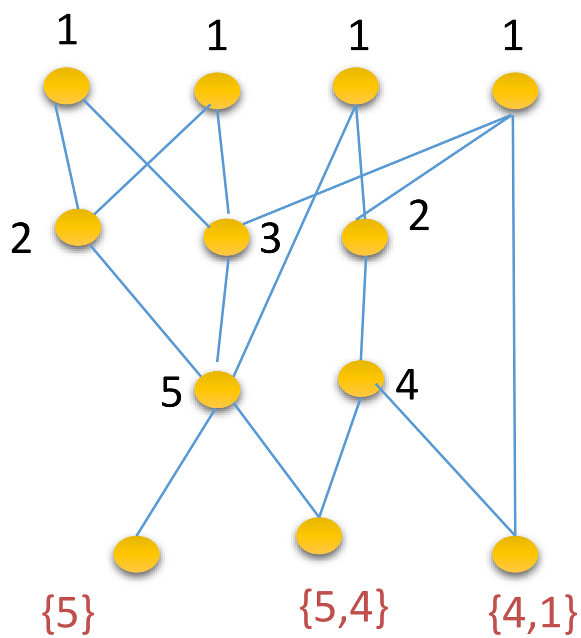
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{\text{lexico}}$ {lex(v'): v' covers v}

Lexicographic Labelling



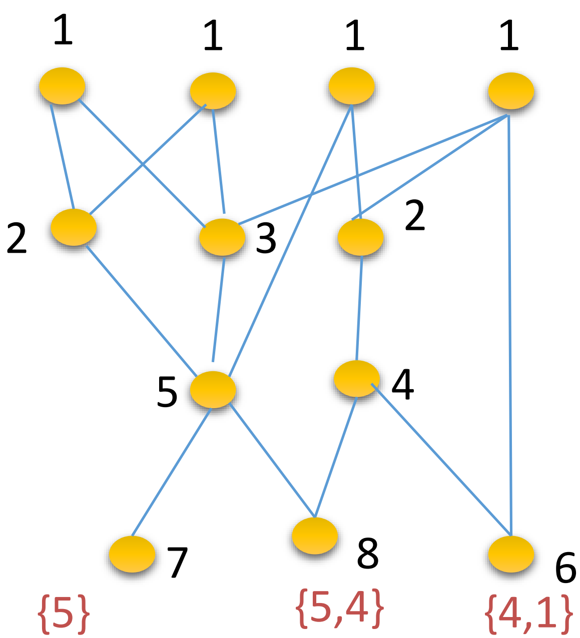
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



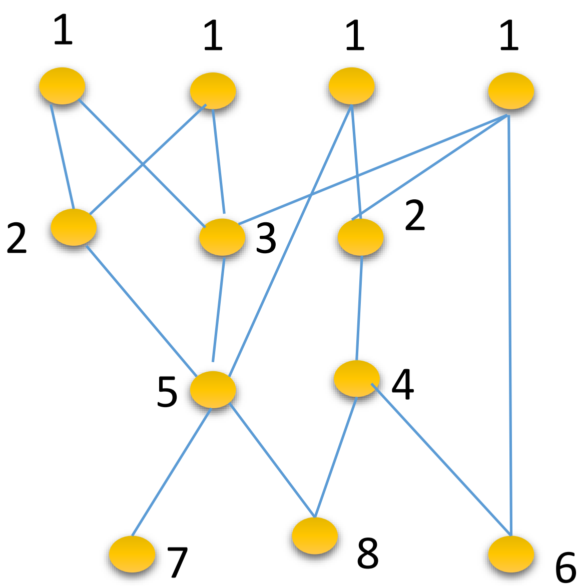
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u’): u’covers u} $<_{lexico}$ {lex(v’): v’ covers v}

Lexicographic Labelling



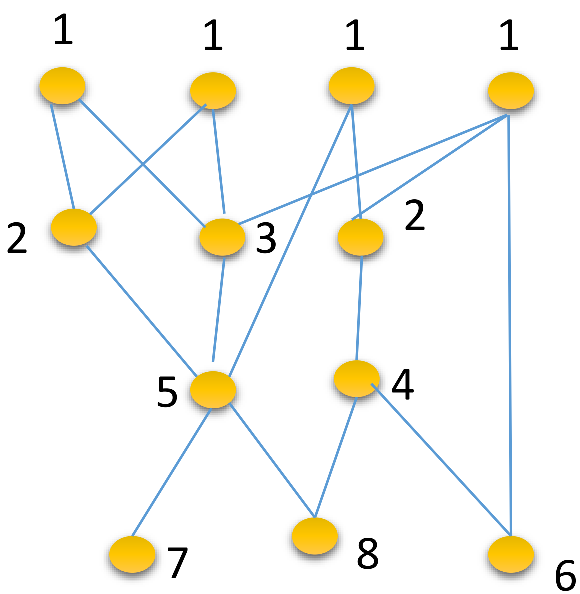
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

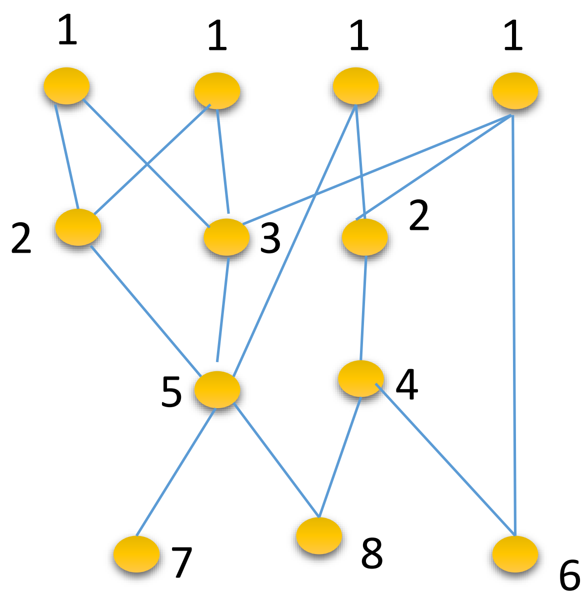
Lexicographic Labelling



New: O(n+m) algorithm for lex-labelling

(Sethi 1976 algorithm also achieves linear time)

