Parameterized Complexity and its Applications in Practice – From Foundations to Implementations –

Johannes K. Fichte

TU Dresden, Germany

Jakarta, Indonesia

Summer 2019 (May 6th – May 16th)

About me

09: Diploma in Theoretical CS with Minor in Physics from HU Berlin, GER under supervision of Martin Grohe

11– : Industry work on ...

15: PhD on Parameterized Complexity and ASP from TU Vienna, AT under supervision of Stefan Szeider (TUV) and Torsten Schaub (Potsdam)

15–18: PostDoc on "Decodyn: Treating Hard Problems with Decomposition and DP"; TU Vienna, AT

18–22: PostDoc at TU Dresden working on parameterized algorithms

Figure: Me @ NOT (Northern Optical Telescope)

Course Format and Topics

Topic:

Foundations of parameterized complexity and recent progress on applications of parameterized algorithmics to:

- Propositional Satisfiability (SAT)
- (Weighted) Model Counting (#SAT/WMC)
- Projected Model Counting (PMC)
- Constraint Satisfaction (CSP)

using backdoors and decomposition techniques.

Prerequisites:

Basic understanding of propositional logic. Preferably completed a Bachelor in mathematics or computer science or being close to doing so. Standard Bachelor lectures about algorithm design/analysis, discrete structures, logic, computational complexity are highly beneficial.

Dates:

Course is held from May 6 until May 17 Mo-Thu at 13:00–15:00.

Registration:

Drop by, right?

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

What's important for me?

Most important

- Ask questions!!!
- If you think something is unclear or you did not understand something (might be also because of the language), ask questions!!!
- If you think there might be a mistake, ask questions!!!

My Goals with the lecture.

- Make you interested in parameterized algorithms.
- Make you think yourself.
- It's NOT about finishing the entire content in time.
- Be mostly informal and give ideas
  (get you interested to read the more formal things)

Introduction

Purpose of the 1st Chapter

(Just) Motivation and Recall Basic Concepts

Introduction

Basics: Domains / Problems

Problem Domains of Interest

We consider

1. Graphs (mostly undirected)
2. Propositional Satisfiability (SAT)
3. Model Counting (#SAT)
4. Weighted Model Counting (WMC)
5. Projected Model Counting (PMC)
6. (Constraint Satisfaction)
7. (Answer Set Programming)

Who is familiar with ...?

No worries: We will introduce those problems in place when needed.

Problem Domains of Interest

We consider

1. Graphs (mostly undirected)
2. Propositional Satisfiability (SAT)
3. Model Counting (#SAT)
4. Weighted Model Counting (WMC)
5. Projected Model Counting (PMC)
6. (Constraint Satisfaction)
7. (Answer Set Programming)

Who is familiar with ...?

No worries: We will introduce those problems in place when needed.

Problem Domains of Interest

We consider

1. Graphs (mostly undirected)
2. Propositional Satisfiability (SAT)
3. Model Counting (#SAT)
4. Weighted Model Counting (WMC)
5. Projected Model Counting (PMC)
6. (Constraint Satisfaction)
7. (Answer Set Programming)

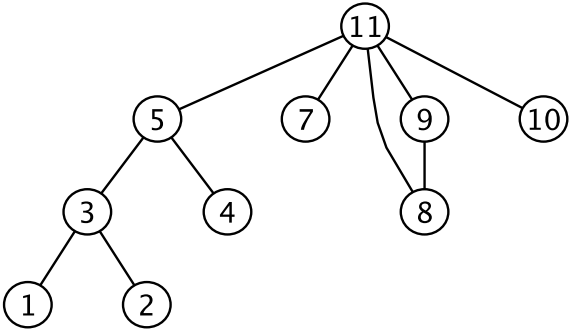
Who is familiar with ...?

No worries: We will introduce those problems in place when needed.

Domains

Graphs

An (undirected) graph G consists of a tuple (V, E) where V is the set of vertices and $E \subseteq \{\, \{u, v\} \subseteq V : u \neq v \,\}$ a set of edges.



How does a problem look like?

Problem

Input:	A graph $G = (V, E)$.
Output:	Decide (yes/no) whether X is true.
Measurement:	Number of vertices+edges (size $\|G\|$).

A Simple Graph Problem

Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover.

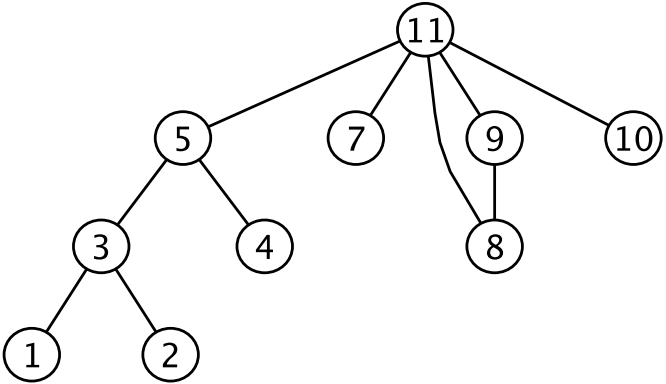
What's a vertex cover?

$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S

Example of a vertex cover

Definition

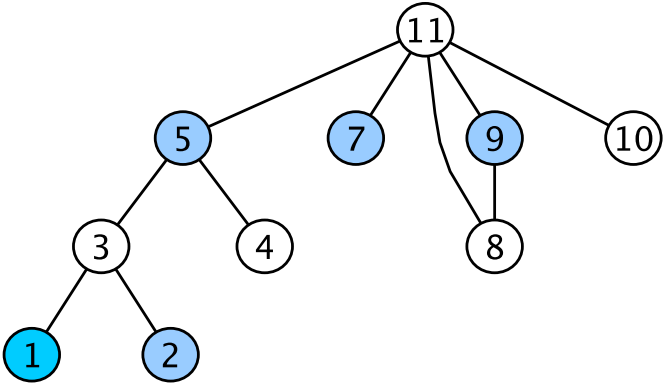
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



Example of a vertex cover

Definition

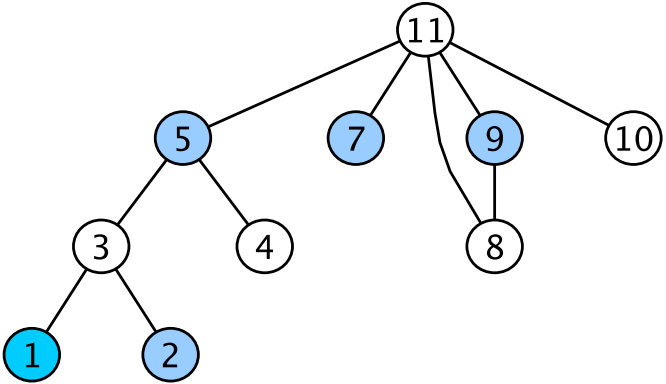
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



Example of a vertex cover

Definition

$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.

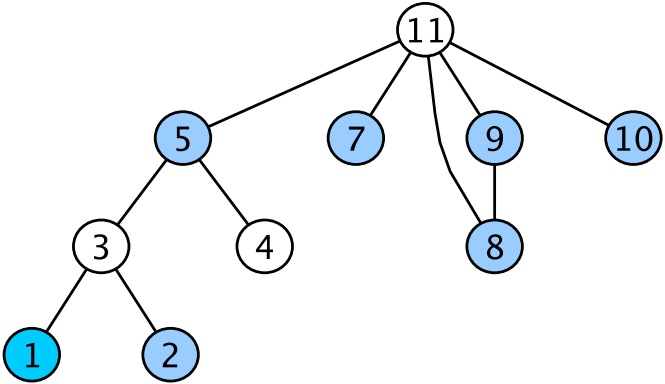


Well. Not quite.
Because $\{11, 10\}$ and $\{11, 8\}$ are not "covered".
$\Rightarrow$ Add 11 or (8 and 10)

Example of a vertex cover

Definition

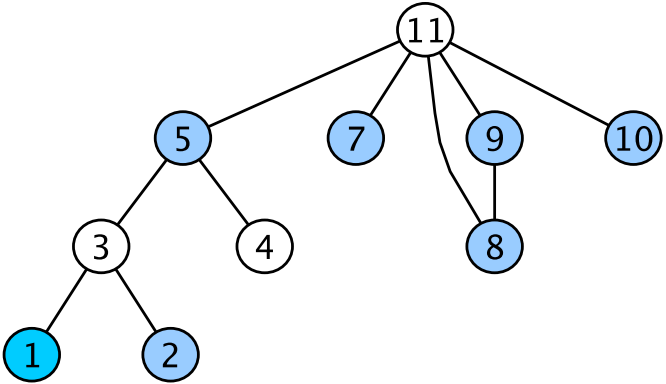
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



Example of a vertex cover

Definition

$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



Now we are good.

