Lexicographic Labellings achieve fast algorithms for bump number, cocomp hamiltonicity and two-processor scheduling

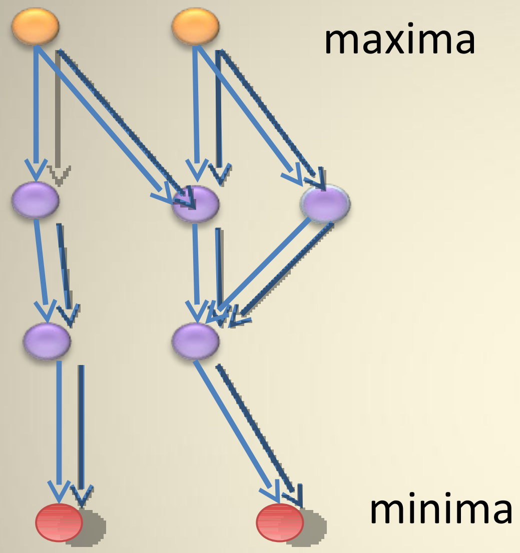
Gara Pruesse
Vancouver Island University

Derek Corneil
Lalla Mouatadid
University of Toronto

Outline

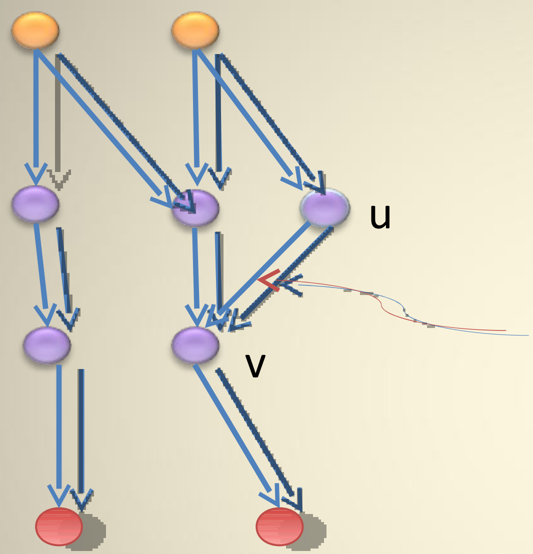
- Introduce Bump Number
- ~~Show relationship with 2-Proc Scheduling~~
- Show relationship with Min Path Cover in Cocomp Graphs
- Introduce Lexicographic Labelling
- Give Greedlex Algorithm
- Prove Greedlex is Correct
- Show how this fits into previous work
- Further work

Posets = partially ordered sets



Hasse Diagram

Posets = partially ordered sets



u *covers* v

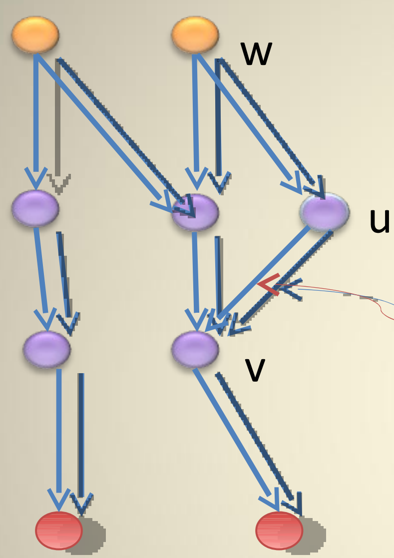
u is an *upper cover* of v

v is a *lower cover* of u

u v

Hasse Diagram

Posets = partially ordered sets

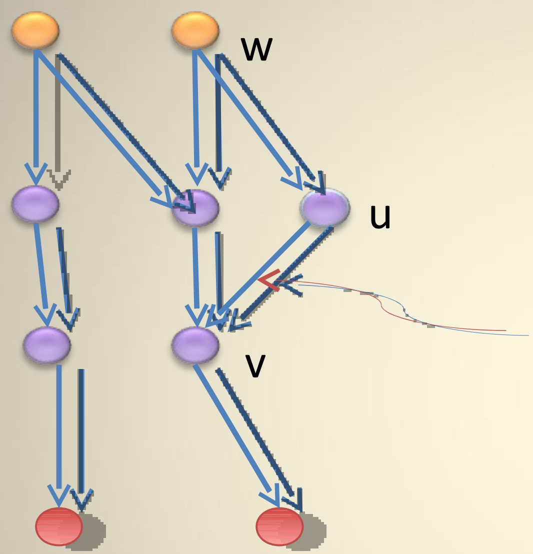


w > v
v < w
v and w are *transitively related*

u *covers* v
u is an *upper cover* of v
v is a *lower cover* of u
u ⋖ v

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

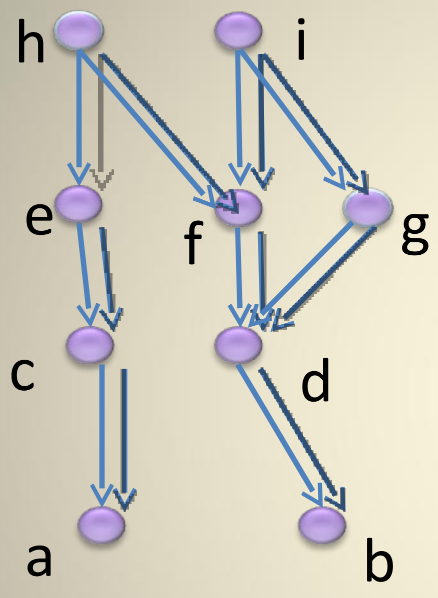
u v

Hasse Diagram

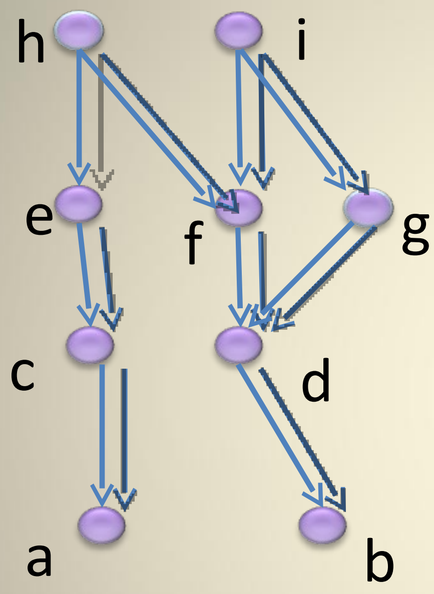
-A compact representation of a set of relations

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

Bumps in linear extensions

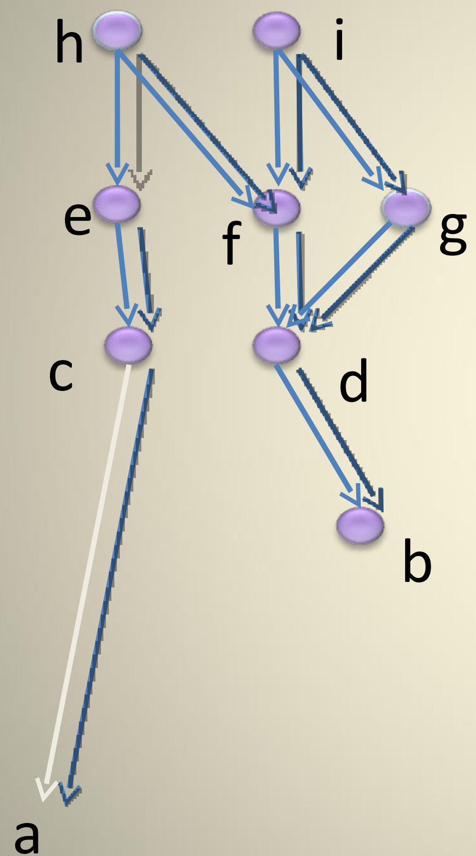


Bumps in linear extensions



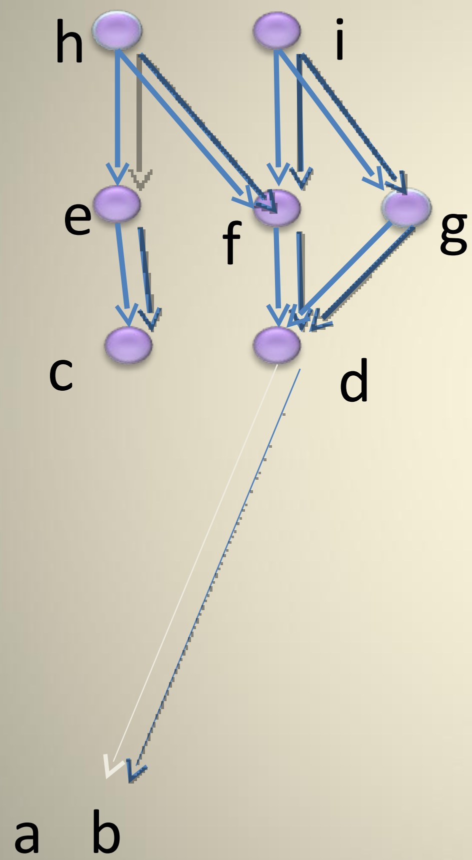
Linear extension (showing bumps)

Bumps in linear extensions



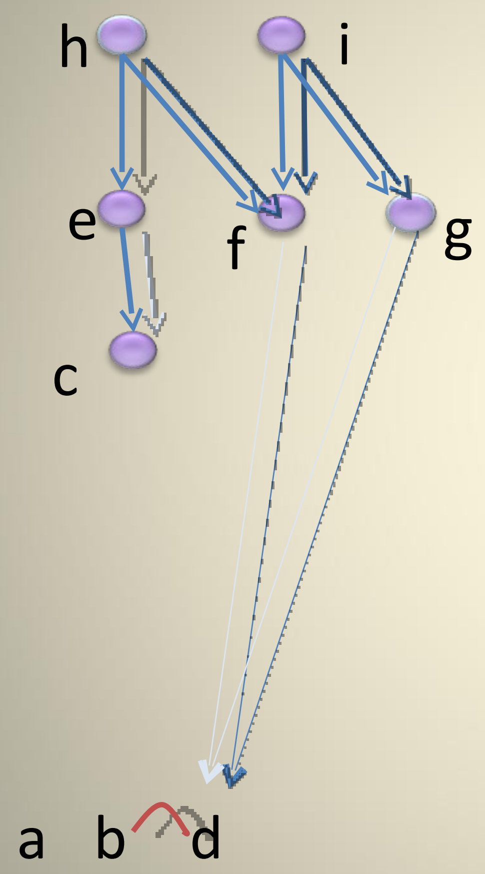
Linear extension (showing bumps)

Bumps in linear extensions



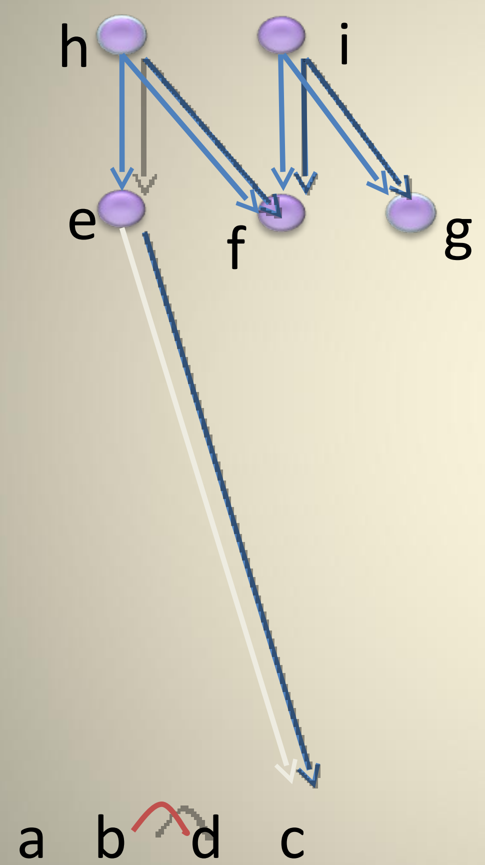
Linear extension (showing bumps)

Bumps in linear extensions



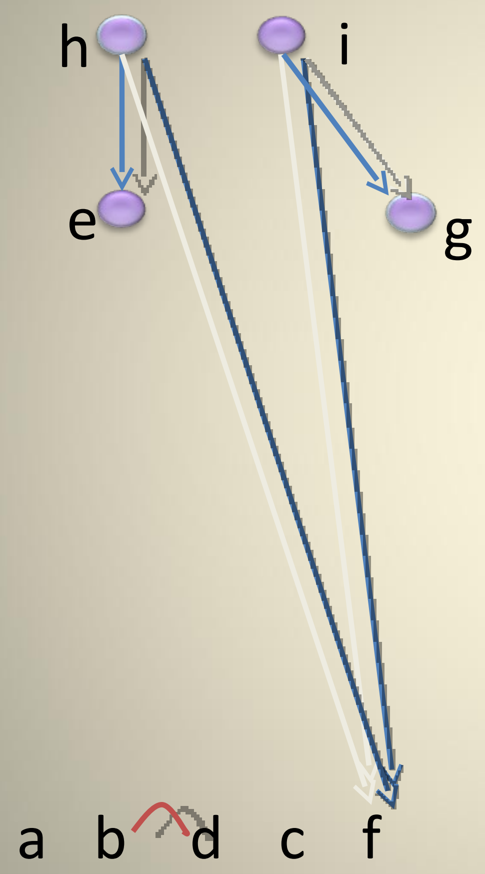
Linear extension (showing bumps)

Bumps in linear extensions



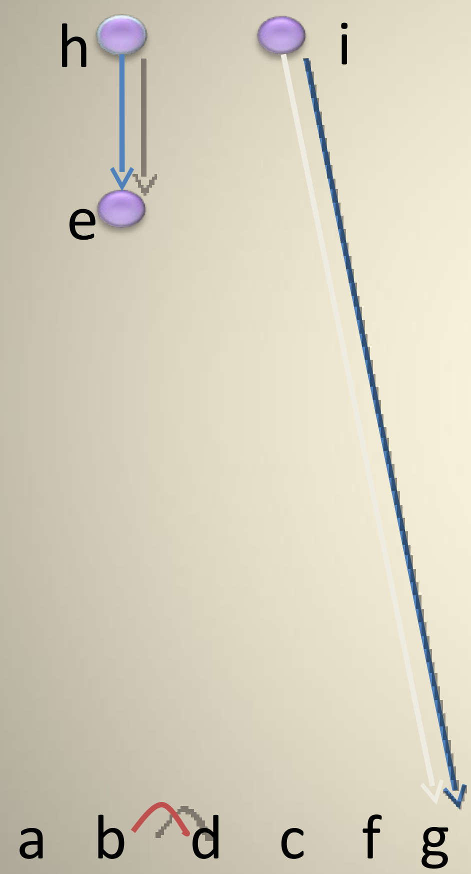
Linear extension (showing bumps)