Lexicographic Labelling

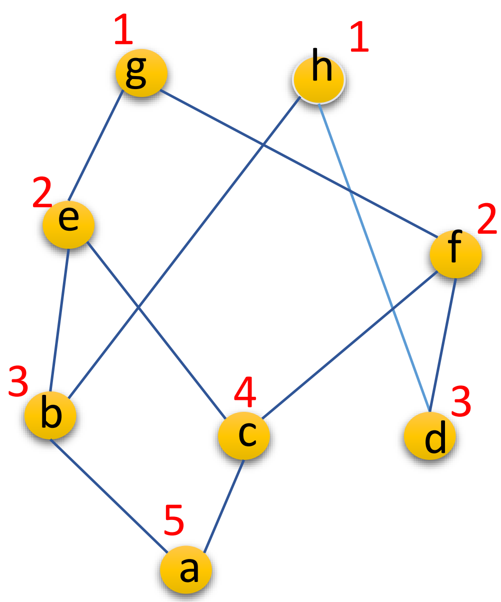


New: O(n+m) algorithm for lex-labelling

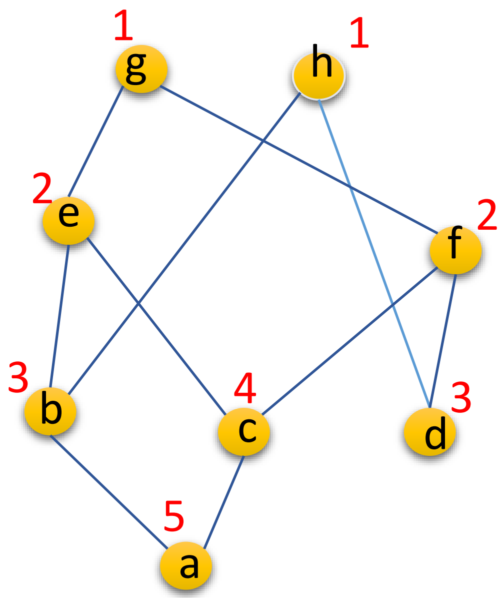
(Sethi 1976 algorithm also achieves linear time)

… or does it?

Greedy + Lex = Greedlex

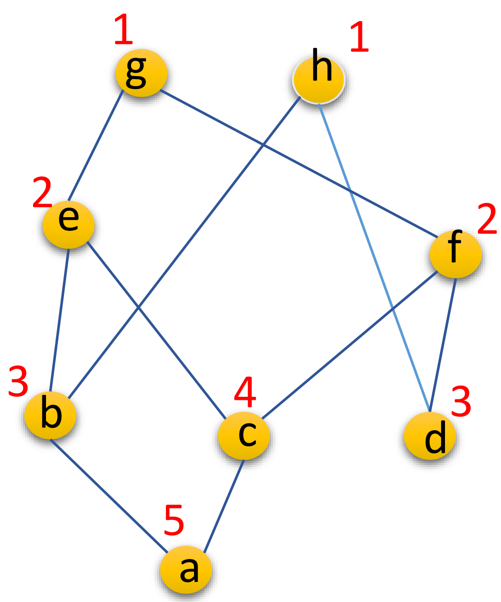


Greedy + Lex = Greedlex



c has a higher lex-number than b

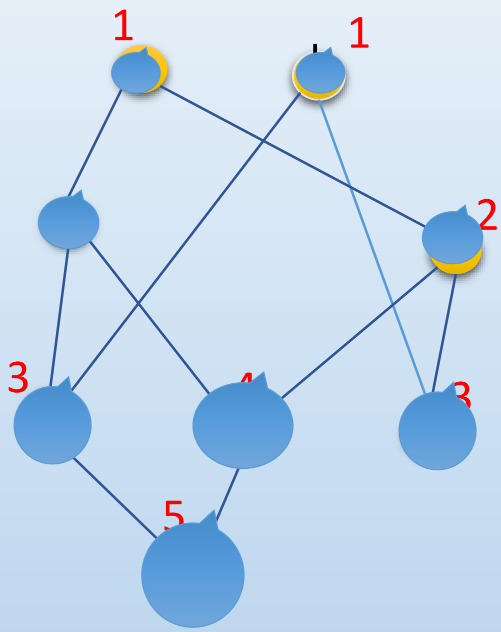
Greedy + Lex = Greedlex



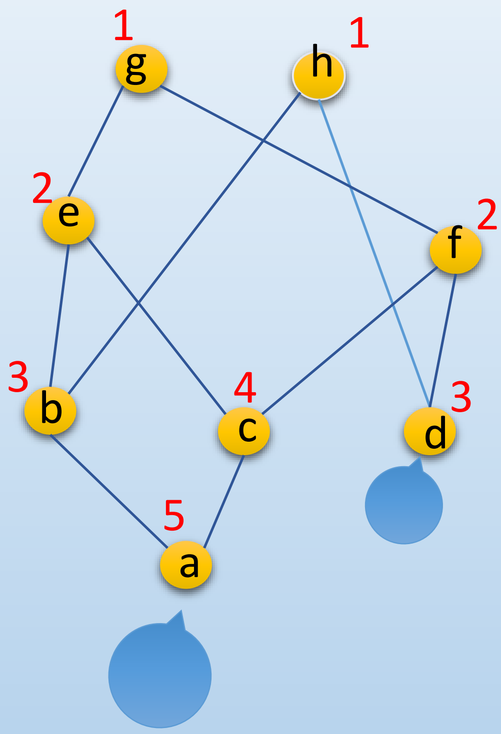
c has a higher lex-number than b

… therefore c has an upper cover that b doesn't have, of high lex number (i.e., a private cover w.r.t. c)