$\{1, 3\}$ $(1 \in S)$,
$\{2, 3\}$ $(2 \in S)$,
$\{3, 5\}$ $(5 \in S)$,
$\{4, 5\}$ $(5 \in S)$,
$\{5, 11\}$ $(5 \in S)$,
$\{7, 11\}$ $(7 \in S)$,
$\{8, 11\}$ $(8 \in S)$,
$\{9, 11\}$ $(9 \in S)$,
$\{10, 11\}$ $(10 \in S)$

A Simple Graph Problem

Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

No. And let me tell you why?

A Simple Graph Problem

Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

No. And let me tell you why?

A Simple Graph Problem

Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

No. And let me tell you why?

A Simple Graph Problem

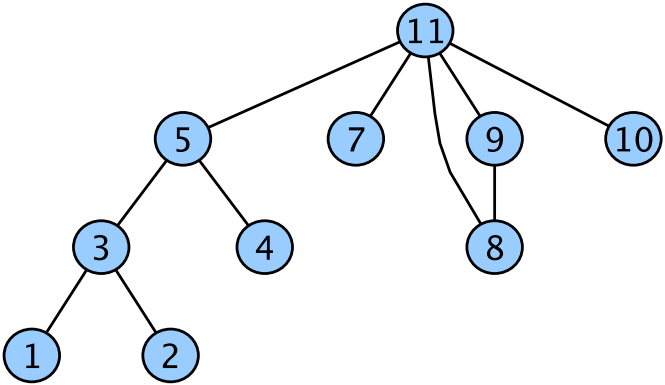
Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover.
Measure:	Number of vertices + edges (size of the graph).

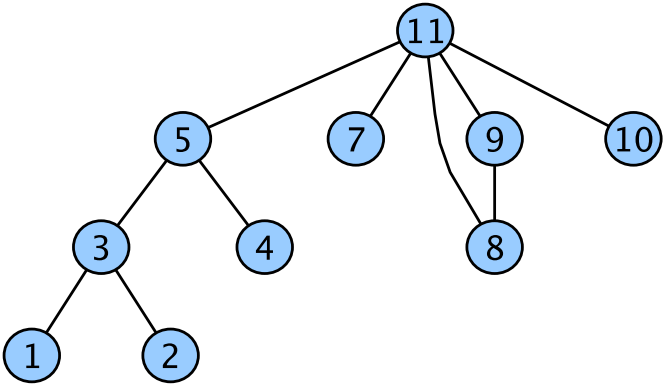
Is it hard to find a vertex cover?

No. And let me tell you why?

Trivial Vertex Cover



Trivial Vertex Cover

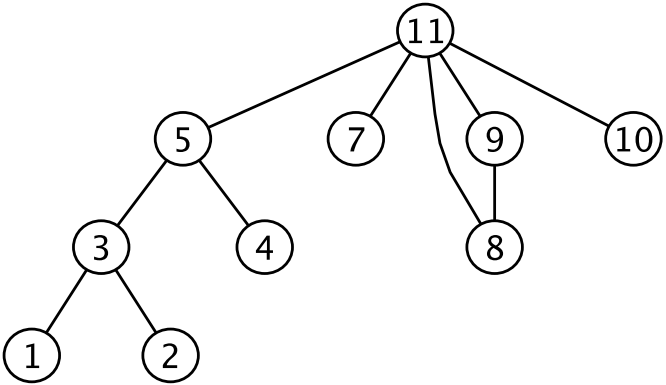


Trivial vertex cover.

$\Rightarrow$ Done.

Can also be done in linear time.
Hence, easy.

Trivial Vertex Cover



Just asking for a vertex cover is boring.
$\Rightarrow$ More interesting: ask for a vertex cover of size at most k or a minimum vertex cover.

A Simple Graph Problem

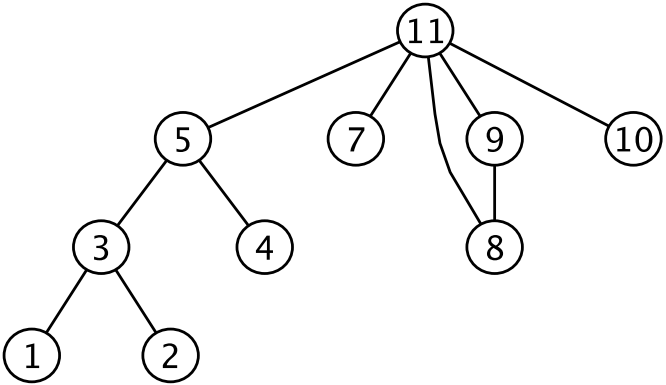
k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

It's known to be NP-complete on arbitrary graphs (about that we'll see in a bit).

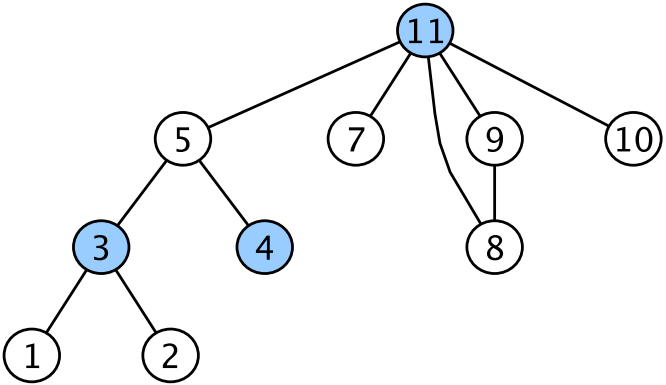
Vertex Cover: Let's try again...



Does the graph have a vertex cover of size at most 3?

Yes.

Vertex Cover: Let's try again...



Does the graph have a vertex cover of size at most 3?

Yes.

A Simple Graph Problem

k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

Best known theoretical algorithm runs in: time $\mathcal{O}(1.2738^k + kn)$ [**?**]
where n.. number of vertices and k size of the vertex cover

Can we do better?

It's known to be NP-complete on arbitrary graphs
(therefore we need some theory).

A Simple Graph Problem

k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

Best known theoretical algorithm runs in: time $\mathcal{O}(1.2738^k + kn)$ [**?**]
where n.. number of vertices and k size of the vertex cover

Can we do better?

It's known to be NP-complete on arbitrary graphs
(therefore we need some theory).

A Simple Graph Problem

k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

Best known theoretical algorithm runs in: time $\mathcal{O}(1.2738^k + kn)$ [**?**]
where n.. number of vertices and k size of the vertex cover

Can we do better?

It's known to be NP-complete on arbitrary graphs
(therefore we need some theory).

Introduction

Basics: Some words on Classical Complexity and Notations

Classical Complexity

What do we measure?

- Runtime
- Space

depending on the size of the input (actual length of the input instance).

What cases are interesting?

- Behavior on any input instance (hence: arbitrarily complicated)
- Best/worst case

  Why not average case? well...

  i) What's the average input?

  ii) Anyhow, the worst case is already complicated enough (for now).