Bumps in linear extensions



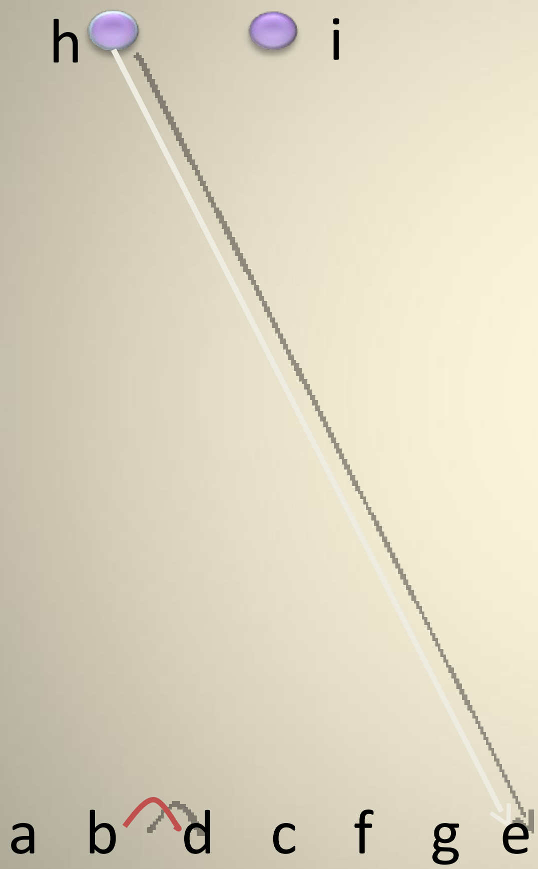
Linear extension (showing bumps)

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

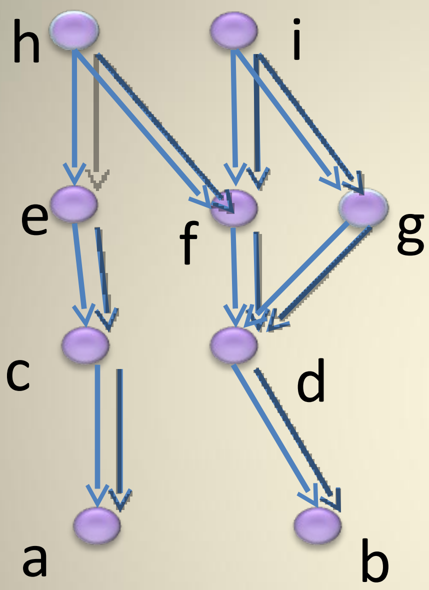
Given poset P, what is the least number of bumps realized by a linear extension of P?

b(P)= bump# of P

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

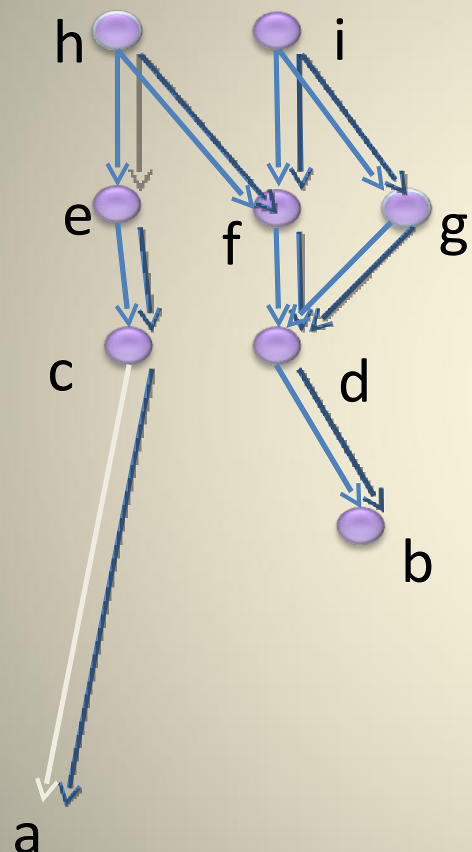
a b d c f g e i h

Greedily seeking min-bump l.e.



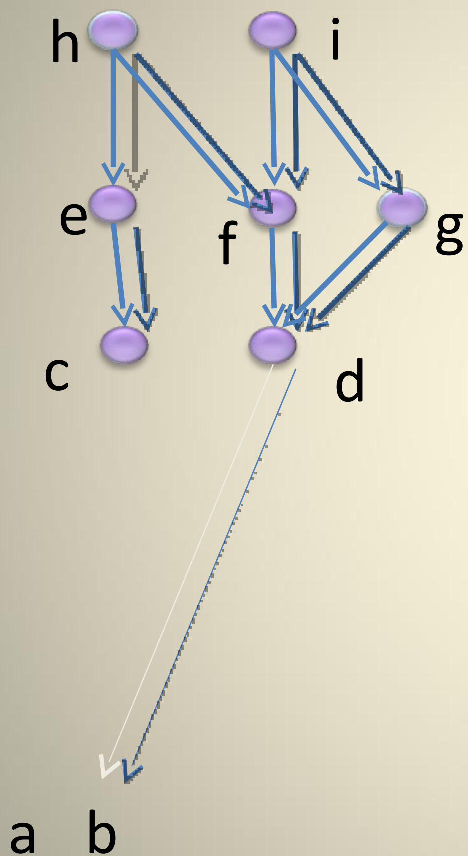
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



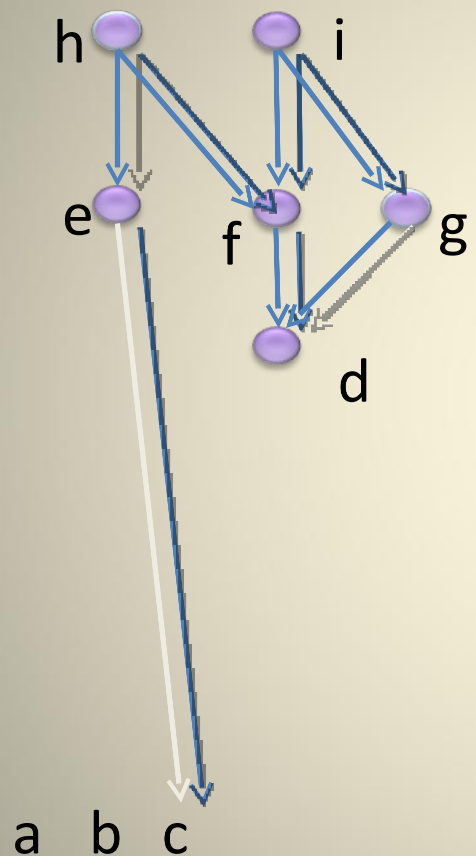
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



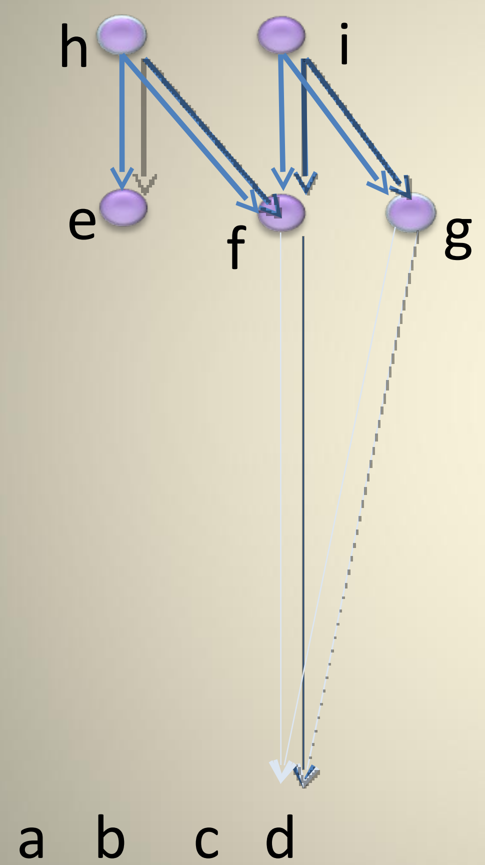
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



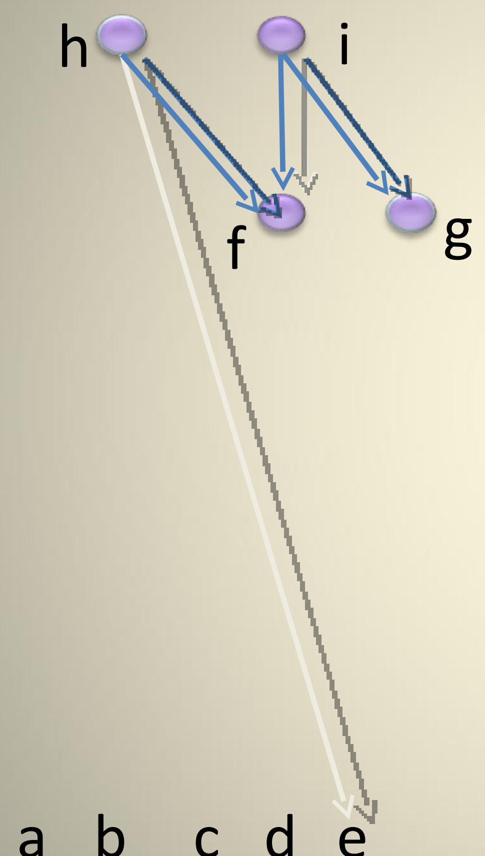
Linear extension (showing bumps)
Greedily selecting to avoid bumps

Greedily seeking min-bump l.e



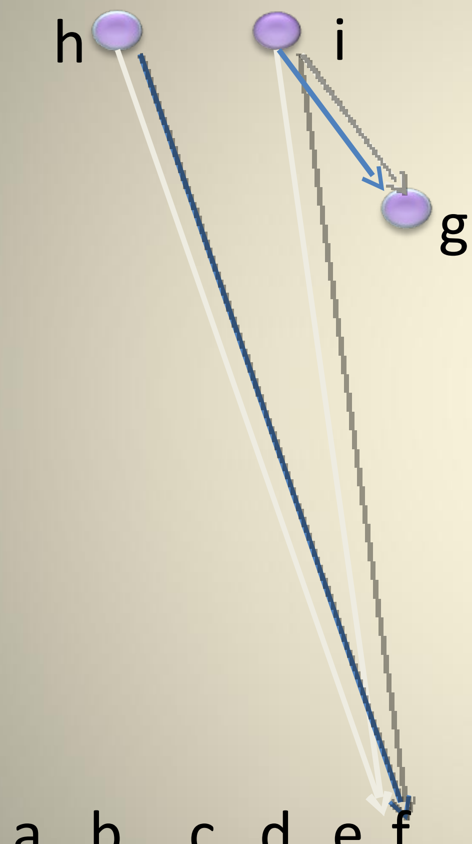
Linear extension (showing bumps)
Greedily selecting to avoid bumps

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

Linear extension (showing bumps)
Greedily selecting to avoid bumps

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

Greedlex Algorithm does these quickly, simply

- 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

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

O(n+m)

… 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:

$\exists$ a simple algorithm

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

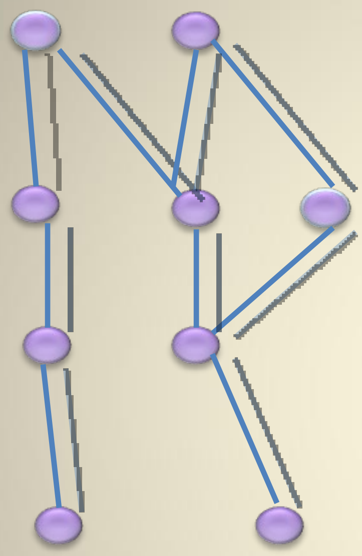
Question:

$\exists$ a simple algorithm YES

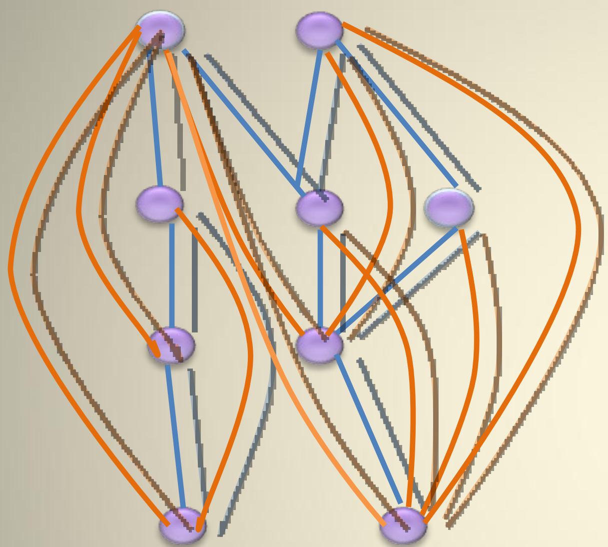
with a short proof YES

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

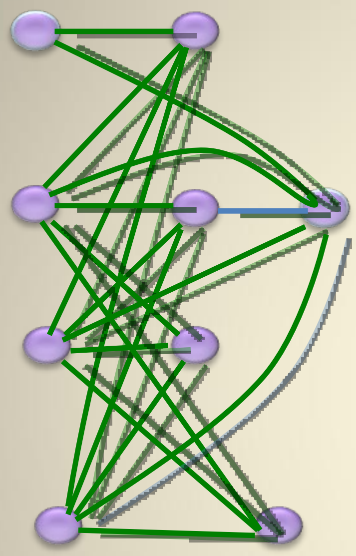
Posets & comparability graphs



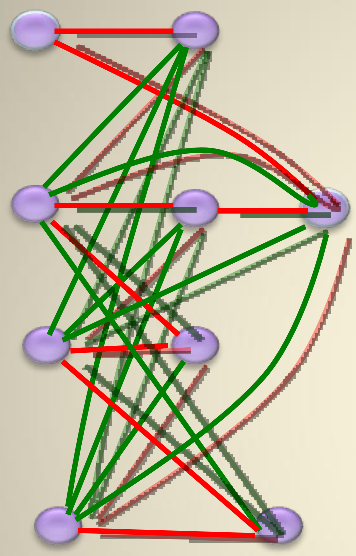
Posets & comparability graphs



Posets & cocomparability graphs

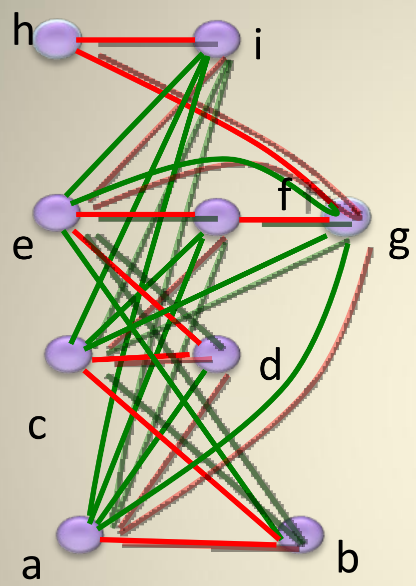


Posets & cocomparability graphs



When is there a Hamilton Path in the cocomparability graph?