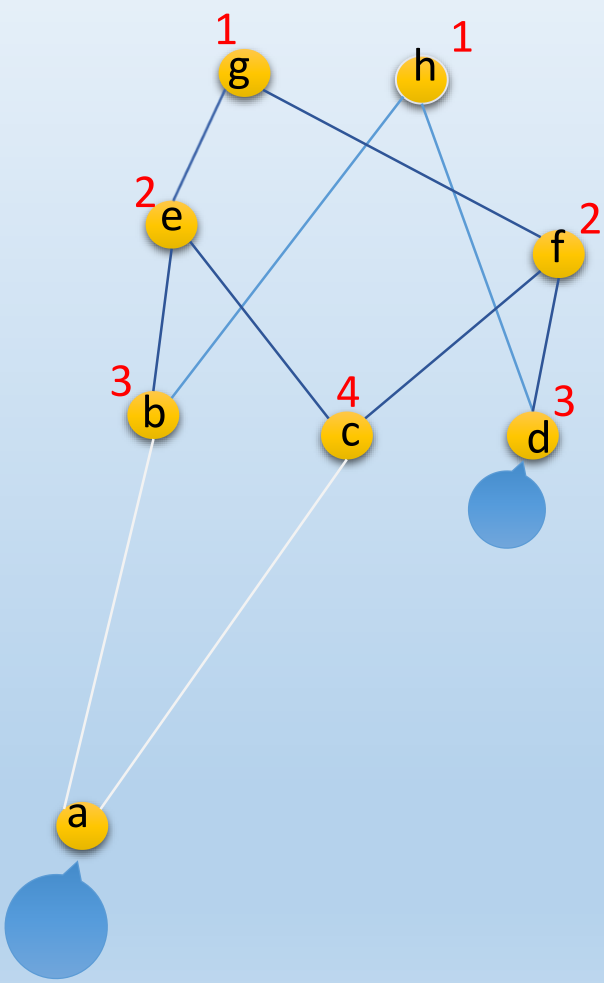
Greedy + Lex = Greedlex



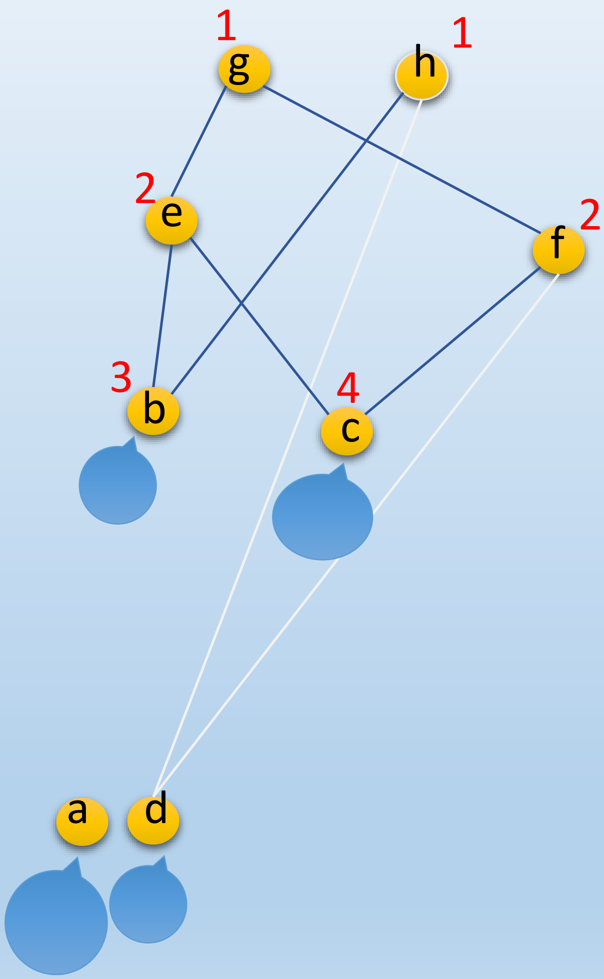
Greedy + Lex = Greedlex



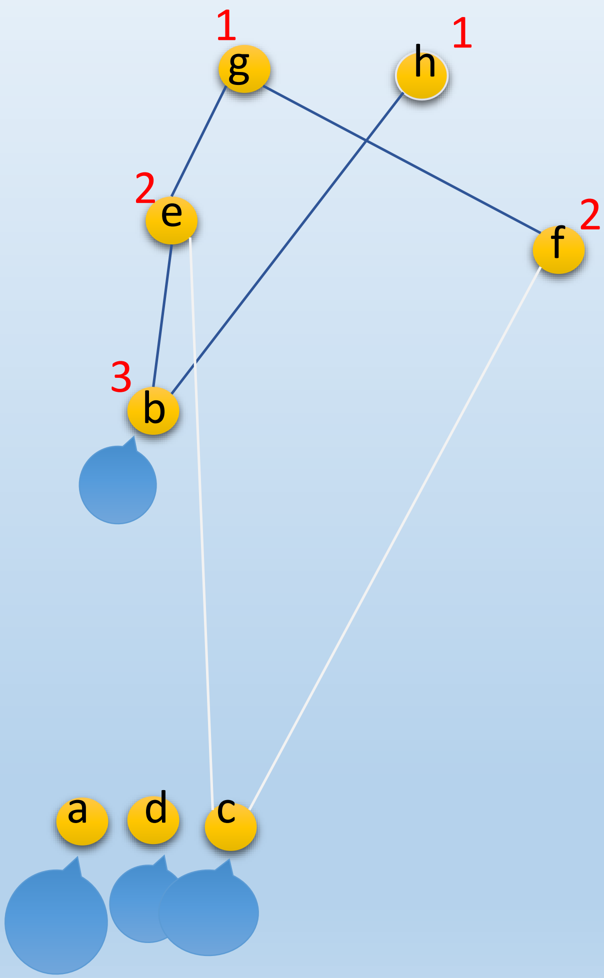
Greedy + Lex = Greedlex



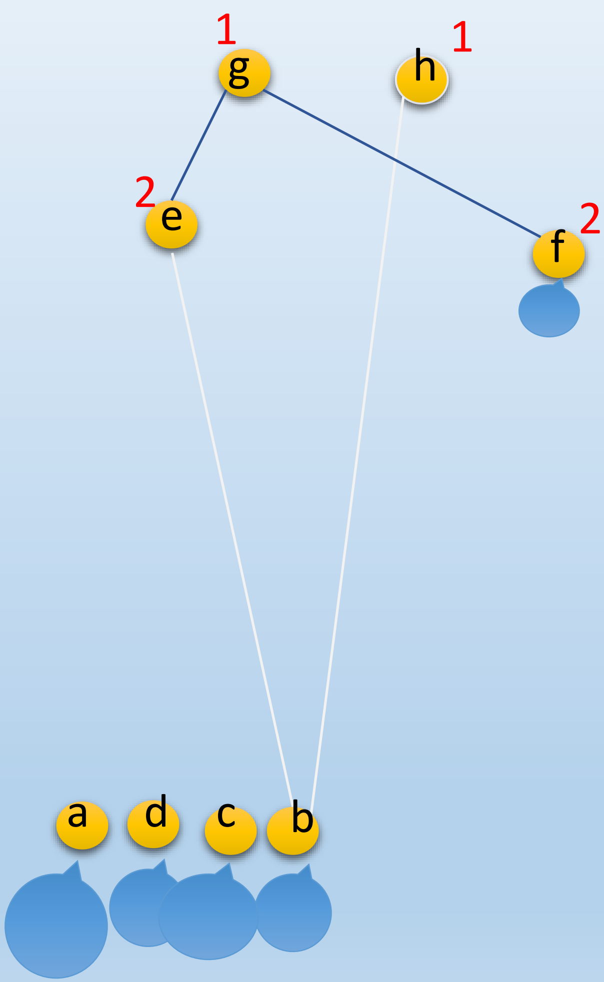
Greedy + Lex = Greedlex



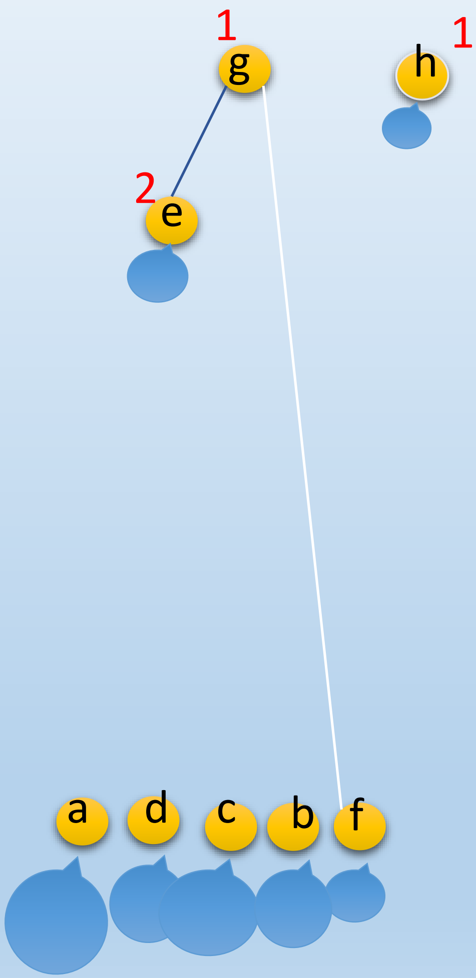
Greedy + Lex = Greedlex



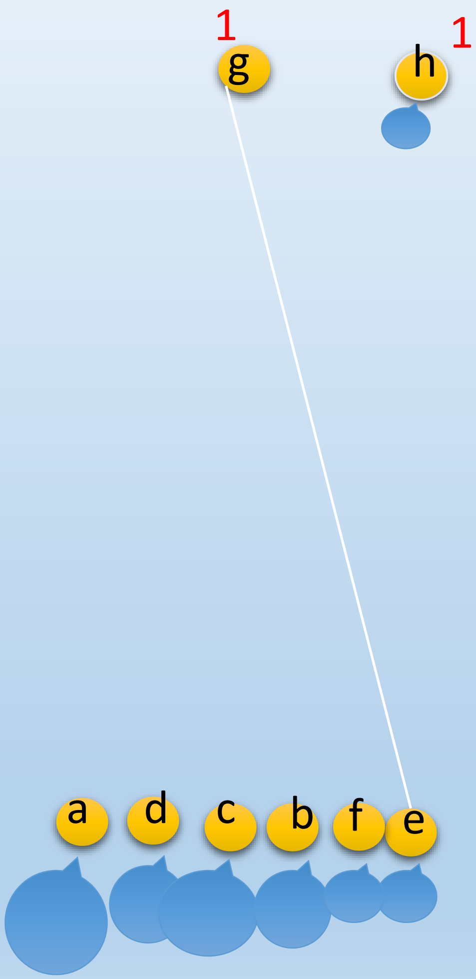
Greedy + Lex = Greedlex



Greedy + Lex = Greedlex

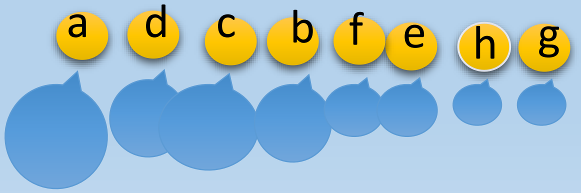


Greedy + Lex = Greedlex



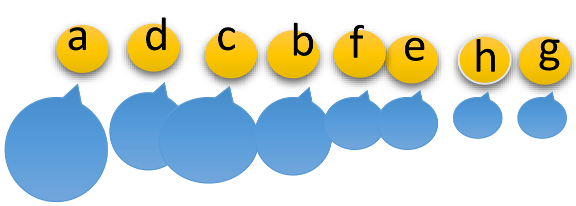
Greedy + Lex = Greedlex

Greedy + Lex = Greedlex



Greedy + Lex = Greedlex

- Label the elements lexicographically
- Shell greedily (avoiding bumps locally) and take the lexicographically largest when given a choice



Greedy + Lex = Greedlex