References: [**?**, **?**]

Classical Complexity

What do we measure?

- Runtime
- Space

depending on the size of the input (actual length of the input instance).

What cases are interesting?

- Behavior on any input instance (hence: arbitrarily complicated)
- Best/worst case

  Why not average case? well...

  i) What's the average input?

  ii) Anyhow, the worst case is already complicated enough (for now).

References: [**?**, **?**]

Classical Complexity

What do we measure?

- Runtime
- Space

depending on the size of the input (actual length of the input instance).

What cases are interesting?

- Behavior on any input instance (hence: arbitrarily complicated)
- Best/worst case
  Why not average case? well...
  i) What's the average input?
  ii) Anyhow, the worst case is already complicated enough (for now).

References: [**?**, **?**]

Remember

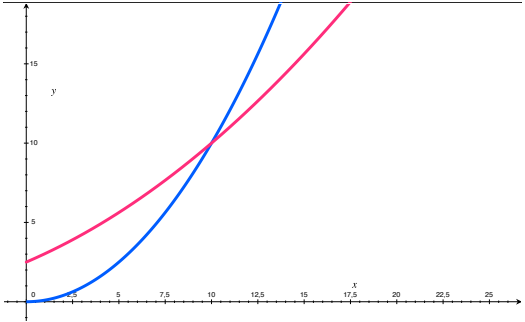
The idea is important in the following...

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. blue; g..red

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

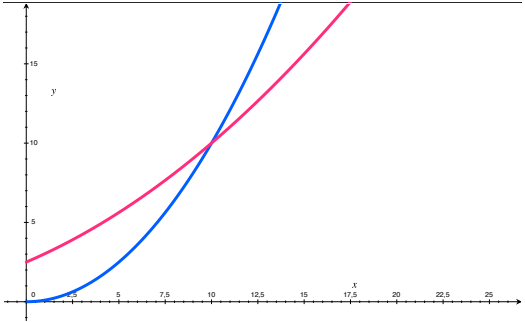
a) f is bounded above by g (at some point f grows faster)

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. blue; g..red

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

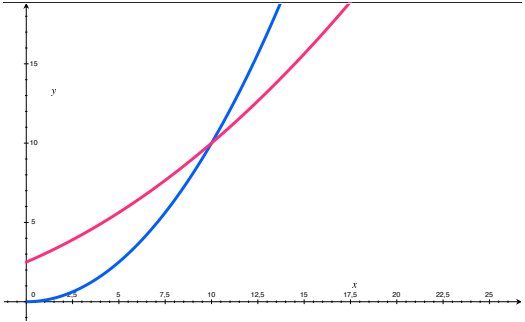
a) f is bounded above by g (at some point f grows faster)

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. blue; g..red

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

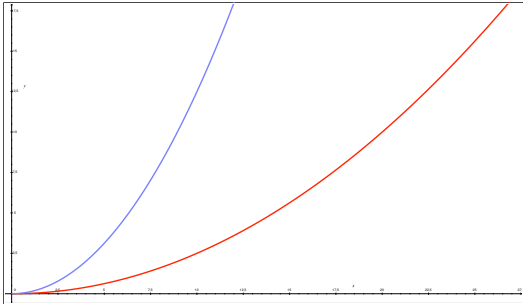
a) f is bounded above by g (at some point f grows faster)

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. left; g..right

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

both... why???
$f = \frac{n^2}{40}$ and $g = \frac{5n^2}{40}$

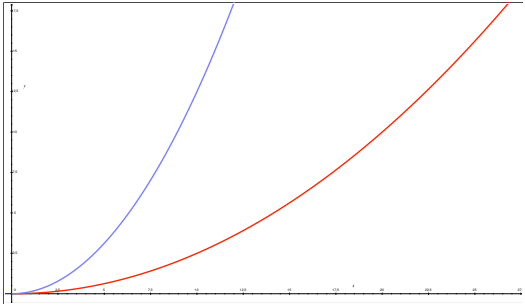
$\Rightarrow$ You don't necessarily see it immediately from a plot.

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. left; g..right

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

both... why???
$f = \frac{n^2}{40}$ and $g = \frac{5 \cdot n^2}{40}$

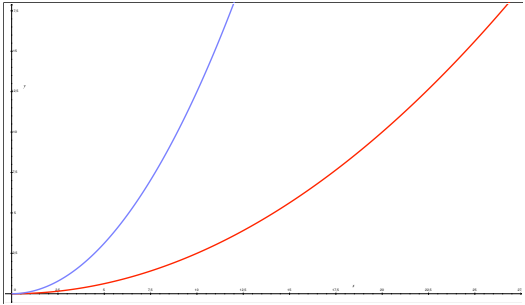
$\Rightarrow$ You don't necessarily see it immediately from a plot.

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. left; g..right

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

both... why???
$f = \frac{n^2}{40}$ and $g = \frac{5 \cdot n^2}{40}$

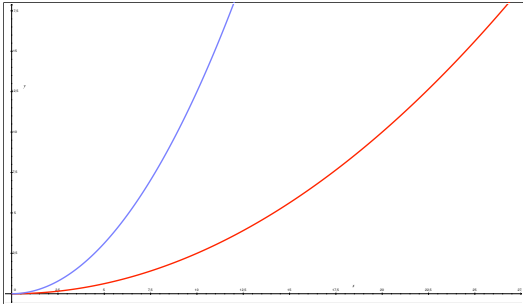
$\Rightarrow$ You don't necessarily see it immediately from a plot.

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

BigO (Big Oh; Big Omicron)

$f \in \mathcal{O}(g(n))$: means f is bounded above by g (up to constant factor) asymptotically (formally, $\exists k > 0\ \exists n_0 : \forall n > n_0 : f(n) \leq k \cdot g(n)$)



f.. left; g..right

Which one is bounded?
a) $f \in \mathcal{O}(g)$ or
b) $g \in \mathcal{O}(f)$

both... why???
$f = \frac{n^2}{40}$ and $g = \frac{5 \cdot n^2}{40}$

$\Rightarrow$ You don't necessarily see it immediately from a plot.

Standard Notations

Notation	Name
$\mathcal{O}(1)$	constant
$\mathcal{O}(n \log n)$	quasi-linear
$\mathcal{O}(n^2)$	quadratic
$\mathcal{O}(n^c)$	polynomial
$\mathcal{O}(c^n)$	exponential
$\mathcal{O}(n!)$	factorial

Remember!

k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

Best known theoretical algorithm runs in: time $\mathcal{O}(1.2738^k + kn)$ [**?**]
where n.. number of vertices and k size of the vertex cover

Is it hard to find a vertex cover?

It's known to be NP-complete on arbitrary graphs
(therefore we need some theory).

Landau-Notations/Asymptotic Notations

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.

Small o (Small oh)

$f \in o(g)$: means f is dominated by g asymptotically
formally, $\forall k > 0 : \exists n_0 : \forall n > n_0 : f(n) < k \cdot g(n)$

Big Omega

$f \in \Omega(g)$: means f is bounded below by g asymptotically
formally, $\exists k > 0 : \exists n_0 : \forall n > n_0 : f(n) \geq k \cdot g(n)$

Big Theta

$f \in \theta(g)$: means f is bounded both above and below by g asymptotically
formally, $f \in \mathcal{O}(g)$ and $f \in \Omega(g)$

(Decision) Problems

Σ_1 and Σ_2 finite alphabets

- $I_1 \in (\Sigma_1)^*$ an instance
- $\|I_1\|$ denotes the size of I_1

Let $L_1 \subseteq (\Sigma_1)^*$ and $L_2 \subseteq (\Sigma_2)^*$ be problems

- Instance $I \in L$ a yes-instance
- Instance $I \notin L$ a no-instance

$X^* = \bigcup_{i \geq 0} X^i = X^0 \cup X^1 \cup X^2 \cup X^3 \cup X^4 \cup \cdots.$

Reductions from L_1 to L_2

L_1, L_2 problems:

(non-deterministic) polynomial-time Turing reduction (Cook reduction) answers problem L_1 in polynomial by using problem L_2 as an oracle

formally:

1. (non-deterministic) algorithm that
2. decides in time $\mathcal{O}(||I||^c)$ for some constant c
3. whether $I \in L_1$ using L_2 as an oracle.