Posets & cocomparability graphs



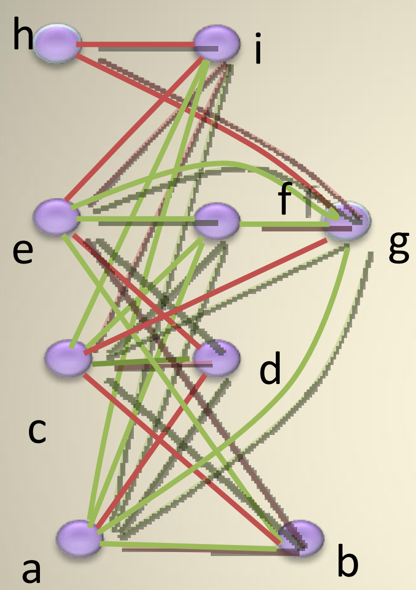
When is there a Hamilton Path in the cocomparability graph?

When there is an ordering of the vertices so that there is an edge between successive vertices

…i.e., so that there is a non-edge in the *comparability* graph

… i.e., so there is *no bump* between successive vertices in the linear extension (assuming your restrict to orderings that obey the partial order).

Posets & cocomparability graphs



When is there a Hamilton Path in the cocomparability graph?

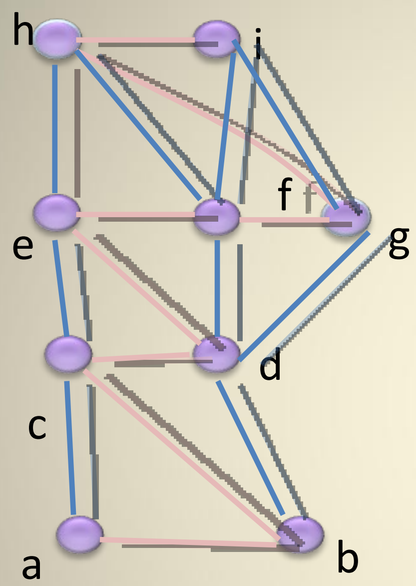
Of course, it is possible to trace the graph in ways that are not obedient to the partial order

a d e i h g c b

Exists Ham Path iff exists cocomp order that is a HamPath iff bump#=0

Exists k-path cover in cocomp graph iff bump# ≤ k

Posets & cocomparability graphs



When is there a k-path cover in the cocomparability graph?

Cocomp graph G ⟺ many posets

Cocomp graph G + cocomp ordering ⟺ one poset

Solve bump on the poset ⟺ Solve min-path-cover on cocomp graph

Solve MPC on cocomp graph using a cocomp order ⟺ Solve bump on the unique underlying poset

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

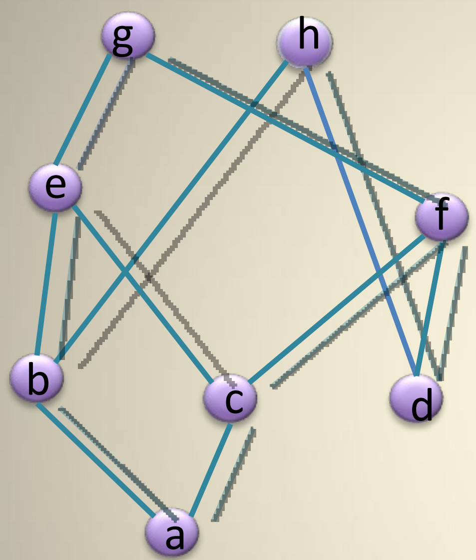
- Min Path Cover Alg (certified) in Cocomp Graphs

Recap:

- Definition of Bump Number
- Relationship (equivalency, up to data representation) to the Minimum Path Cover/Hamiltonicity of Cocomp Graphs
- Is related to Two-Processor Scheduling

- Introduce Lexicographic Labelling
- Give the Greedlex Algorithm solving Bump
- Prove Greedlex is correct
  - State the Lex-Yanking Lemma
  - Show that the Lex-Yanking Lemma implies Greedlex is Correct
  - Prove the Lex-Yanking Lemma
- How this work fits into previous results

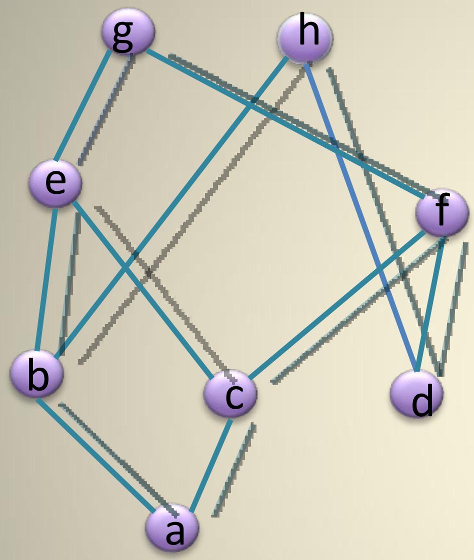
Greedy bump#



Greedy Approach

d a … oops

Greedy bump#

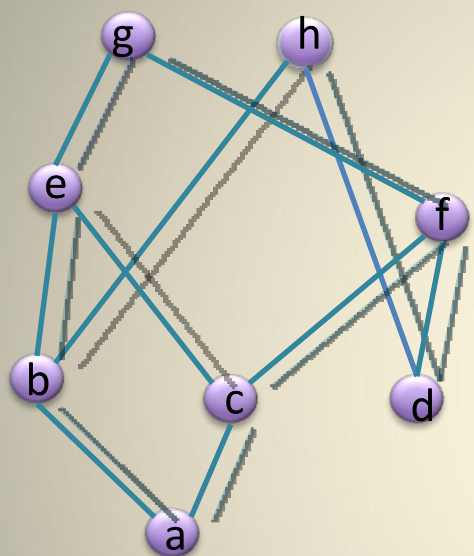


Greedy Approach

d a ... oops

a d b c h e f ...oops

Greedy bump#



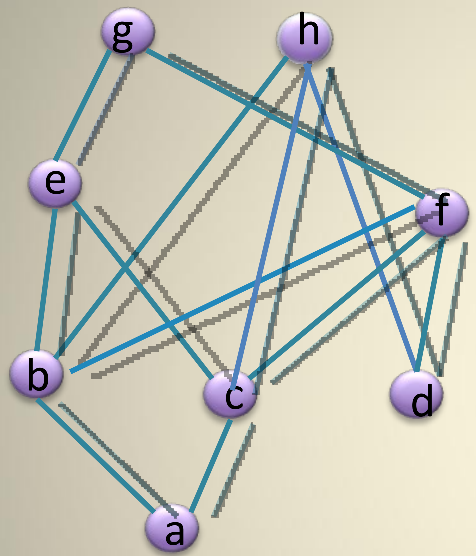
Greedy Approach

d a ... oops

a d b c h e f ...oops

a d c b f e h g

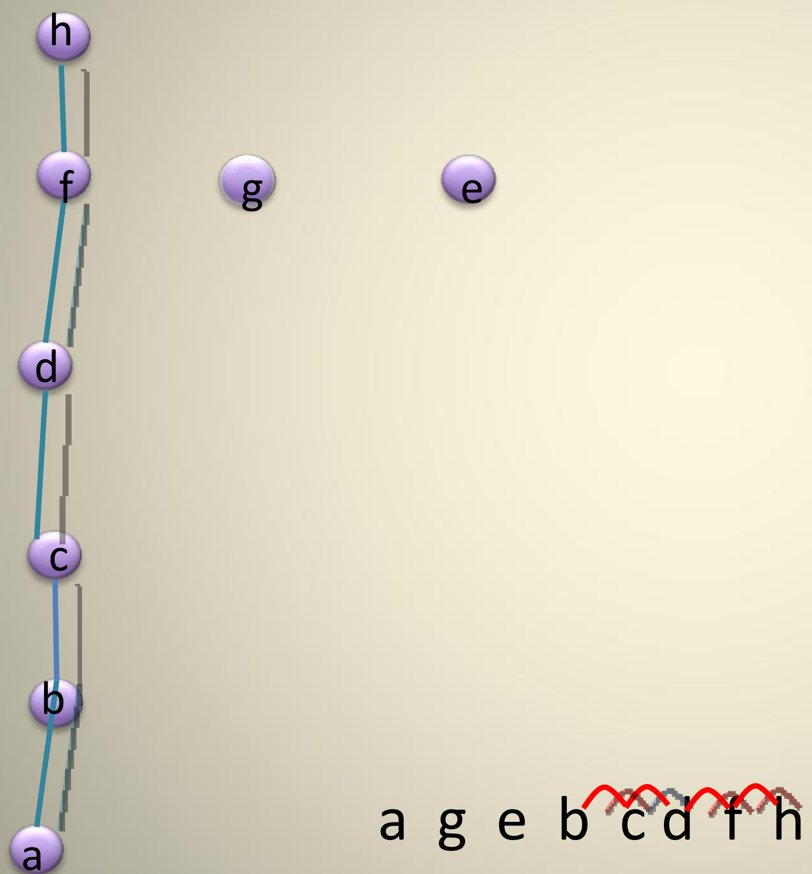
How can a bump be unavoidable



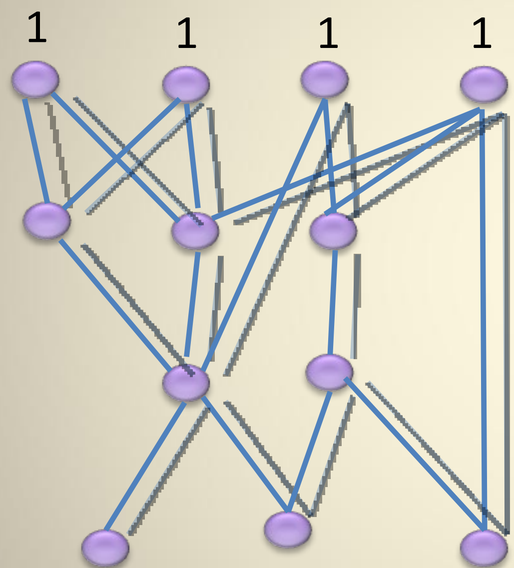
a d b c …

Now all minima e f g are upper covers of c

How can a bump be ~~un~~avoidable

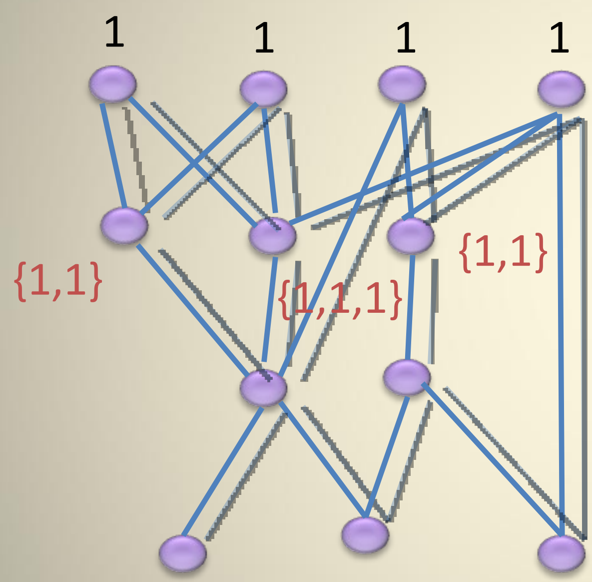


Lexicographic Labelling



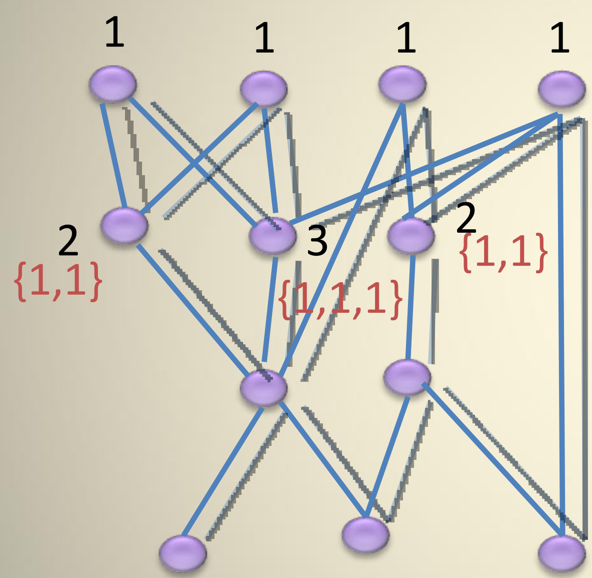
- Give minima arbitrary lex#

Lexicographic Labelling



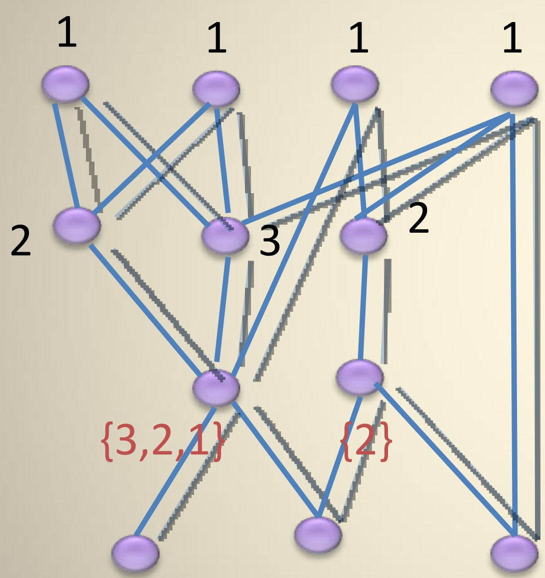
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