- Label the elements lexicographically
- Shell greedily (avoiding bumps locally) and take the lexicographically largest when given a choice

Claim: This always yields the min-bump l.e.

Lex-Yanking Lemma

b xxx..x a xx...x has k bumps and a is min

Lex-Yanking Lemma

b xxx..x a xx…x has k bumps and a is min

⇒ ∃ l.e. a x’x’x’…b…x’ with k or fewer bumps

(Balloon size indicates relative Lex value)

Lex-Yanking Lemma

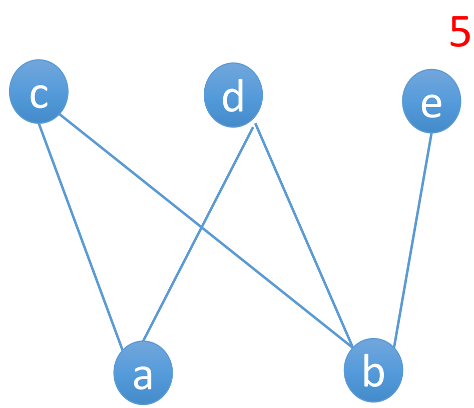
b xxx..x a xx…x has k bumps and a is min

⇒ ∃ l.e. a x’x’x’…b…x’ with k or fewer bumps

(Balloon size indicates relative Lex value)

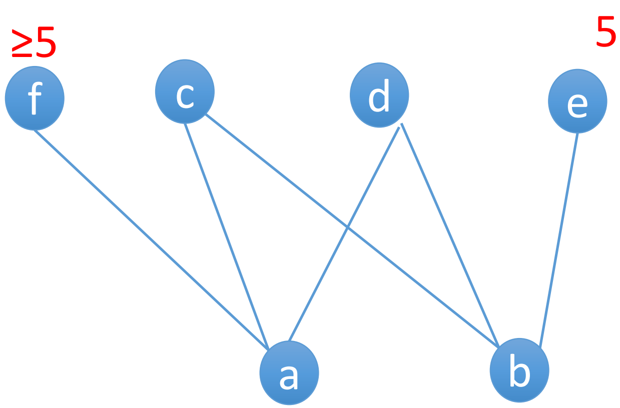
Claim: If the lex-yanking lemma holds on a poset and all its induced subposets, then Greedlex produces a bump-optimal linear extension.