L_1, L_2 decision problems:

Polynomial-time many-to-one reduction (Karp reductions) transforms instances of decision problem L_1 into instances of decision problem L_2 in polynomial time

Note: single oracle call (considering L_2 as an oracle)

$\Rightarrow$ Conditions for Karp reductions stronger than for Cook reductions

formally:

1. function $r : (\Sigma_1)^* \to (\Sigma_2)^*$ such that
2. for all $I \in (\Sigma_1)^*$ we have $I \in L_1$ if and only if $r(I) \in L_2$ and
3. r is computable in time $\mathcal{O}(||I||^c)$ for some constant c.

Reductions from L_1 to L_2

L_1, L_2 problems:

(non-deterministic) polynomial-time Turing reduction (Cook reduction) answers problem L_1 in polynomial by using problem L_2 as an oracle

formally:

1. (non-deterministic) algorithm that
2. decides in time $\mathcal{O}(||I||^c)$ for some constant c
3. whether $I \in L_1$ using L_2 as an oracle.

L_1, L_2 decision problems:

Polynomial-time many-to-one reduction (Karp reductions) transforms instances of decision problem L_1 into instances of decision problem L_2 in polynomial time

Note: single oracle call (considering L_2 as an oracle)

$\Rightarrow$ Conditions for Karp reductions stronger than for Cook reductions

formally:

1. function $r : (\Sigma_1)^* \to (\Sigma_2)^*$ such that
2. for all $I \in (\Sigma_1)^*$ we have $I \in L_1$ if and only if $r(I) \in L_2$ and
3. r is computable in time $\mathcal{O}(||I||^c)$ for some constant c.

Complexity Classes

L: polynomial-time solvable if

1. decide by an algorithm whether $I \in L$ (yes/no)
2. in polynomial-time

non-deterministically solvable if it's a non-deterministic algorithm

L decision problem $\Rightarrow$ identify L with the set of all yes-instances I

Important Classes

- P: all polynomial-time solvable decision problems
- NP: all non-deterministically polynomial-time solvable decision problems
- co-C: all decision problems whose complement (the same problem with yes and no answers swapped) is in C, e.g. co-NP.

Hardness/Completeness for Complexity Class

Problem L

1. NP-hard if for every problem $L_2 \in$ NP there is a polynomial-time reduction to L
2. NP-complete if $L \in$ NP and NP-hard

Extends to other complexity classes C

(C-hard, C-complete)

Remember!

k-Vertex Cover Problem

Input:	A graph $G = (V, E)$.
Output:	Decide whether G has a vertex cover of size at most k.
Measure:	Number of vertices + edges (size of the graph).

Is it hard to find a vertex cover?

Best known theoretical algorithm runs in: time $\mathcal{O}(1.2738^k + kn)$ [?]
where n.. number of vertices and k size of the vertex cover

Is it hard to find a vertex cover?

It's known to be NP-complete on arbitrary graphs
(therefore we need some theory).

Implications from classical complexity?

We know:

k-Vertex Cover is NP-complete
$\Rightarrow$ similarly hard as many other problems

Is k-Vertex Cover always hard or can we do better?

Let's take a look on an example.

Implications from classical complexity?

We know:

k-Vertex Cover is NP-complete
$\Rightarrow$ similarly hard as many other problems

Is k-Vertex Cover always hard or can we do better?

Let's take a look on an example.

Implications from classical complexity?

We know:

k-Vertex Cover is NP-complete
$\Rightarrow$ similarly hard as many other problems

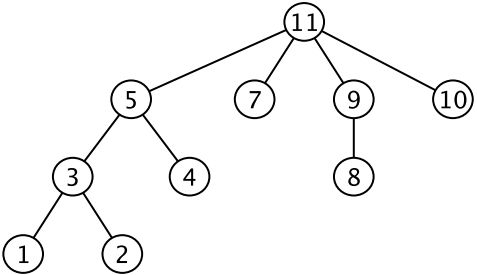
Is k-Vertex Cover always hard or can we do better?

Let's take a look on an example.

Example: Vertex Cover on Trees

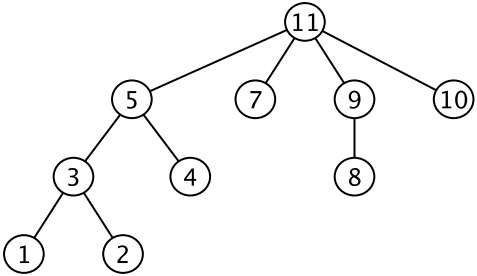
Assume we have a tree instead of an arbitrary graph.

Again: Is there a vertex cover of size at most 4?



Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?

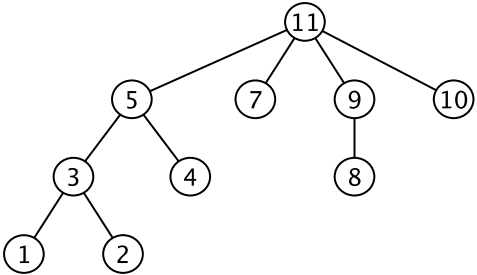


How can we find a vertex cover? Let's give it a try.

But, we start from the leaves (instead of the 101 algorithms examples).

Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?

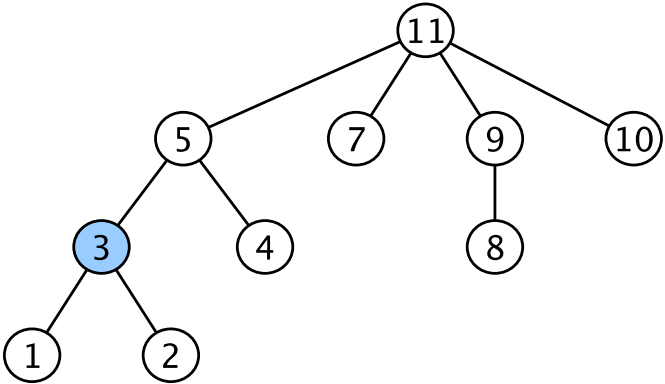


How can we find a vertex cover? Let's give it a try.

But, we start from the leaves (instead of the 101 algorithms examples).

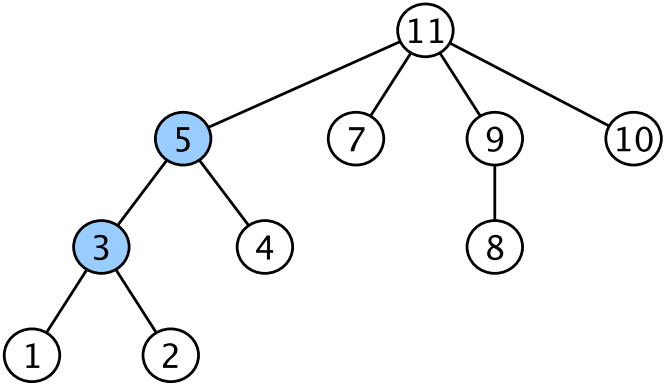
Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?



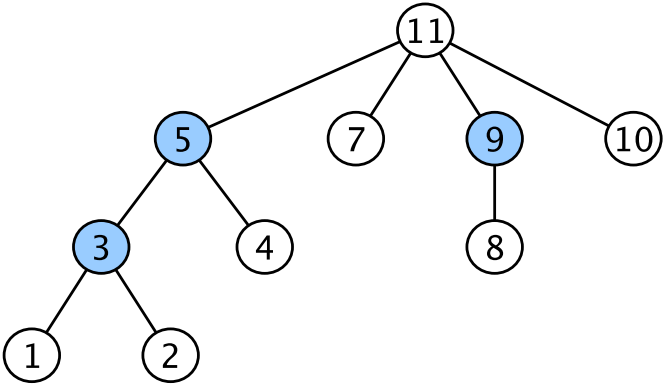
Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?