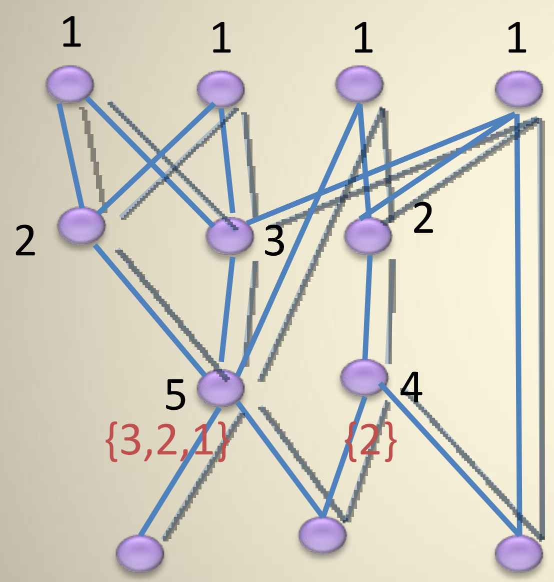
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



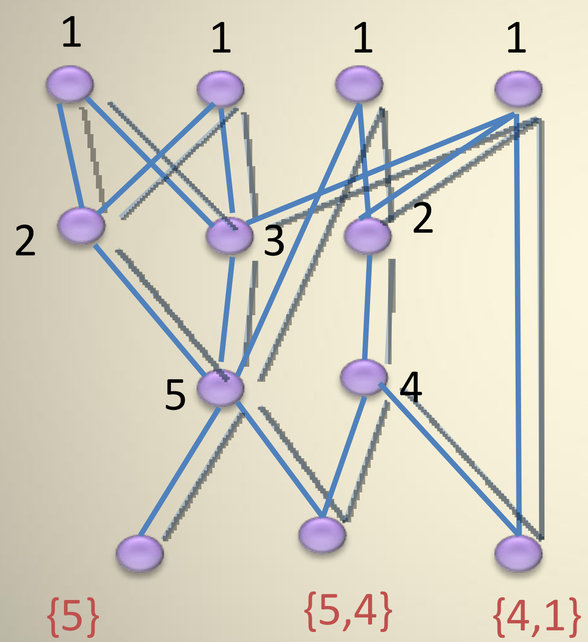
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



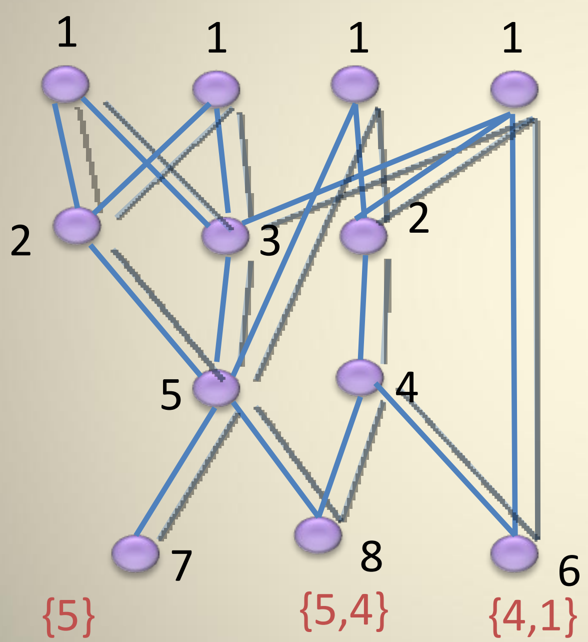
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



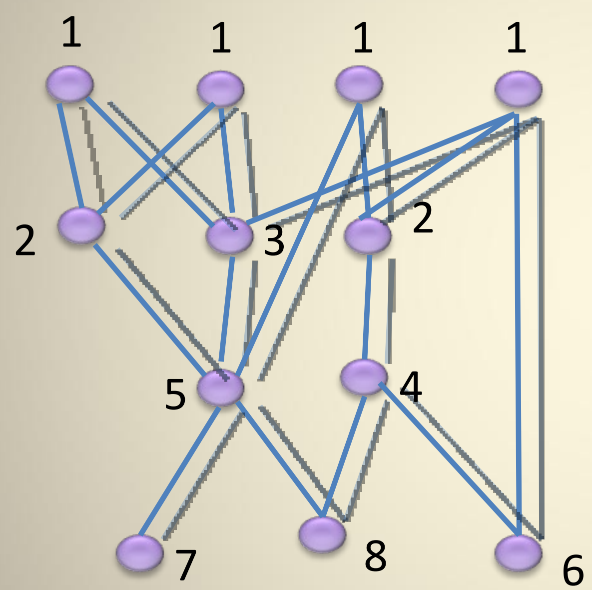
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{lexico}$ {lex(v'): v' covers v}

Lexicographic Labelling



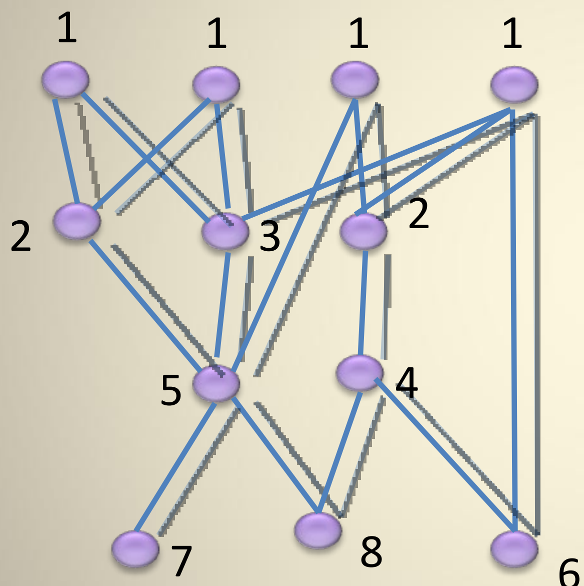
- Give minima arbitrary lex#
- Assign lex# so that lex(u)<lex(v) whenever {lex(u'): u'covers u} $<_{\text{lexico}}$ {lex(v'): v' covers v}

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}

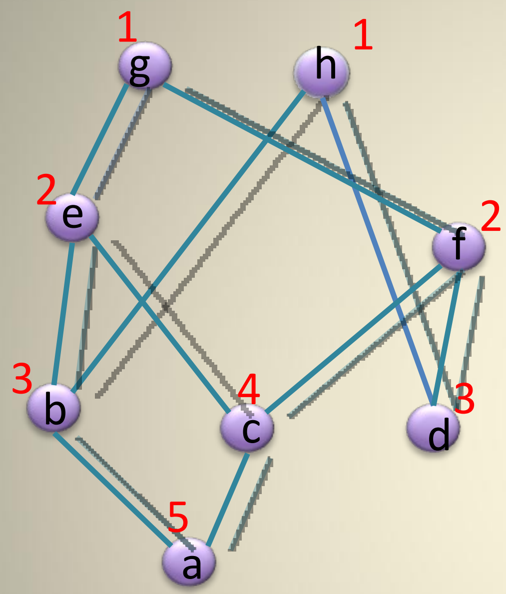
Lexicographic Labelling



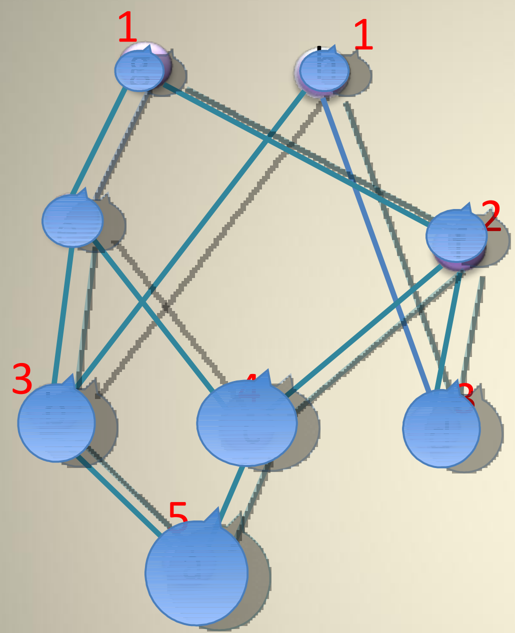
New: O(n+m) algorithm for lex-labelling

(Sethi 1976 algorithm also acheives linear time)