Proof of LexYanking Lemma



If lex(a)≥lex(b) and b has a private cover (not covering a)…

Proof of LexYanking Lemma



If lex(a)≥lex(b) and b has a private cover (not covering a)…

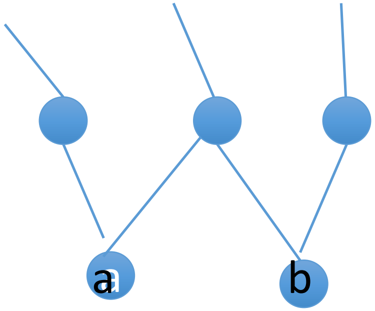
Then a has a private cover with lex# at least as large.

Proof of LexYanking Lemma

By induction on n=|V(P)|. Base cases n=0,1 are trivial.

Let P be a poset on n>1 elements, and suppose LexYanking Lemma holds for all smaller posets. (Then also Greedlex works on smaller posets.)

b xxx a xxxx a l.e. with k bumps, lex(a)≥lex(b), a and b min



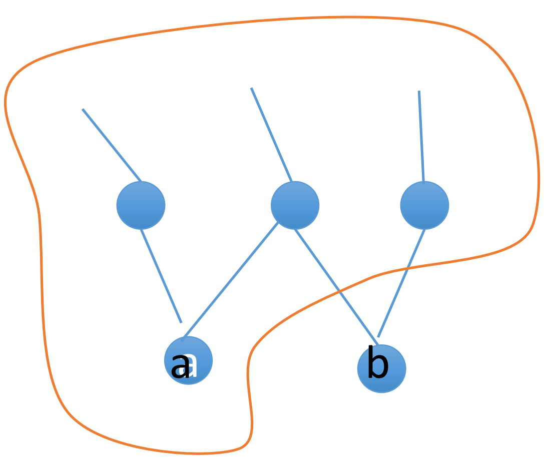
Proof of LexYanking Lemma

b xxx a xxxx   a l.e. with k bumps, lex(a)≥lex(b), a and b min

The poset \ {b} is smaller, so by Ind. Hyp., LexYanking holds, and Greedlex produces a min-bump suffix to follow b

b yyy a yyyy

All these elements have lex# > lex(a)



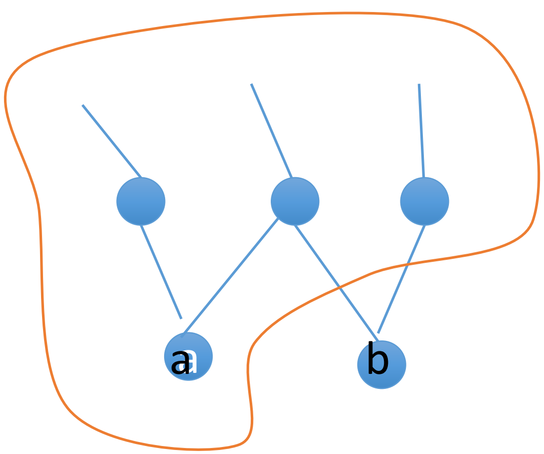
Proof of LexYanking Lemma

b xxx a xxxx a l.e. with k bumps, lex(a)≥lex(b), a and b min

The poset \ {b} is smaller, so by Ind. Hyp., LexYanking holds, and Greedlex produces a min-bump suffix to follow b

b yyy a yyyy

All these elements have lex# > lex(a) ≥ lex(b)

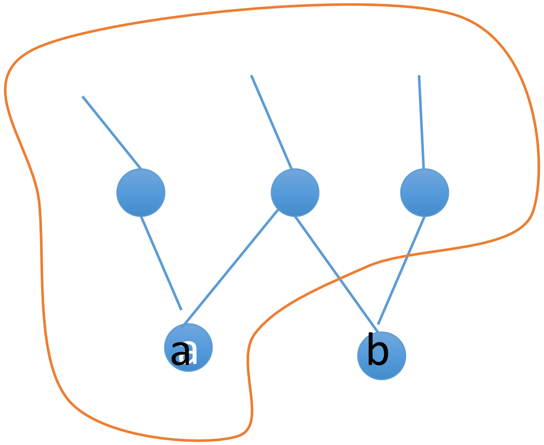


Proof of LexYanking Lemma

b xxx a xxxx   a l.e. with k bumps, lex(a)≥lex(b), a and b min

The poset \ {b} is smaller, so by Ind. Hyp., LexYanking holds, and Greedlex produces a min-bump suffix to follow b

b yyy a yyyy



All these elements have lex# > lex(a) ≥ lex(b)
Hence all are incomparable with b
They are also incomparable with a

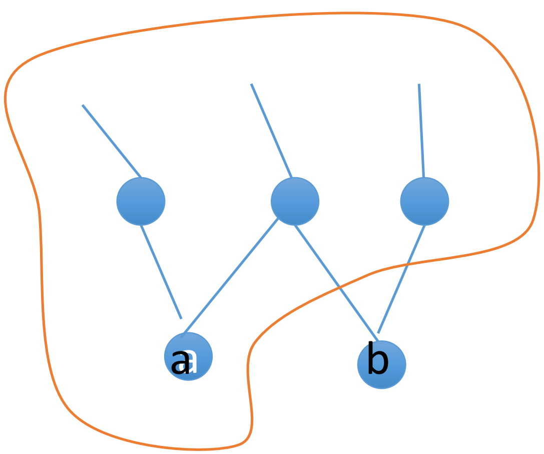
Swap: a yyy b yyyy

Proof of LexYanking Lemma

b xxx a xxxx   a l.e. with k bumps, lex(a)≥lex(b), a and b min

The poset \ {b} is smaller, so by Ind. Hyp., LexYanking holds, and Greedlex produces a min-bump suffix to follow b

b yyy a yyyy



All these elements have lex# > lex(a) ≥ lex(b)
Hence all are incomparable with b
They are also incomparable with a

Swap: a yyy b yyyy
May have introduced a bump

Proof of LexYanking Lemma

a yyy b yyyy

Suppose a bump was introduced after the b, and there was no such bump when a was in the same spot.