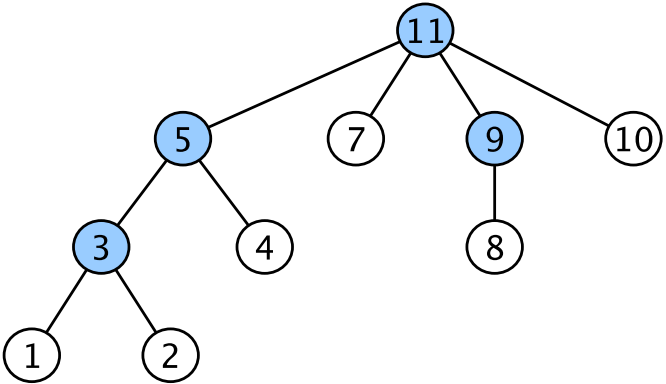
Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?



Example: Vertex Cover on Trees

Assume we have a tree instead of an arbitrary graph.
Again: Is there a vertex cover of size at most 4?



Example: Vertex Cover on Trees

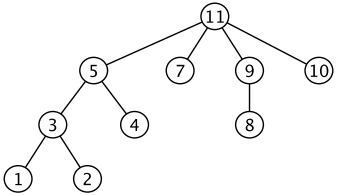
Again: Is there a vertex cover of size at most 4?

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
surprise 1,2,3,4,5, ...
at each node v recursively compute:

Let's generalize

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$ (we put the solution into S)

Traverse tree in post-order (LRN)

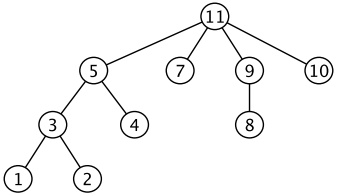
surprise 1,2,3,4,5, . . .

at each node v recursively compute:

1. ...

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN= left, right, node)

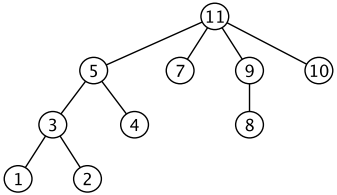
surprise 1,2,3,4,5, . . .

at each node v recursively compute:

1. ...

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

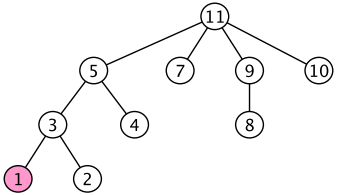
surprise 1,2,3,4,5, ...

at each node v recursively compute:

1. ...

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



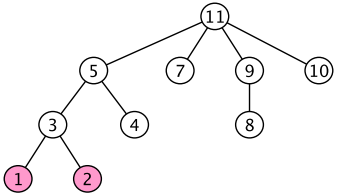
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

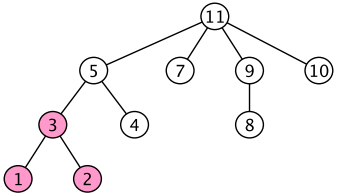
Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



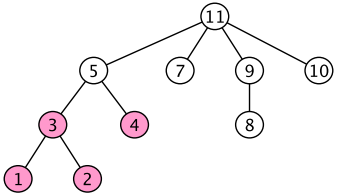
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



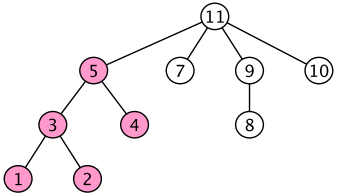
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



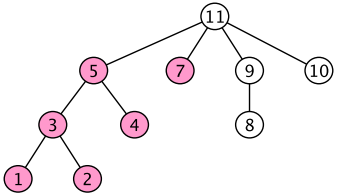
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



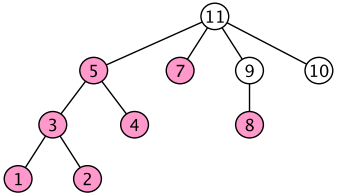
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



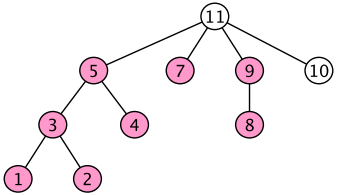
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



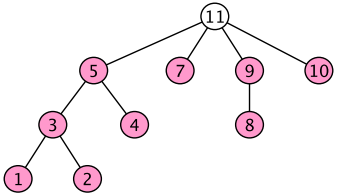
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



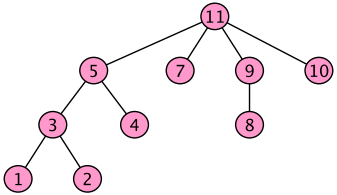
Do dynamic programming. Let $S = \emptyset$

Traverse tree in post-order (LRN)

1,2,3,4,5, . . .

Example: Vertex Cover on Trees

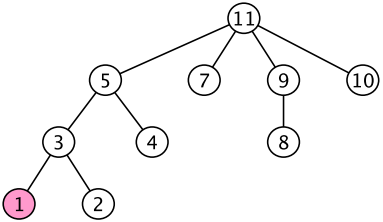
Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
1,2,3,4,5, ...

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?

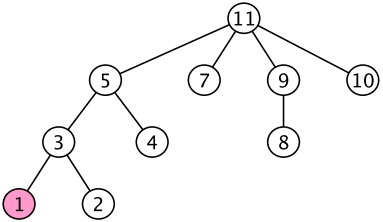


Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?

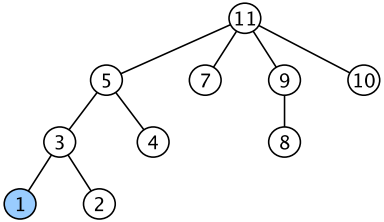


Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$ (Both options possible, since we have no edge at the leaves and $|S_v| \leq 4$)
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



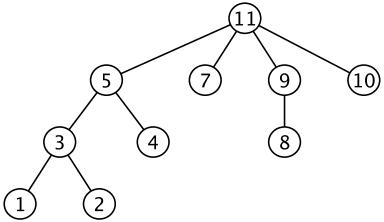
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$ (Both options possible, since we have no edge at the leaves and $|S_v| \leq 4$)
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



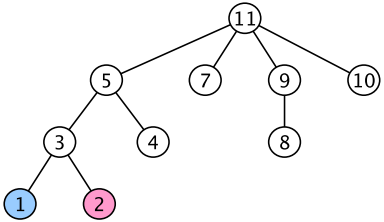
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$ Hence: $\mathcal{S}_1 := \{\{1\}, \emptyset\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

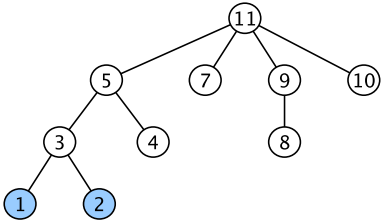
i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

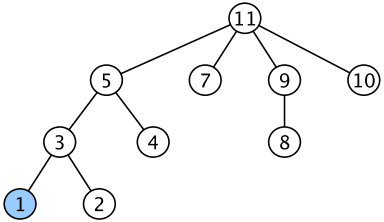
i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

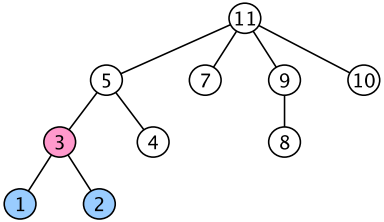
i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