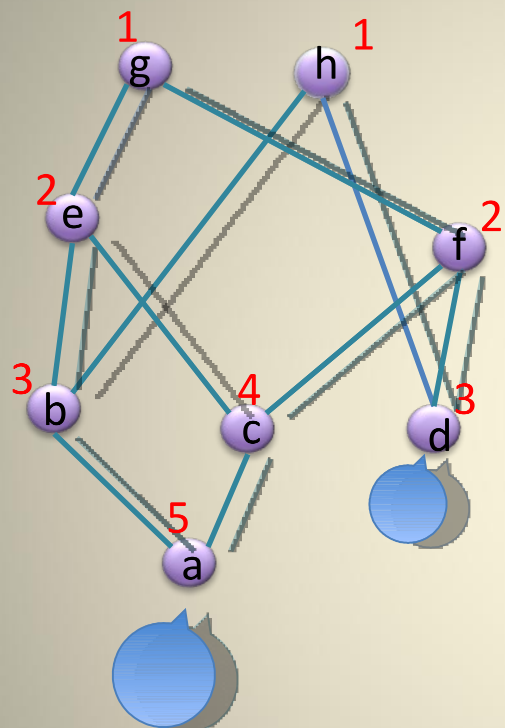
Greedy + Lex = Greedlex



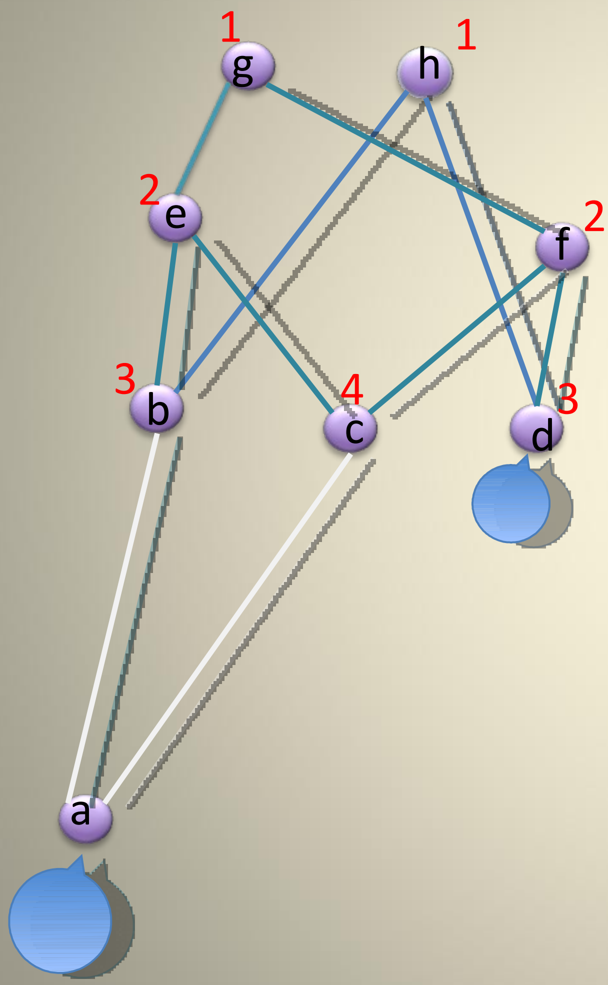
Greedy + Lex = Greedlex



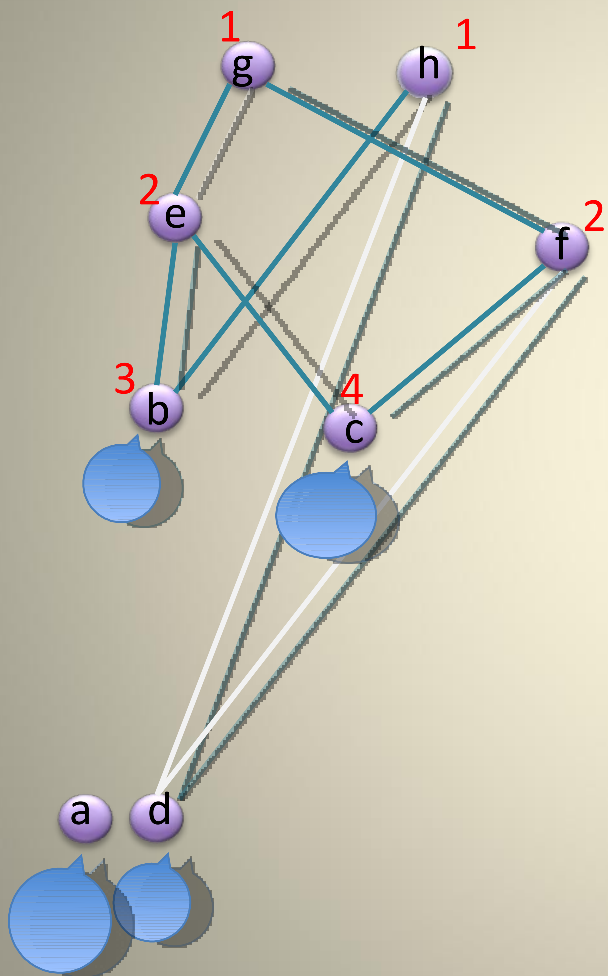
Greedy + Lex = Greedlex



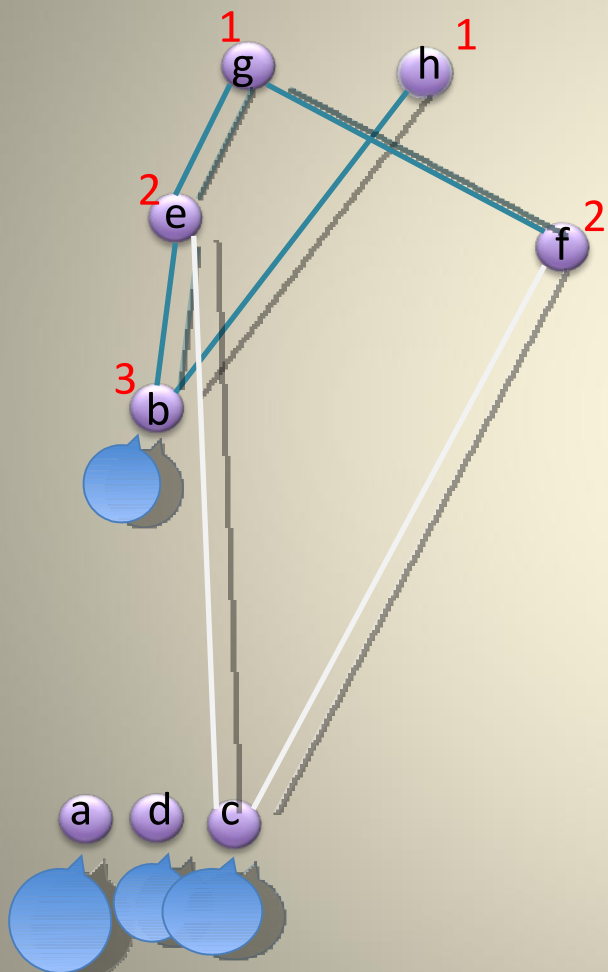
Greedy + Lex = Greedlex



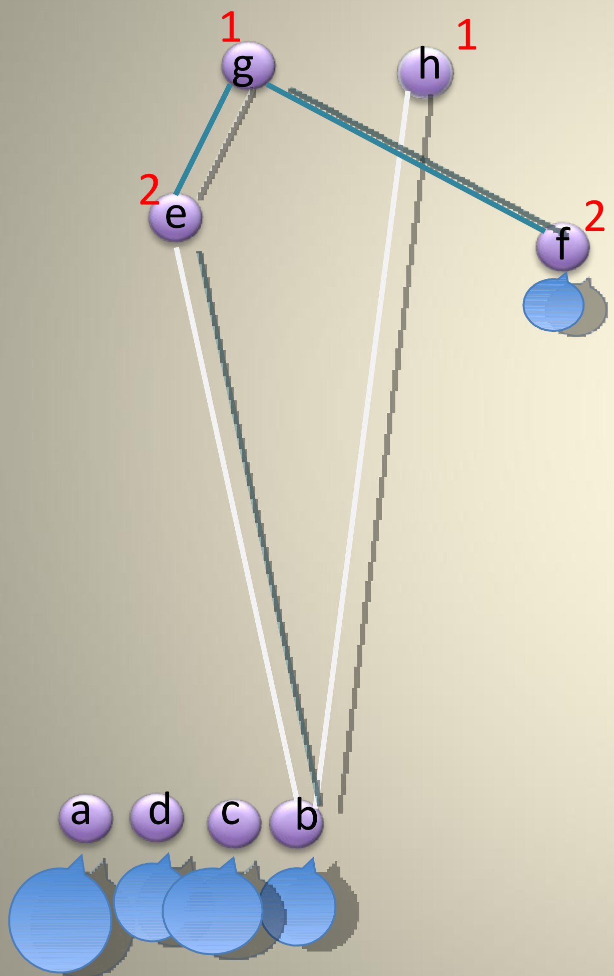
Greedy + Lex = Greedlex



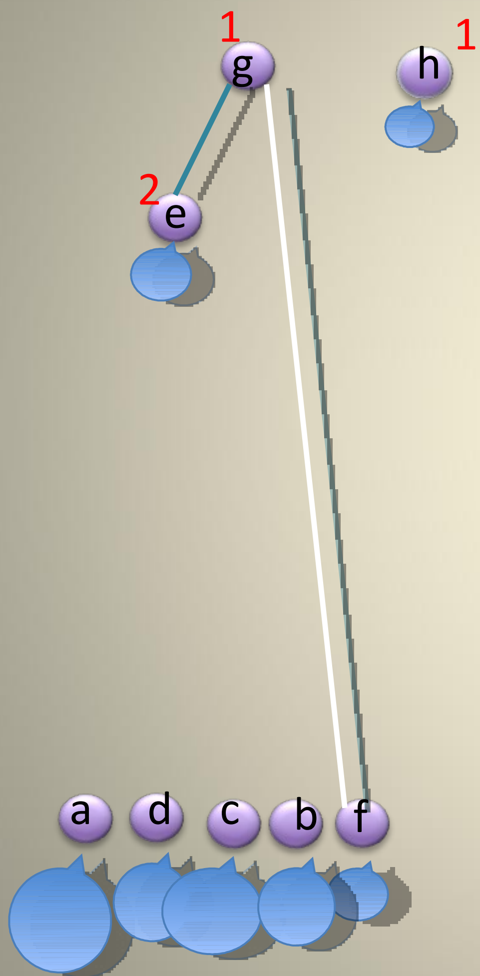
Greedy + Lex = Greedlex



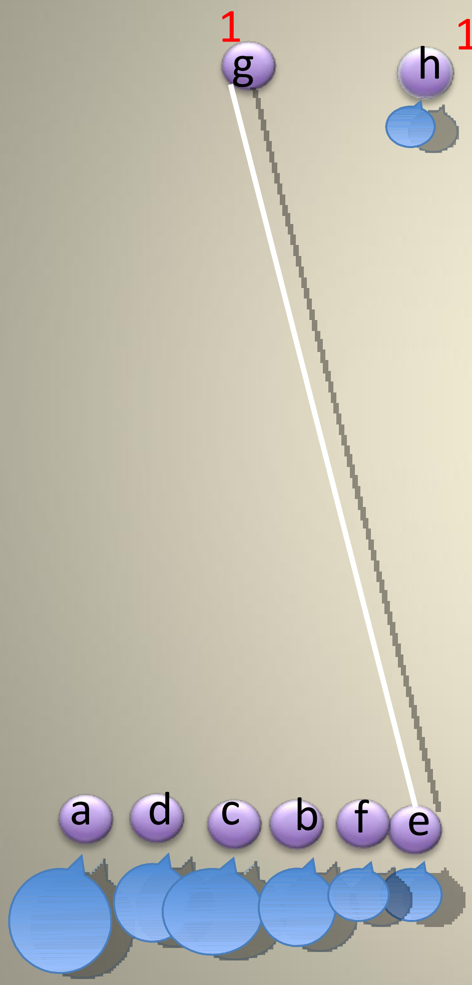
Greedy + Lex = Greedlex



Greedy + Lex = Greedlex

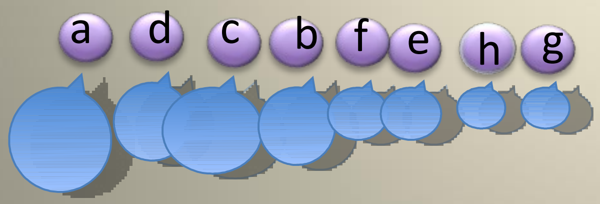


Greedy + Lex = Greedlex



Greedy + Lex = Greedlex

Greedy + Lex = Greedlex



Greedlex Alg for bump#

1. Lex label all v in V(P)
2. Shell P, always removing

    (a) a non-cover of last-shelled u, if exists

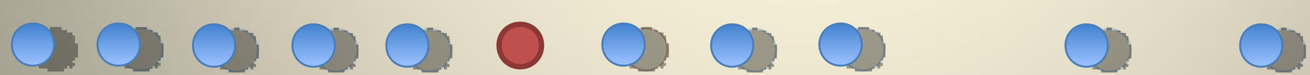
    (b) the highest lex-labelled v allowed by (a)

This always yields the min-bump l.e.!

Proof of Correctness

First, an observation:

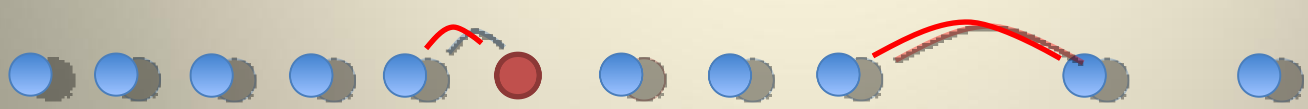
When shelling to produce a low-bump l.e., if you make one bad selection, how many *added bumps* can that introduce?



Proof of Correctness

First, an observation:

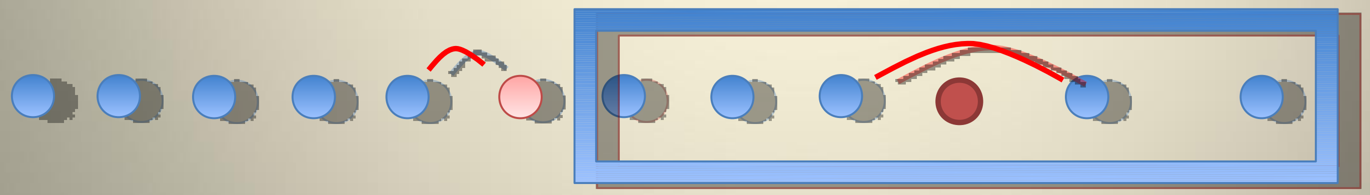
When shelling to produce a low-bump l.e., if you make one bad selection, how many *added bumps* can that introduce?



Proof of Correctness

First, an observation:

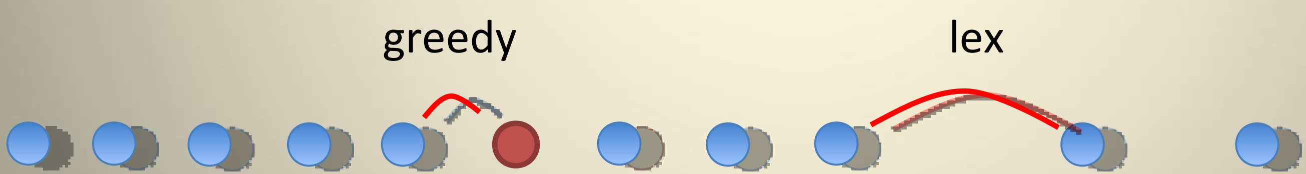
When shelling to produce a low-bump l.e., if you make one bad selection, how many *added bumps* can that introduce?



Proof of Correctness

First, an observation:

When shelling to produce a low-bump l.e., if you make one bad selection, how many *added bumps* can that introduce?



Lex-Yanking Lemma

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

$\Rightarrow$ ∃ l.e. a x'x'x'…b…x' with k or fewer bumps

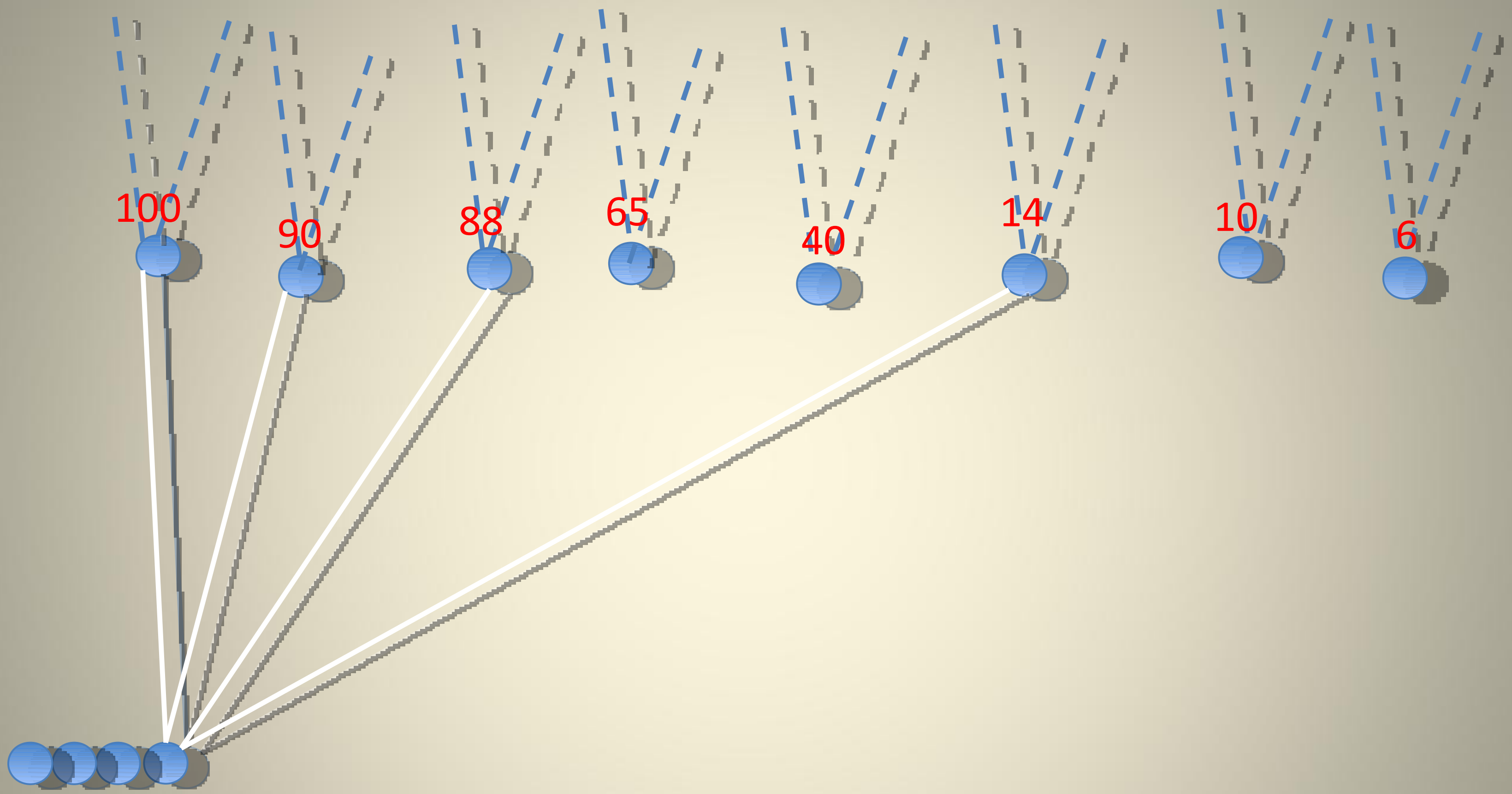
(Balloon size indicates relative Lex value)

Recap:

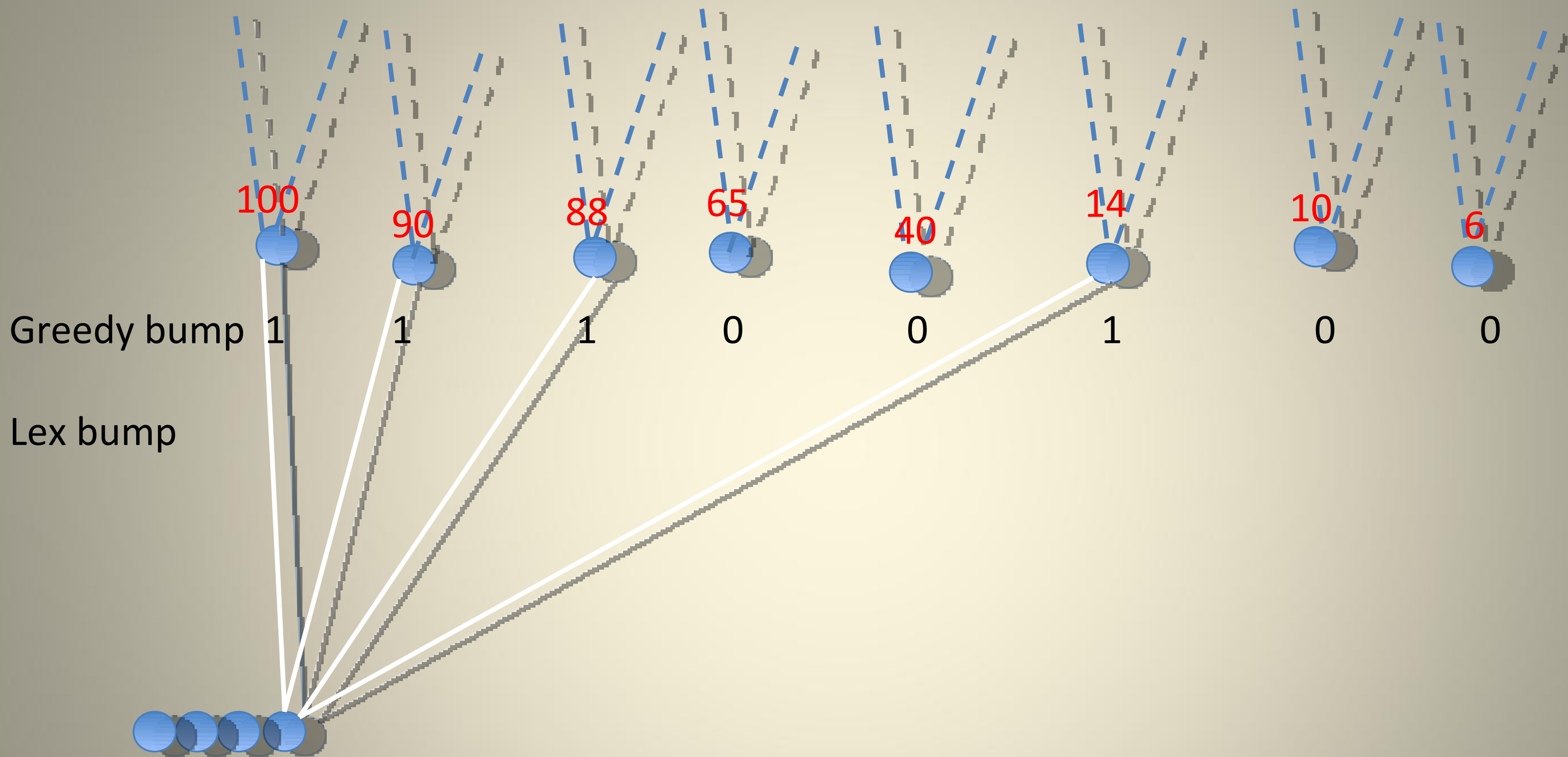
- Definition of Bump Number
- Relationship (equivalency, up to data representation) to the Minimum Path Cover/Hamiltonicity of Cocomp Graphs
- Is related to Two-Processor Scheduling

- Introduce Lexicographic Labelling
- Give the Greedlex Algorithm solving Bump
- Prove Greedlex is correct
  - State the Lex-Yanking Lemma
  - Show that the Lex-Yanking Lemma implies Greedlex is Correct
  - Prove the Lex-Yanking Lemma
- How this work fits into previous results

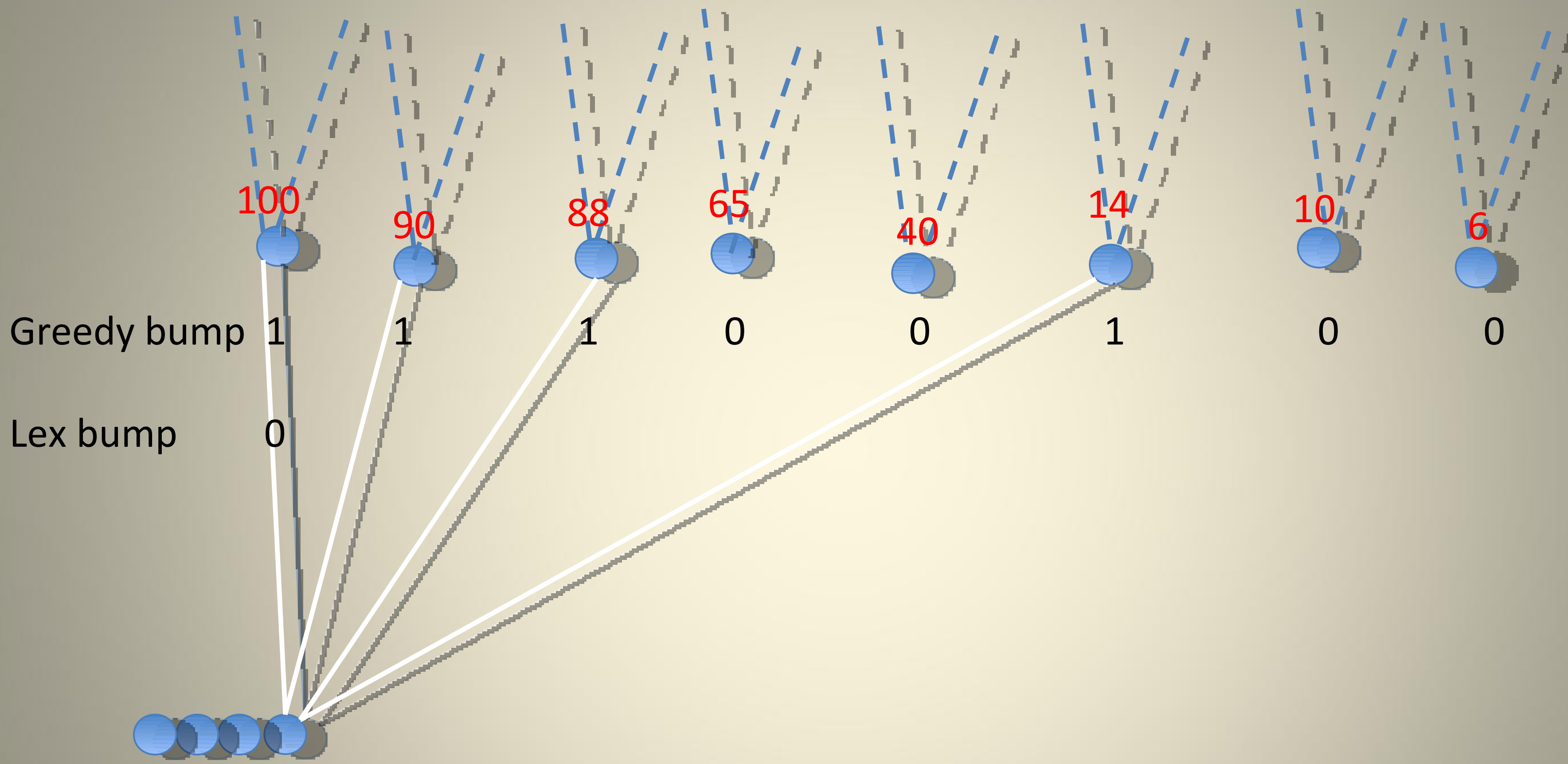
The LexYanking Lemma implies Greedlex works:



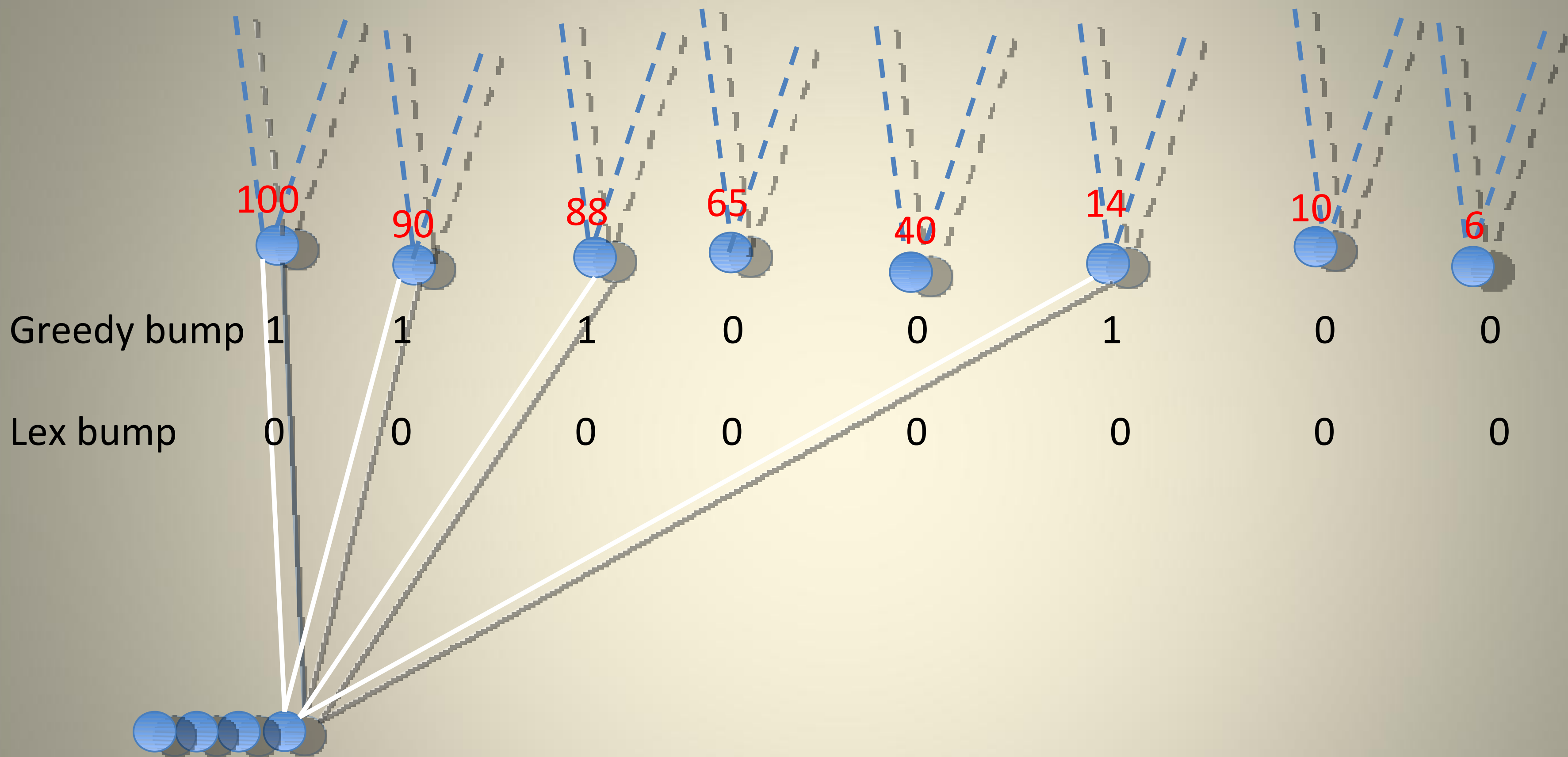
The LexYanking Lemma implies Greedlex works:



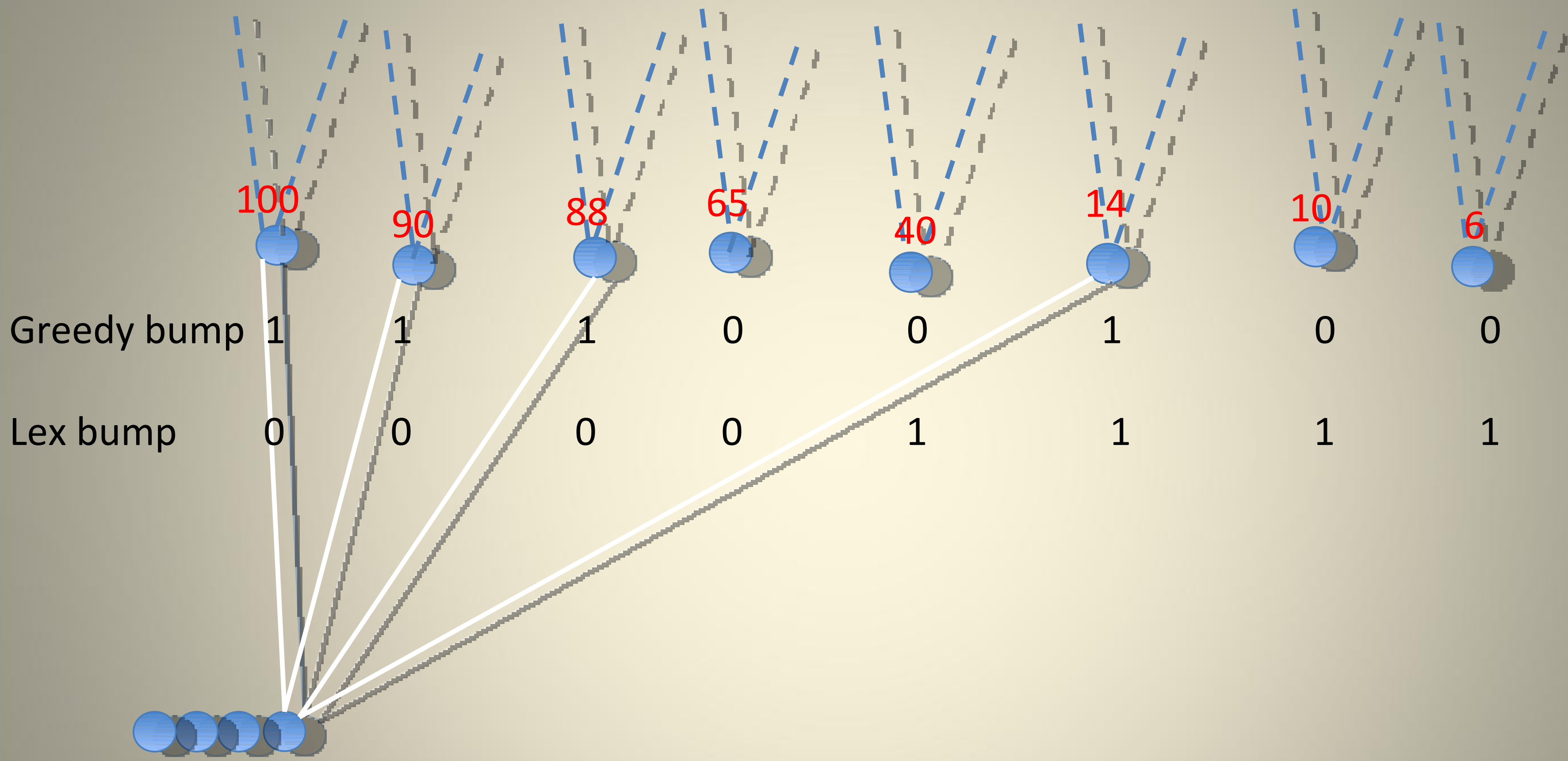
First, let us note that the LexYanking Lemma implies Greedlex works:



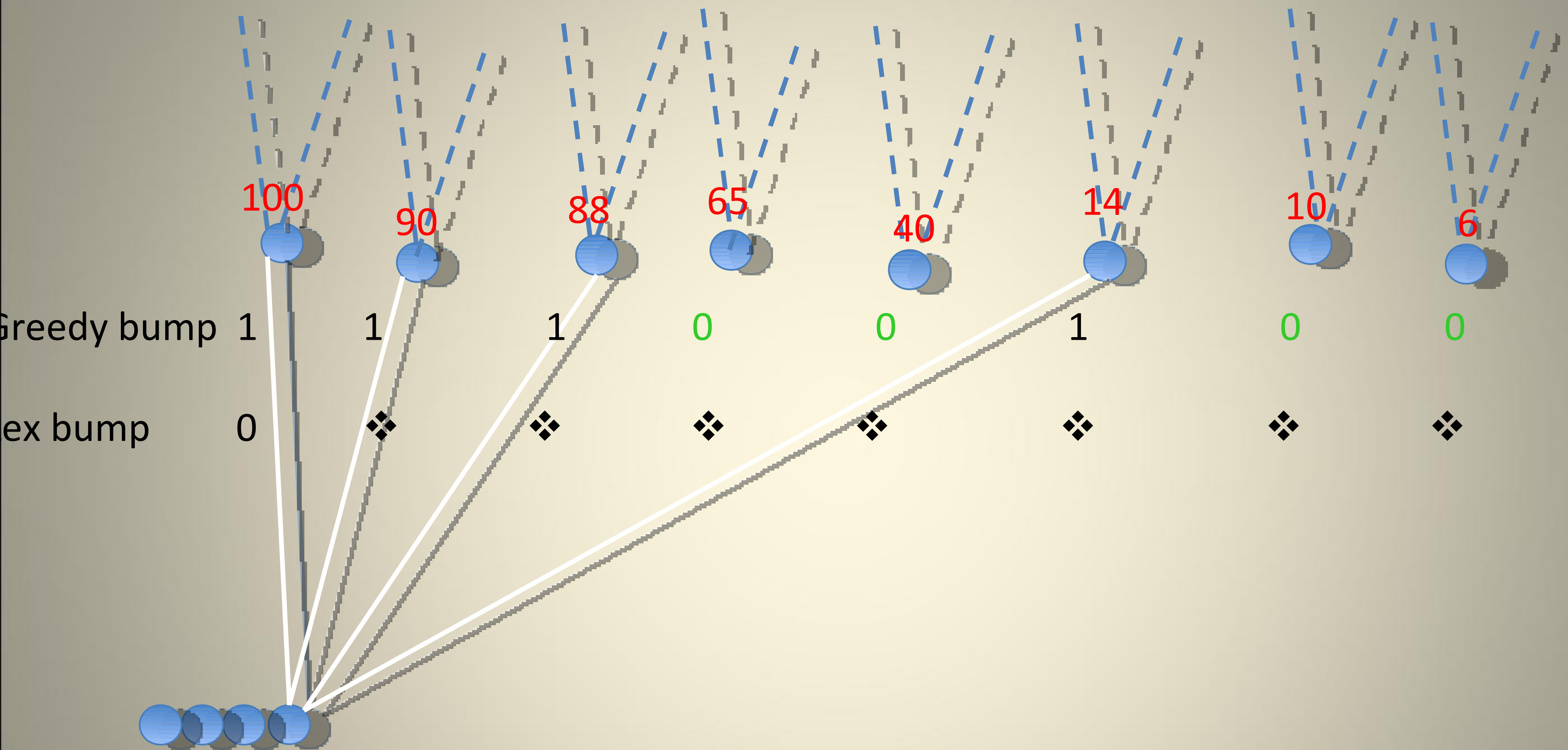
First, let us note that the LexYanking Lemma implies Greedlex works:



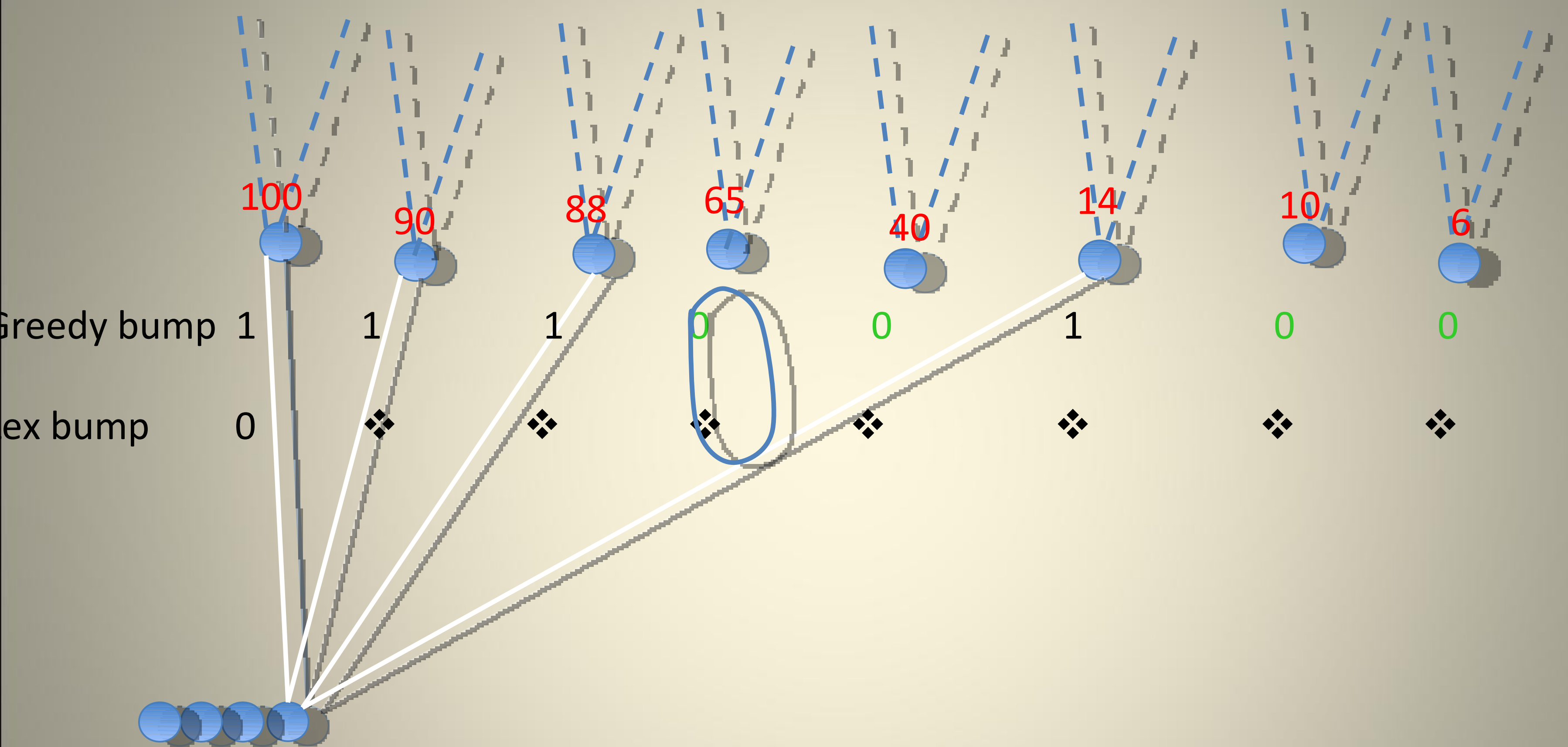
First, let us note that the LexYanking Lemma implies Greedlex works:



First, let us note that the LexYanking Lemma implies Greedlex works:



First, let us note that the LexYanking Lemma implies Greedlex works:

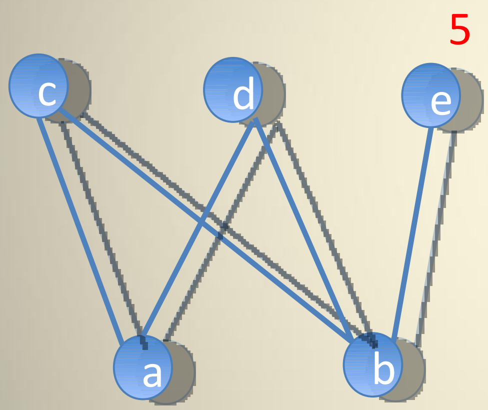


LexYanking Lemma ➔ the Greedlex Algorithm will always yield min-bump l.e.

Proof of LexYanking Lemma

b x x x a x x x   has k bumps, lex(a) ≥ lex(b), a is minimal

⇒ a x’x’x’ b x’x’x’   has ≤ k bumps

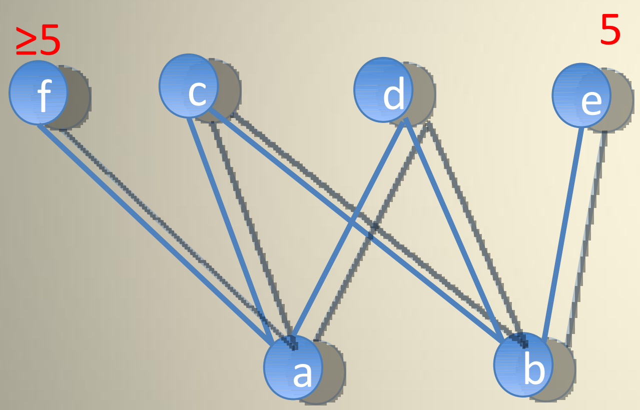


If lex(a)≥lex(b) and b has a private neighbour…

Proof of LexYanking Lemma

b x x x a x x x has k bumps, lex(a) ≥ lex(b), a is minimal

⇒ a x'x'x' b x'x'x' has ≤ k bumps



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

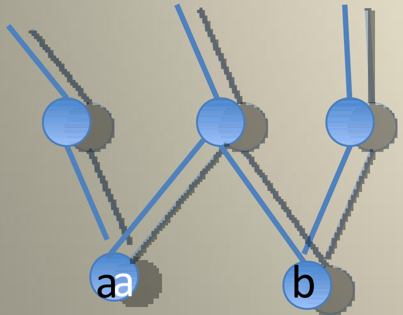
Then a has a private cover with lex# at least as large.

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



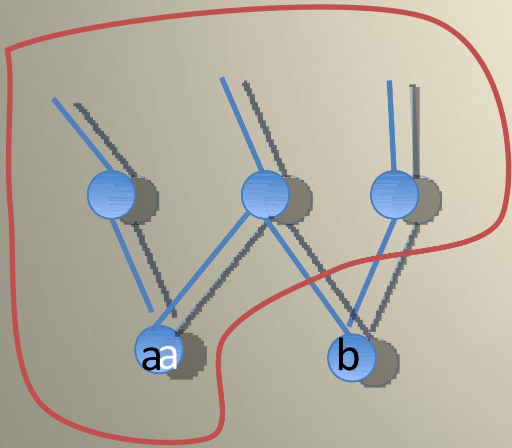
Proof of LexYanking Lemma

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)



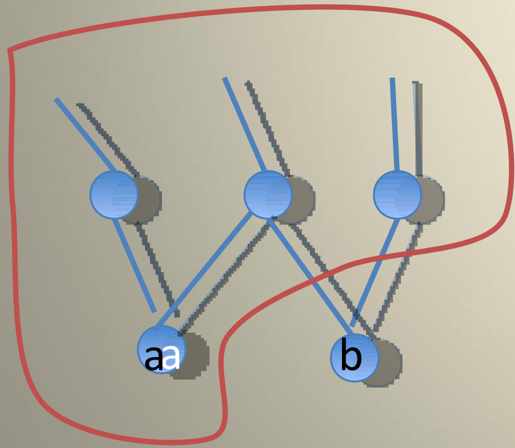
Proof of LexYanking Lemma

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)

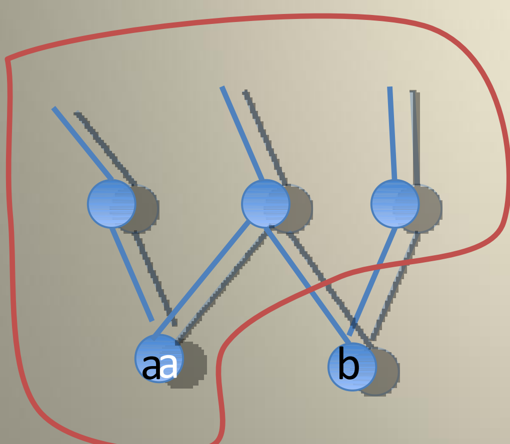


Proof of LexYanking Lemma

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

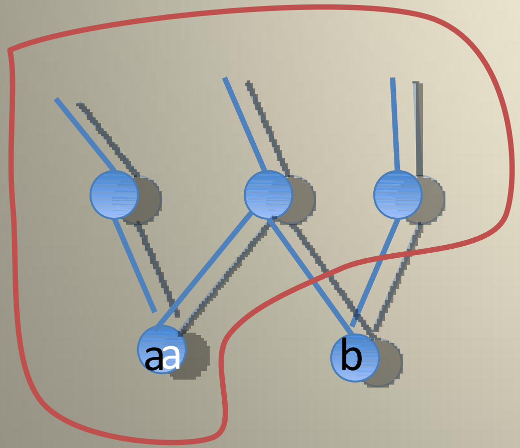
Swap: a yyy b yyyy

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 y1yyy

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

Then y1 is a private cover of b (with respect to a).

Proof of LexYanking Lemma

a yyy b y1yyy

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

Then y1 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 y1y…c..y

Proof of LexYanking Lemma

a yyy b y1yyy

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

Then y1 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 y1y…c..y

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

a yyy b c z…y1..z

Proof of LexYanking Lemma

a yyy b y1yyy

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

Then y1 is a private neighbour of b (with respect to a).