Proof of LexYanking Lemma

a yyy b $y_1$yyy

Suppose a bump was introduced after the b, and there was no such bump when a was in the analogous spot.

Then y_1 is a private cover of b (with respect to a).

Proof of LexYanking Lemma

$a\ yyy\ b\ y_1yyy$

Suppose a bump was introduced after the b, and there was no such bump when a was in the same spot.

Then y_1 is a private cover of b (with respect to a).

Then a has some private cover c (w.r.t. b), with lex(c)≥ lex(y_1).

$a\ yyy\ b\ y_1y\ldots c..y$

Proof of LexYanking Lemma

a yyy b $y_1$yyy

Suppose a bump was introduced after the b, and there was no such bump when a was in the same spot.

Then y_1 is a private neighbour of b (with respect to a).

Then a has some private neighbour c (w.r.t. b), with lex(c)≥ lex(y_1).

a yyy b y_1y…c..y

Then c can be yanked forward in the suffix, by the Ind. Hyp., without increasing bumps

a yyy b c z…y_1..z

Proof of LexYanking Lemma

a yyy b $y_1$yyy

Suppose a bump was introduced after the b, and there was no such bump when a was in the same spot.

Then y_1 is a private neighbour of b (with respect to a).

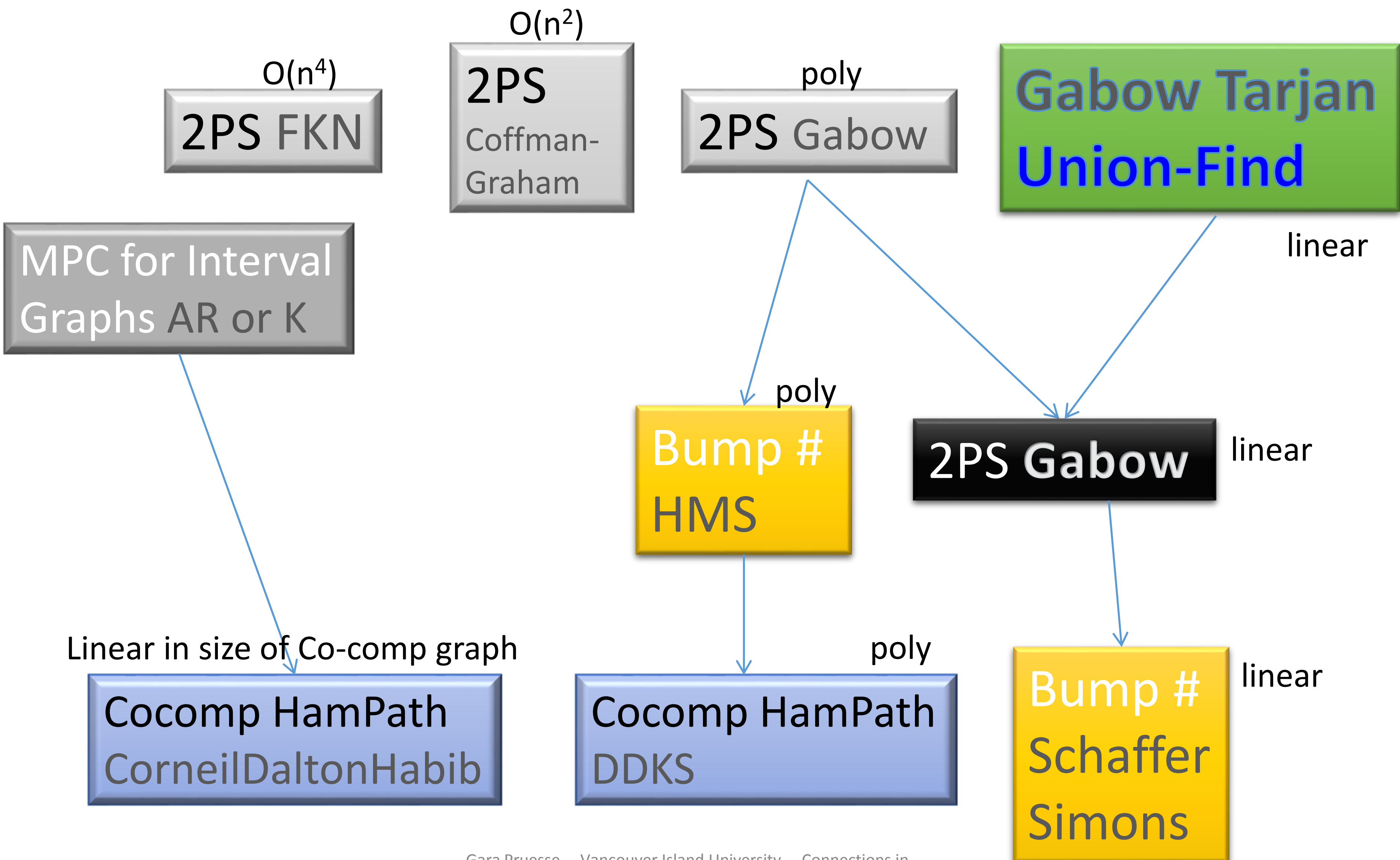
Then a has some private neighbour c (w.r.t. b), with lex(c)≥ lex(y_1).