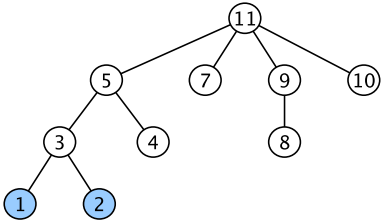
i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

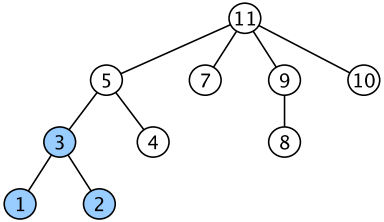
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

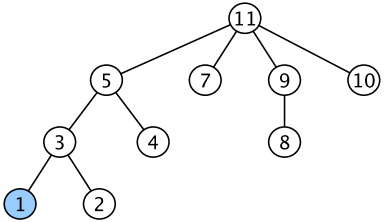
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

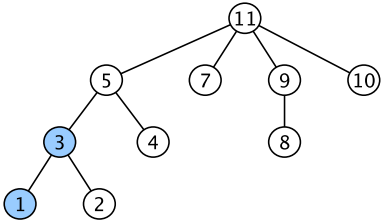
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

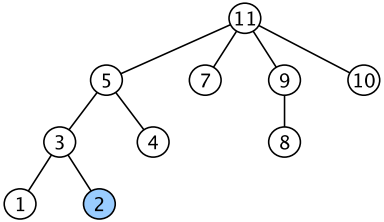
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

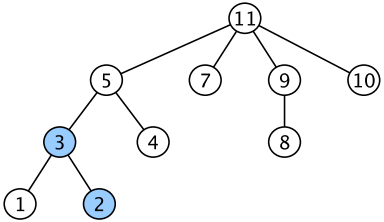
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

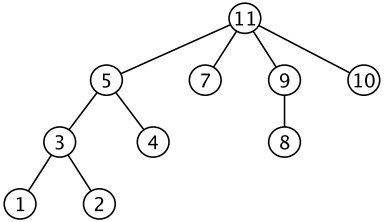
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

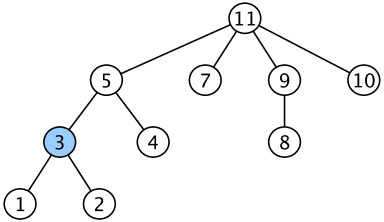
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

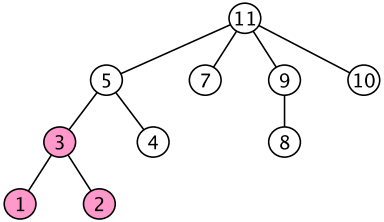
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

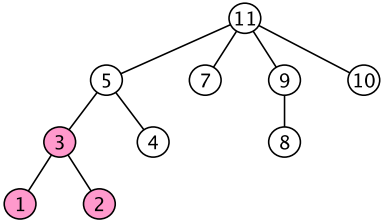
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.3	$S_{1.1}$	$S_{2.2}$	$\{\}$	$\{1\}$	1	✓	✗
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.5	$S_{1.2}$	$S_{2.1}$	$\{\}$	$\{2\}$	1	✓	✗
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.7	$S_{1.2}$	$S_{2.2}$	$\{\}$	$\{\}$	0	✓	✗
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

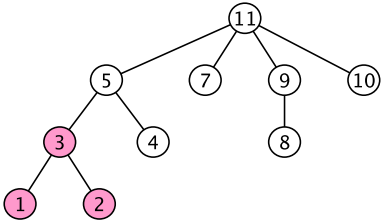
What's possible:

i	L	R	N	S_i	c	Size	VC[a]
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2	✓	✓
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3	✓	✓
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2	✓	✓
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2	✓	✓
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1	✓	✓

[a] for $T[\{1,2,3\}]$

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

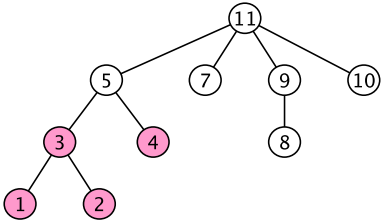
1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Remember it for node 3 (keep only min):

i	L	R	N	S_i	c
3.1	$S_{1.1}$	$S_{2.1}$	$\{\}$	$\{1,2\}$	2
3.2	$S_{1.1}$	$S_{2.1}$	$\{3\}$	$\{1,2,3\}$	3
3.4	$S_{1.1}$	$S_{2.2}$	$\{3\}$	$\{1,3\}$	2
3.6	$S_{1.2}$	$S_{2.1}$	$\{3\}$	$\{2,3\}$	2
3.8	$S_{1.2}$	$S_{2.2}$	$\{3\}$	$\{3\}$	1

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 3 +4:

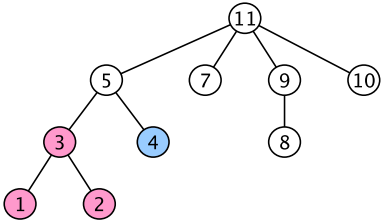
i	$S_{3.x}\|3$	c
3.1	$\{\}$	2
3.8	$\{3\}$	1

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 3 +4:

i	$S_{3.x}\|3$	c
3.1	$\{\}$	2
3.8	$\{3\}$	1

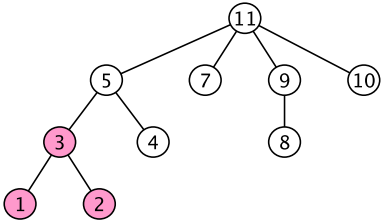
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{4.x}$	c
4.1	$\{4\}$	1
4.2	$\emptyset$	0

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 3 +4:

i	$S_{3.x}\|3$	c
3.1	$\{\}$	2
3.8	$\{3\}$	1

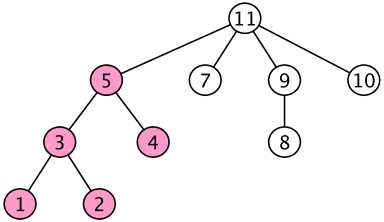
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{4.x}$	c
4.1	$\{4\}$	1
4.2	$\emptyset$	0

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 3 +4:

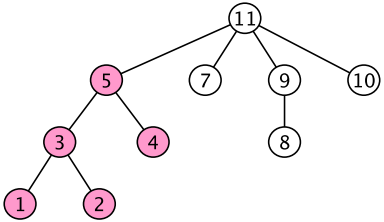
i	$S_{3.x}\|3$	c	i	$S_{4.x}$	c
3.1	$\{\}$	2	4.1	$\{4\}$	1
3.8	$\{3\}$	1	4.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 3 +4:

i	$S_{3.x}\|3$	c
3.1	$\{\}$	2
3.8	$\{3\}$	1

i	$S_{4.x}$	c
4.1	$\{4\}$	1
4.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

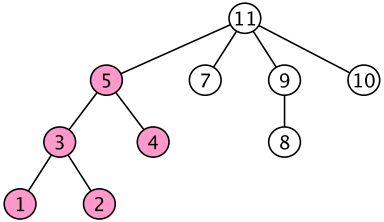
1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

What's possible:

i	L	R	N	S_i	c	Size	VC
5.1	$S_{3.1}$	$S_{4.1}$	$\{\}$	$\{4\}$	3	✓	✗
5.2	$S_{3.1}$	$S_{4.1}$	$\{5\}$	$\{4,5\}$	4	✓	✓
5.3	$S_{3.1}$	$S_{4.2}$	$\{\}$	$\{\}$	2	✓	✗
5.4	$S_{3.1}$	$S_{4.2}$	$\{5\}$	$\{3\}$	3	✓	✗
5.5	$S_{3.8}$	$S_{4.1}$	$\{\}$	$\{3,4\}$	2	✓	✓
5.6	$S_{3.8}$	$S_{4.1}$	$\{5\}$	$\{3,4,5\}$	3	✓	✓
5.7	$S_{3.8}$	$S_{4.2}$	$\{\}$	$\{3\}$	1	✓	✗
5.8	$S_{3.8}$	$S_{4.2}$	$\{5\}$	$\{3,5\}$	2	✓	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