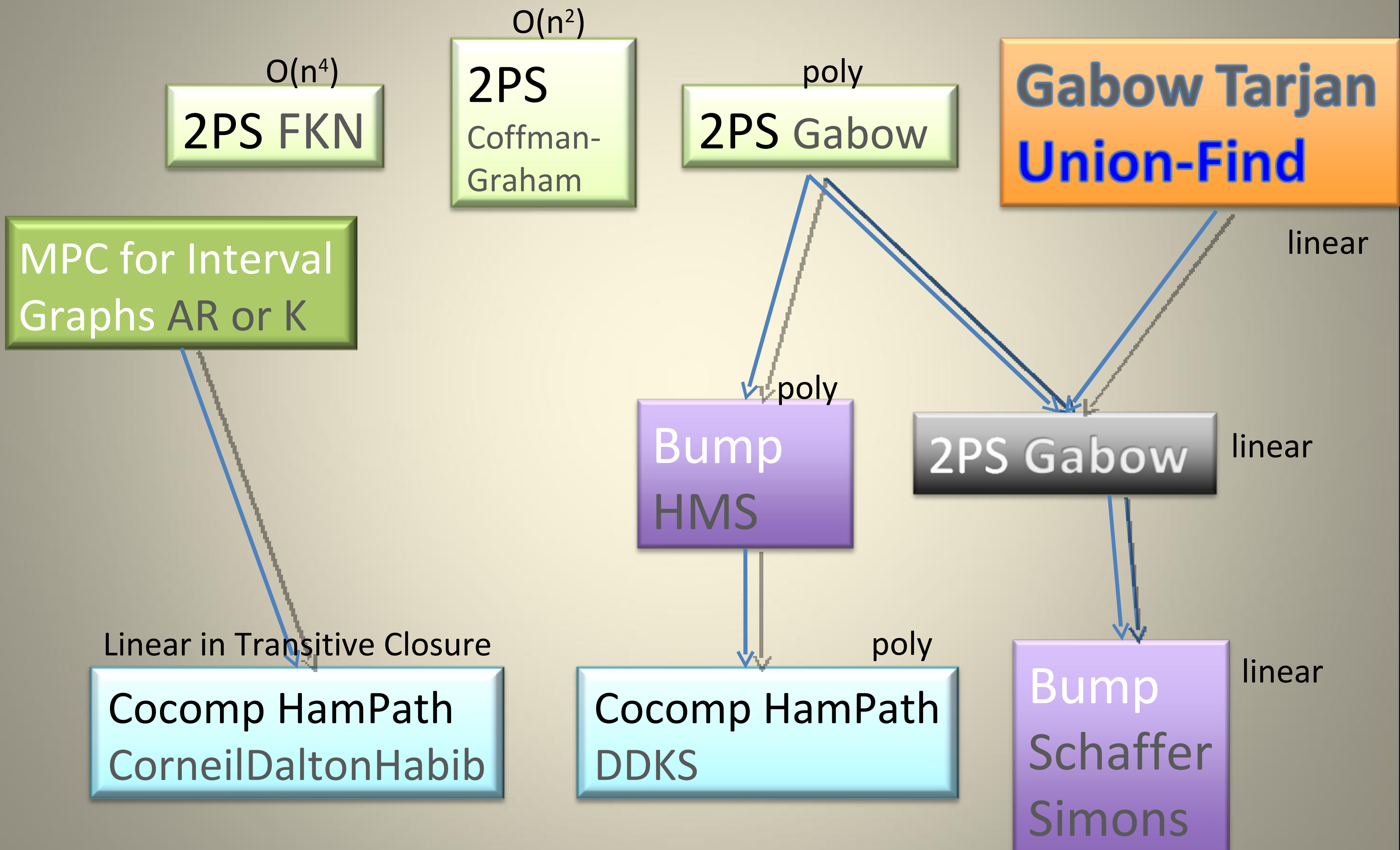
Then a has some private neighbour c (w.r.t. b), with lex(c)≥ lex(y_1).

a yyy b y1y…c..y

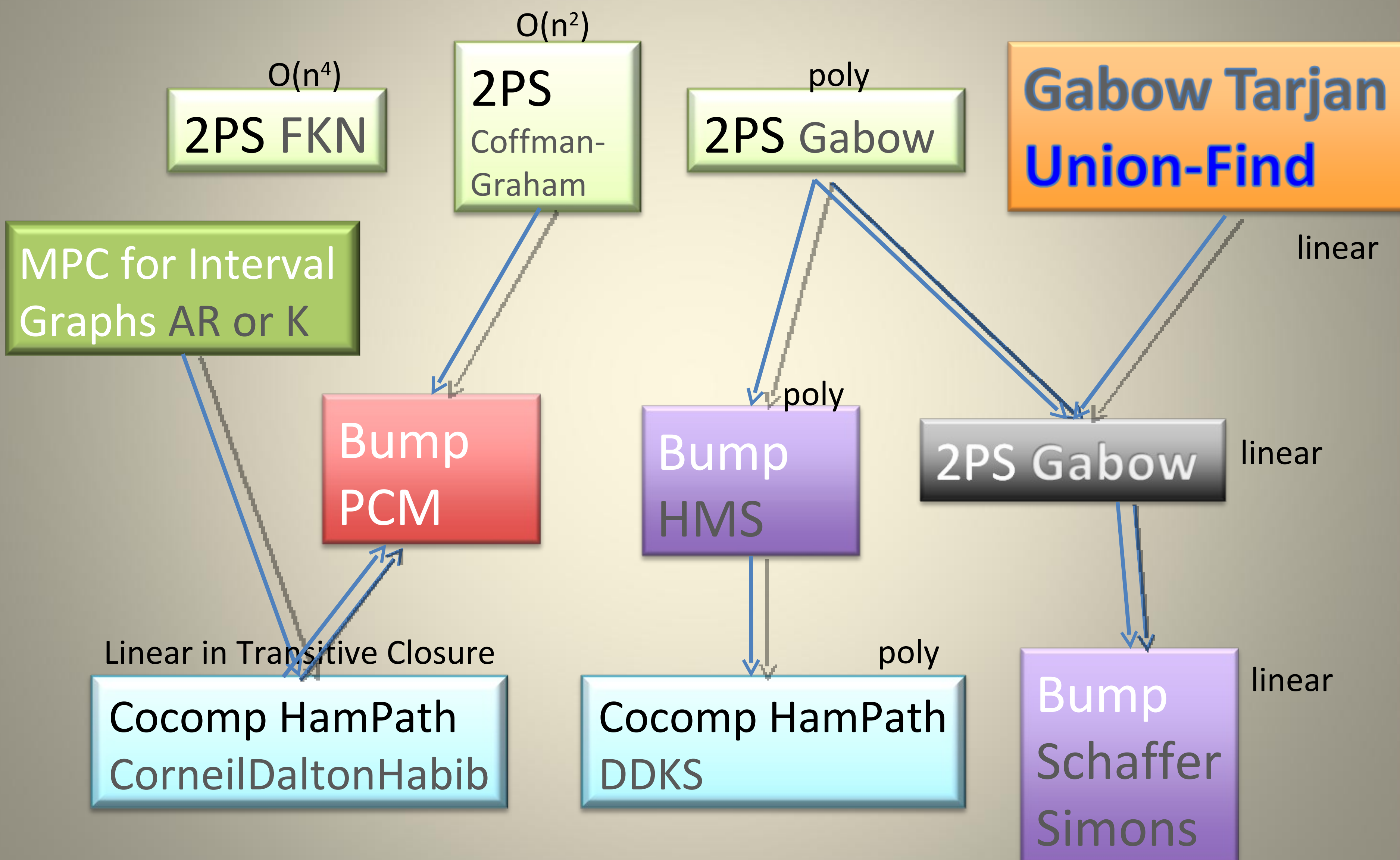
a yyy b c z…y1..z

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?



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:

- Terminal elements in the poset.... (see Garth Isaak's work on Path Partitions)
- What about representations that are in between transitive closure and reduction?
- What about AT-free graphs?
  - Contains the cocomp graphs

Thank You!

Me:

Gara Pruesse

Vancouver Island University

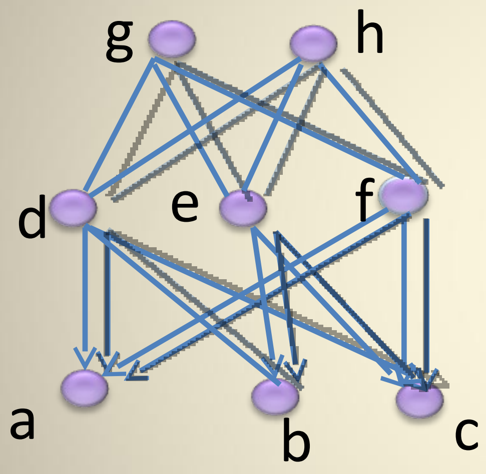
Coauthors:

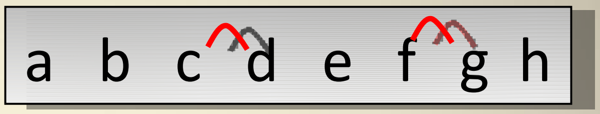
Derek Corneil

Lalla Mouatadid

University of Toronto

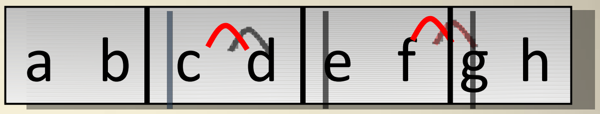
2-Processor Schedules

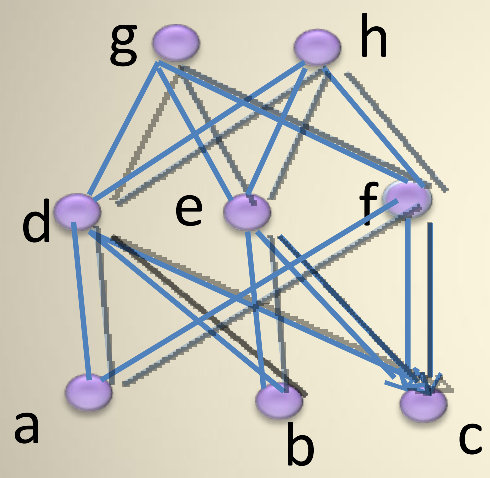




Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

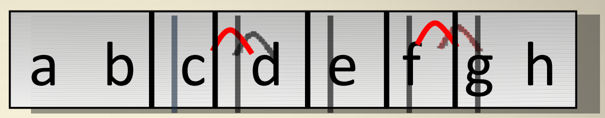
2-Processor Schedules

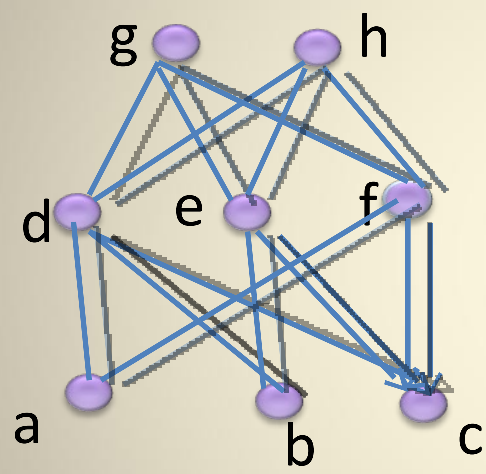




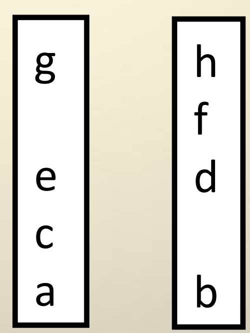
Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

2-Processor Schedules

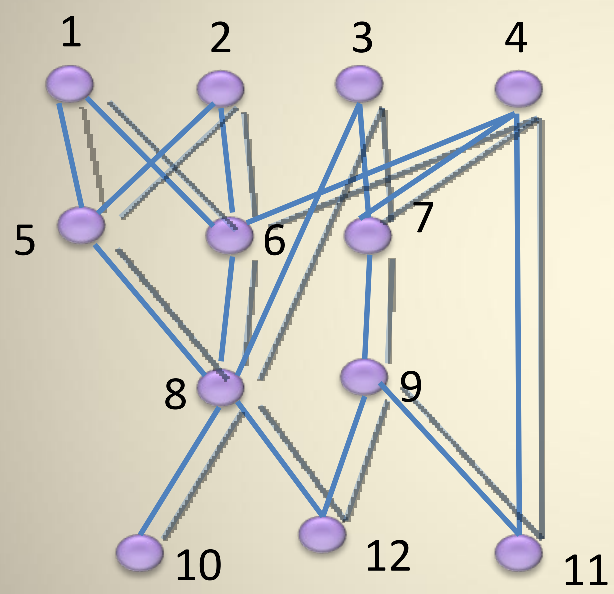




Want to schedule these unit-length jobs on two identical processor so that no job is executed before all of its lower covers have completed execution.

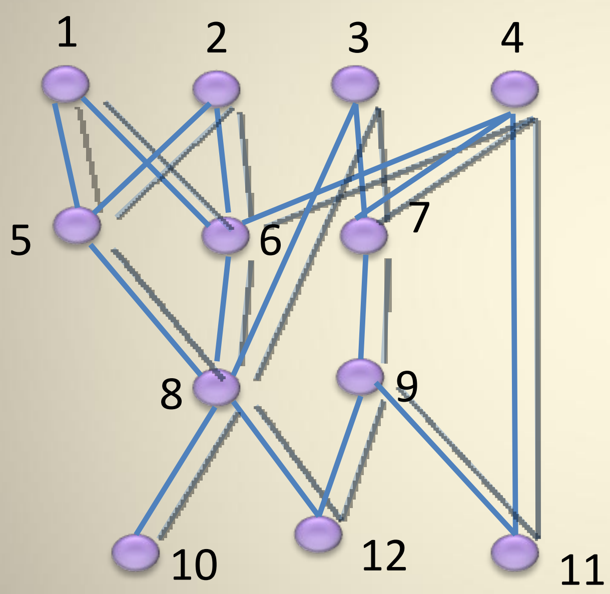


Coffman-Graham Lexicographic Labelling



- 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

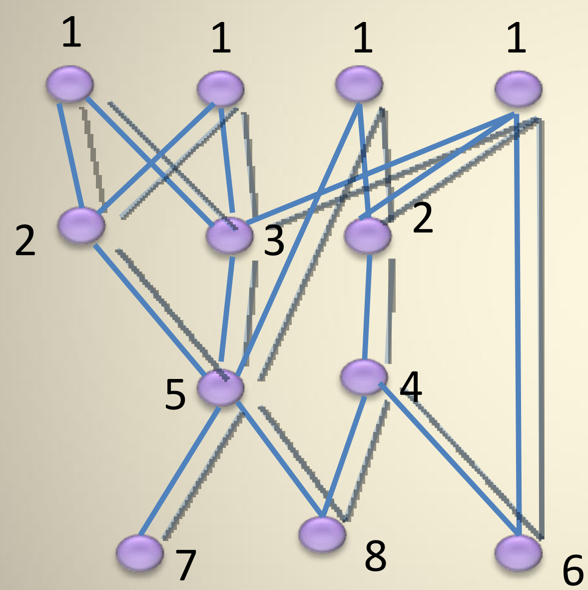
Coffman-Graham Lexicographic Labelling



- 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, 1986) 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 $O(n\ \alpha(n) + m))$