a yyy b y_1y…c..y

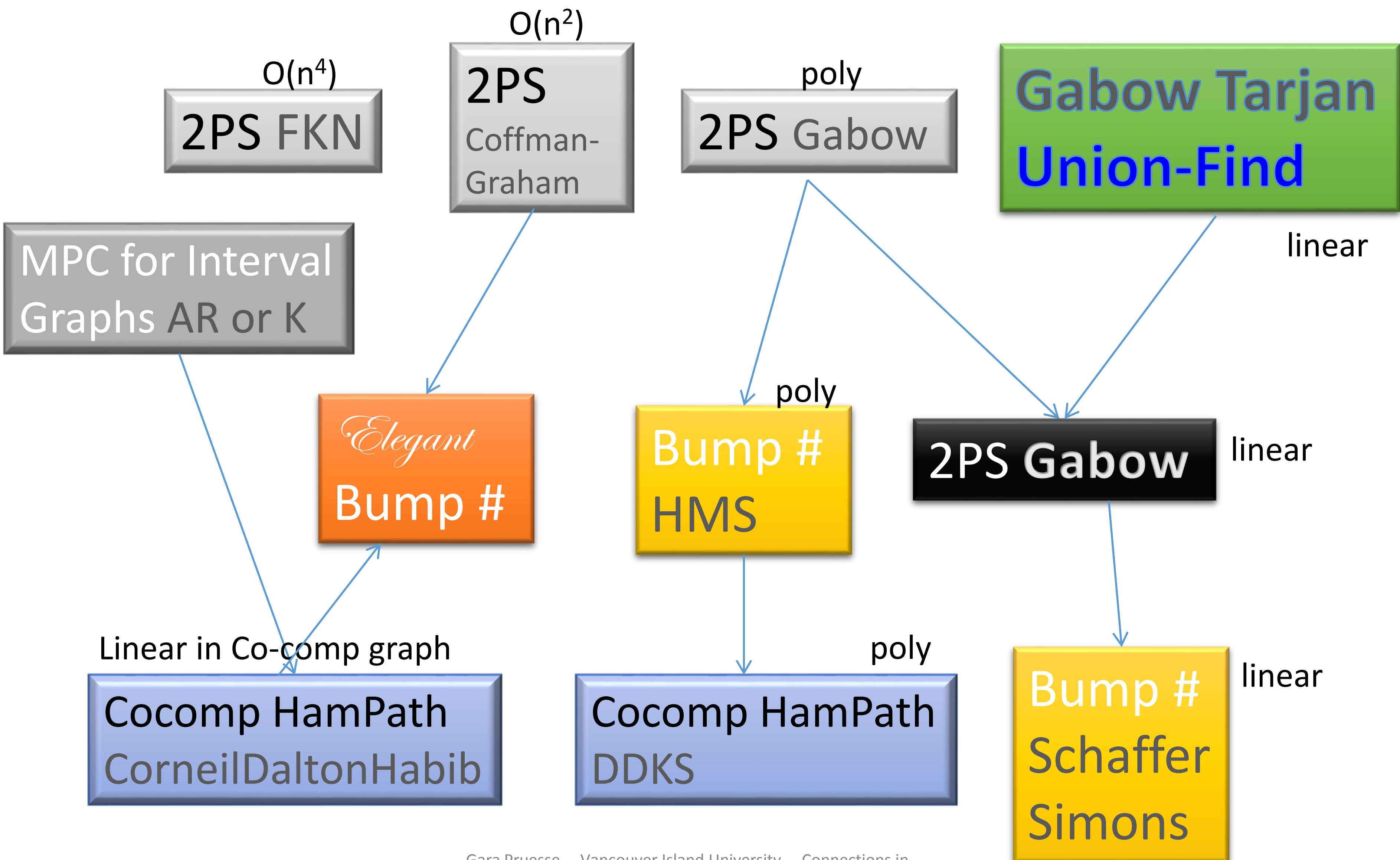
a yyy b c z…y_1..z

Then c can be yanked forward in the suffix, by the Ind. Hyp., without increasing bumps and destroying the bump after b.
[if c is not a min, take c's descendent]. ■

Where does this work fit in?



Where does this work fit in?



Where does this work fit in?

The beautiful idea ->

$O(n^2)$ 2PS
Coffman- Graham

Elegant
Bump #

Cocomp Path Cover
Corneil Dalton Habib

2PS Gabow
+
Gabow Tarjan
Union-Find

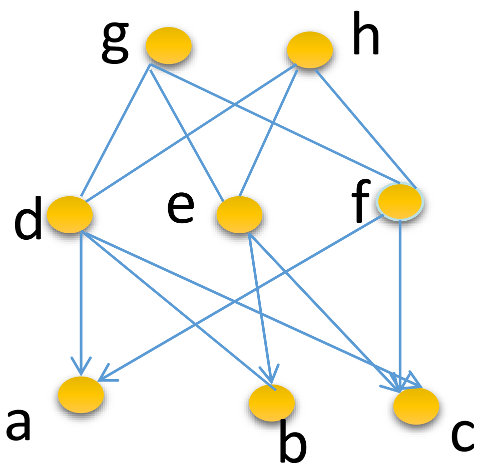
Linear 2PS
Gabow

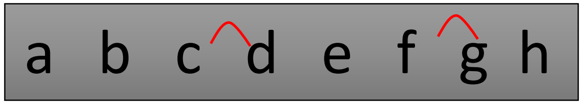
Linear
Bump #
Schaffer
Simons

How fast is the *Elegant* Algorithm?

- O(n log w)   w is size of maximum antichain
                n is size of the cover graph
- Can be quickly programmed