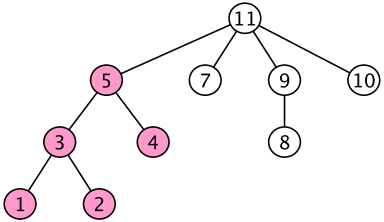
1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

What's possible:

i	L	R	N	S_i	c	Size	VC
5.1	$S_{3.1}$	$S_{4.1}$	$\{\}$	$\{4\}$	3	✓	✗
5.2	$S_{3.1}$	$S_{4.1}$	$\{5\}$	$\{4,5\}$	4	✓	✓
5.3	$S_{3.1}$	$S_{4.2}$	$\{\}$	$\{\}$	2	✓	✗
5.4	$S_{3.1}$	$S_{4.2}$	$\{5\}$	$\{3\}$	3	✓	✗
5.5	$S_{3.8}$	$S_{4.1}$	$\{\}$	$\{3,4\}$	2	✓	✓
5.6	$S_{3.8}$	$S_{4.1}$	$\{5\}$	$\{3,4,5\}$	3	✓	✓
5.7	$S_{3.8}$	$S_{4.2}$	$\{\}$	$\{3\}$	1	✓	✗
5.8	$S_{3.8}$	$S_{4.2}$	$\{5\}$	$\{3,5\}$	2	✓	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

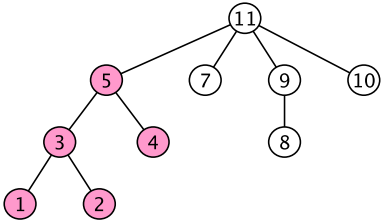
1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

What's possible:

i	L	R	N	S_i	c	Size	VC
5.2	$S_{3.1}$	$S_{4.1}$	$\{5\}$	$\{4,5\}$	4	✓	✓
5.5	$S_{3.8}$	$S_{4.1}$	$\{\}$	$\{3,4\}$	2	✓	✓
5.6	$S_{3.8}$	$S_{4.1}$	$\{5\}$	$\{3,4,5\}$	3	✓	✓
5.8	$S_{3.8}$	$S_{4.2}$	$\{5\}$	$\{3,5\}$	2	✓	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

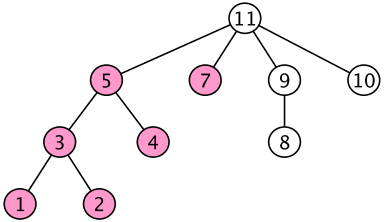
i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

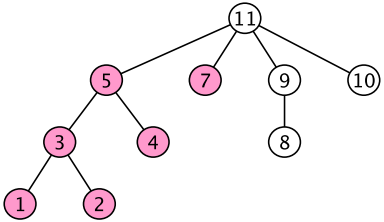
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

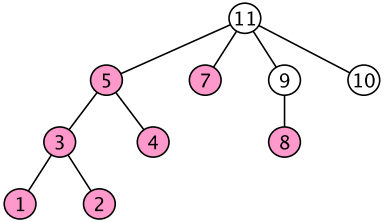
i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

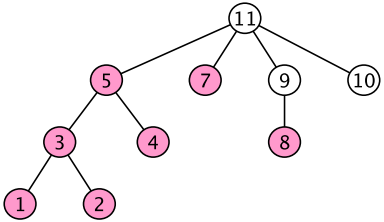
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{8.x}$	c
8.1	$\{8\}$	1
8.2	$\emptyset$	0

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

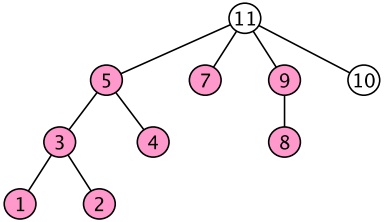
1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{8.x}$	c
8.1	$\{8\}$	1
8.2	$\emptyset$	0

i	L	N	$S_{9.x}$	c	$c \leq 4$	VC
9.1	$\{8\}$	$\{\}$	$\{8\}$	1	✓	✓
9.2	$\emptyset$	$\{\}$	$\{\}$	0	✓	✗
9.3	$\{8\}$	$\{9\}$	$\{8, 9\}$	1	✓	✓
9.4	$\emptyset$	$\{9\}$	$\{9\}$	0	✓	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

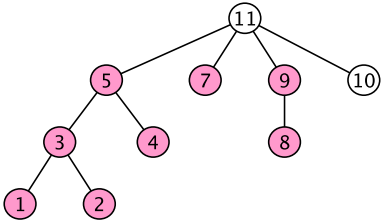
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	L	N	$S_{9.x}$	c	$c \leq 4$	VC
9.1	$\{8\}$	$\{\}$	$\{8\}$	1	✓	✓
9.2	$\emptyset$	$\{\}$	$\{\}$	0	✓	✗
9.3	$\{8\}$	$\{9\}$	$\{8, 9\}$	1	✓	✓
9.4	$\emptyset$	$\{9\}$	$\{9\}$	0	✓	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

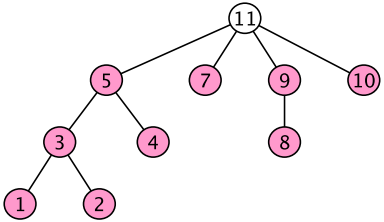
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

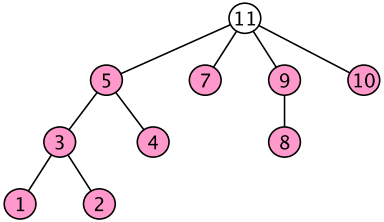
i	$S_{10.x}$	c
10.1	$\{\}$	0
10.4	$\{10\}$	1

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?

i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

i	$S_{10.x}$	c
10.1	$\{\}$	0
10.4	$\{10\}$	1



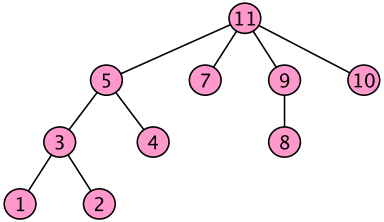
Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

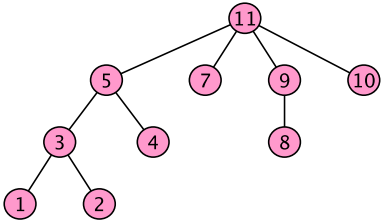
i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

i	$S_{10.x}$	c
10.1	$\{\}$	0
10.4	$\{10\}$	1

i	N	S_i	c	$c \leq 4$	VC
11.1	$\{\}$	$\{\}$	3	✓	✗
11.2	$\{\}$	$\{5\}$	3	✓	✗
11.3	$\{\}$	$\{7\}$	4	✗	✗
11.4	$\{\}$	$\{9\}$	3	✓	✗
11.5	$\{\}$	$\{10\}$	4	✓	✗
11.6	$\{\}$	$\{5,7\}$	4	✓	✗
11.7	$\{\}$	$\{5,9\}$	3	✓	✗
11.8	$\{\}$	$\{5,10\}$	4	✓	✗
11.9	$\{\}$	$\{7,9\}$	4	✓	✗
11.10	$\{\}$	$\{7,10\}$	5	✗	✗
11.11	$\{\}$	$\{9,10\}$	4	✓	✗
11.12	$\{\}$	$\{5,7,9\}$	4	✓	✗
11.13	$\{\}$	$\{5,7,10\}$	5	✗	✗
11.14	$\{\}$	$\{5,9,10\}$	4	✓	✗
11.15	$\{\}$	$\{7,9,10\}$	5	✗	✗
11.16	$\{\}$	$\{5,7,9,10\}$	5	✗	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