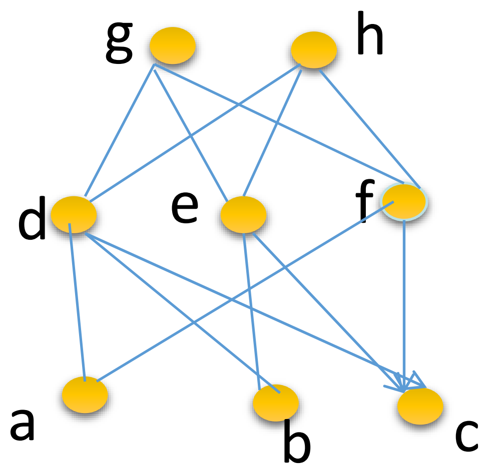
2-Processor Schedules

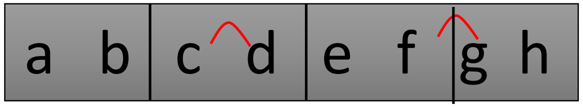




Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

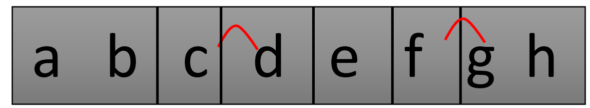
2-Processor Schedules

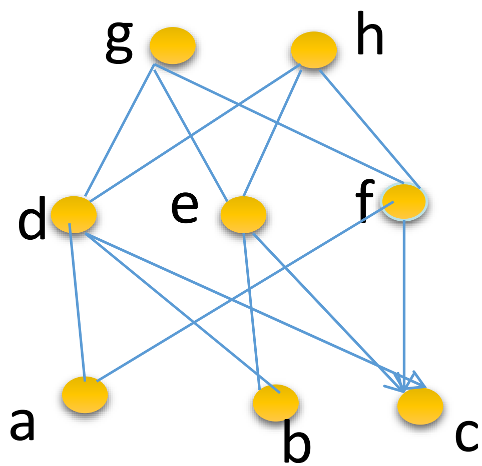




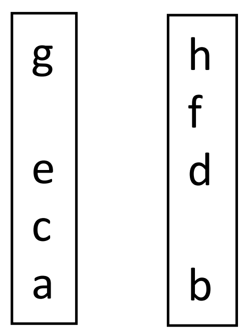
Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

2-Processor Schedules

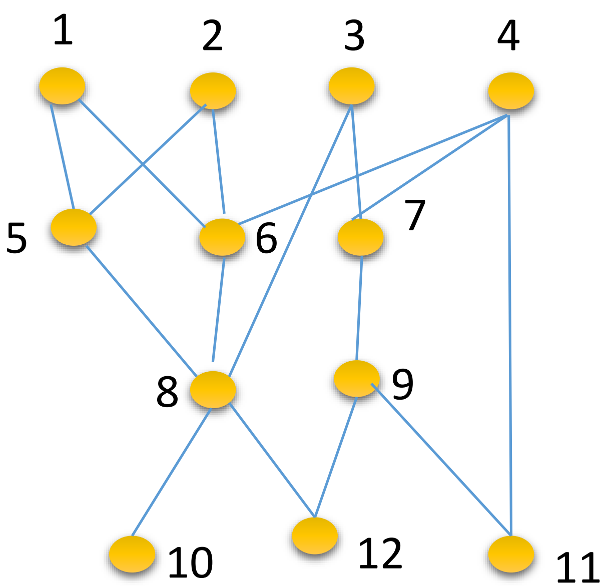




Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

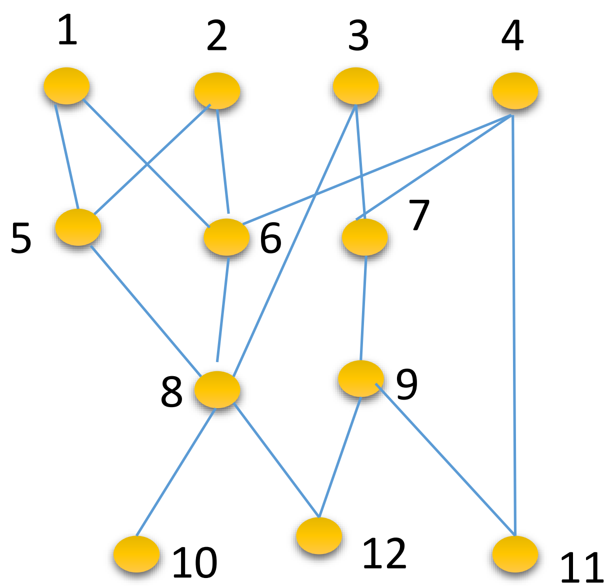


Coffman-Graham Lexicographic Labelling



- Give t minima ~~arbitrary~~ lex#'s 1…t arbitrarily
- Assign lex#s t+1…n so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}, breaking ties arbitrarily

Coffman-Graham Lexicographic Labelling

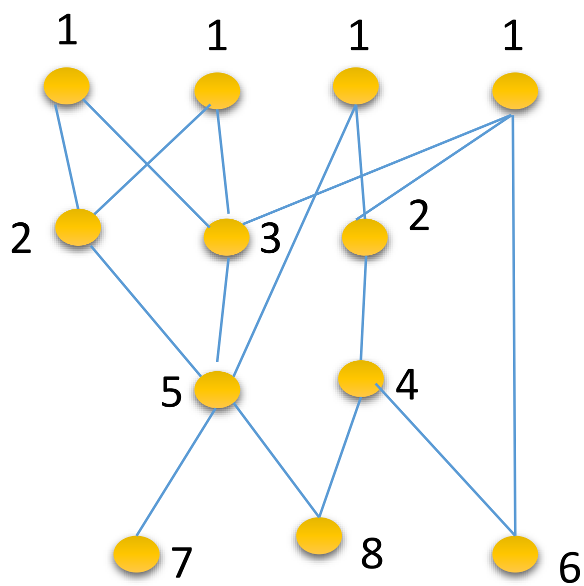


- Give t minima ~~arbitrary~~ lex#'s 1…t arbitrarily

- Assign lex#s t+1…n so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}, breaking ties arbitrarily

(Sethi, 1976) O(n+m) algorithm for C-G lex labelling

Lexicographic Labelling and 2PS



- Coffman and Graham '72 used it for 2-proc scheduling $O(n^2)$
- Sethi '76 also used it for a 2PS; lex labelling takes $O(n + m)$ though the remainder of the 2PS alg takes $O(n\,\alpha(n) + m))$

Further Work

Completed:

- Solve 2-Proc Sched using Greedlex
- Greedlex can work on either transitive closure or transitive reduction
- Greedlex can generate all min-bump linear extensions (all MinPath Covers in Cocomp graphs)

Open:

- Show how to do it all in linear time
- What if the communication delays are weighted?
- What about representations that are in between transitive closure and reduction?
- What about AT-free graphs?
  - Contains the cocomp graphs

Thank You!

Gara Pruesse

Vancouver Island University

Coauthors:

Derek Corneil

Lalla Mouatadid

University of Toronto

Hamiltonicity of Cocomp Graphs

Keil 1985

- Ham'n cycle in Interval graphs alg

Deogun Steiner 1990

- Poly-time Ham'n Cycle

Deogun Kratsch Steiner 1997

- 1-tough cocomp graphs are hamiltonian –

Damaschke Deogun Kratsch Steiner 1991

- Hamilton Path in cocomps using bump number algorithm

Corneil Dalton Habib 2013

- Min Path Cover Alg (certified) in Cocomp Graphs