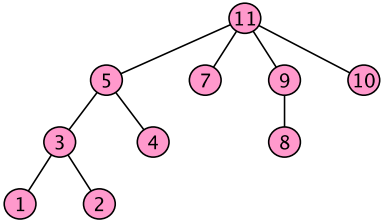
i	$S_{10.x}$	c
10.1	$\{\}$	0
10.4	$\{10\}$	1

i	N	S_i	c	$c \leq 4$	VC
11.1	$\{\}$	$\{\}$	3	✓	✗
11.2	$\{\}$	$\{5\}$	3	✓	✗
11.3	$\{\}$	$\{7\}$	4	✗	✗
11.4	$\{\}$	$\{9\}$	3	✓	✗
11.5	$\{\}$	$\{10\}$	4	✓	✗
11.6	$\{\}$	$\{5,7\}$	4	✓	✗
11.7	$\{\}$	$\{5,9\}$	3	✓	✗
11.8	$\{\}$	$\{5,10\}$	4	✓	✗
11.9	$\{\}$	$\{7,9\}$	4	✓	✗
11.10	$\{\}$	$\{7,10\}$	5	✗	✗
11.11	$\{\}$	$\{9,10\}$	4	✓	✗
11.12	$\{\}$	$\{5,7,9\}$	4	✓	✗
11.13	$\{\}$	$\{5,7,10\}$	5	✗	✗
11.14	$\{\}$	$\{5,9,10\}$	4	✓	✗
11.15	$\{\}$	$\{7,9,10\}$	5	✗	✗
11.16	$\{\}$	$\{5,7,9,10\}$	5	✗	✓

None work of those work. So take 11 into account.

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



Remember it for node 5 +7:

i	N	c
5.5	$\{\}$	2
5.8	$\{5\}$	2

i	$S_{7.x}$	c
7.1	$\{7\}$	1
7.2	$\emptyset$	0

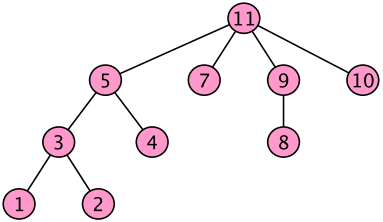
i	$S_{9.x}\|9$	c
9.1	$\{\}$	1
9.4	$\{9\}$	1

i	$S_{10.x}$	c
10.1	$\{\}$	0
10.4	$\{10\}$	1

i	N	S_i	c	$c \leq 4$	VC
11.1	$\{11\}$	$\{11\}$	4	✓	✓
11.2	$\{11\}$	$\{5, 11\}$	4	✓	✓
11.3	$\{11\}$	$\{7, 11\}$	5	✗	✓
11.4	$\{11\}$	$\{9, 11\}$	4	✓	✓
11.5	$\{11\}$	$\{10, 11\}$	5	✗	✓
11.6	$\{11\}$	$\{5, 7, 11\}$	5	✗	✓
11.7	$\{11\}$	$\{5, 9, 11\}$	4	✗	✓
11.8	$\{11\}$	$\{5, 10, 11\}$	5	✗	✓
11.9	$\{11\}$	$\{7, 9, 11\}$	5	✗	✓
11.10	$\{11\}$	$\{7, 10, 11\}$	6	✗	✓
11.11	$\{11\}$	$\{9, 10, 11\}$	5	✗	✓
11.12	$\{11\}$	$\{5, 7, 9, 11\}$	5	✗	✓
11.13	$\{11\}$	$\{5, 7, 10, 11\}$	6	✗	✓
11.14	$\{11\}$	$\{5, 9, 10, 11\}$	5	✗	✓
11.15	$\{11\}$	$\{7, 9, 10, 11\}$	6	✗	✓
11.16	$\{11\}$	$\{5, 7, 9, 10, 11\}$	6	✗	✓

Example: Vertex Cover on Trees

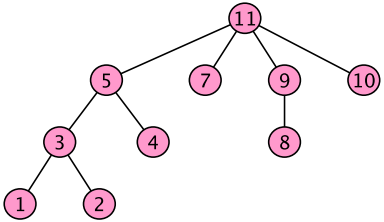
Again: Is there a vertex cover of size at most 4?



i	N	S_i	c	$c \leq 4$	VC
11.1	$\{11\}$	$\{11\}$	4	✓	✓
11.2	$\{11\}$	$\{5, 11\}$	4	✓	✓
11.3	$\{11\}$	$\{7, 11\}$	5	✗	✓
11.4	$\{11\}$	$\{9, 11\}$	4	✓	✓
11.5	$\{11\}$	$\{10, 11\}$	5	✗	✓
11.6	$\{11\}$	$\{5, 7, 11\}$	5	✗	✓
11.7	$\{11\}$	$\{5, 9, 11\}$	4	✗	✓
11.8	$\{11\}$	$\{5, 10, 11\}$	5	✗	✓
11.9	$\{11\}$	$\{7, 9, 11\}$	5	✗	✓
11.10	$\{11\}$	$\{7, 10, 11\}$	6	✗	✓
11.11	$\{11\}$	$\{9, 10, 11\}$	5	✗	✓
11.12	$\{11\}$	$\{5, 7, 9, 11\}$	5	✗	✓
11.13	$\{11\}$	$\{5, 7, 10, 11\}$	6	✗	✓
11.14	$\{11\}$	$\{5, 9, 10, 11\}$	5	✗	✓
11.15	$\{11\}$	$\{7, 9, 10, 11\}$	6	✗	✓
11.16	$\{11\}$	$\{5, 7, 9, 10, 11\}$	6	✗	✓

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
11.1	$\{11\}$	4
11.2	$\{5, 11\}$	4
11.4	$\{9, 11\}$	4

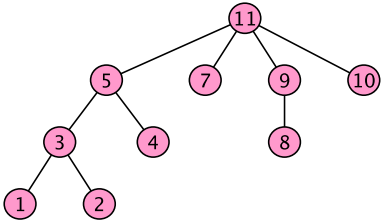
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Finally.
Check the root. If there are solutions left.
Then, we have a solution for our vertex cover problem.
BUT: only decision problem. For the outputting a solution we have to go top down again and reconstruct a solution.

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
11.1	$\{11\}$	4
11.2	$\{5, 11\}$	4
11.4	$\{9, 11\}$	4

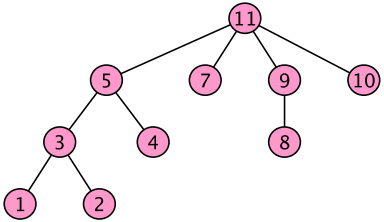
Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Finally.
Check the root. If there are solutions left.
Then, we have a solution for our vertex cover problem.
BUT: only decision problem. For the outputting a solution we have to go top down again and reconstruct a solution.
$\Rightarrow$ We know the answer to the vertex cover (at most k) problem.

Example: Vertex Cover on Trees

Again: Is there a vertex cover of size at most 4?



i	S_i	c
11.1	$\{11\}$	4
11.2	$\{5, 11\}$	4
11.4	$\{9, 11\}$	4

Do dynamic programming. Let $S = \emptyset$
Traverse tree in post-order (LRN)
at each node v recursively compute:

1. If v is a leaf: (i) $S_v := \{v\}$ and (ii) $S_v := \{\}$
2. If v is an inner node, we have already computed sets $S_a, S_b, S_c, \ldots$ for the children $a, b, c, \ldots$ of v
   Take set that is vertex cover and satisfies cardinality

Finally.
Check the root. If there are solutions left.
Then, we have a solution for our vertex cover problem.
BUT: only decision problem. For the outputting a solution we have to go top down again and reconstruct a solution.

Why did we make it so complicated?
Well, the example can almost trivially be extended to arbitrary graphs if we have a decomposition of a graph into a tree. But more about that in a sec.

Example: Vertex Cover on Graphs

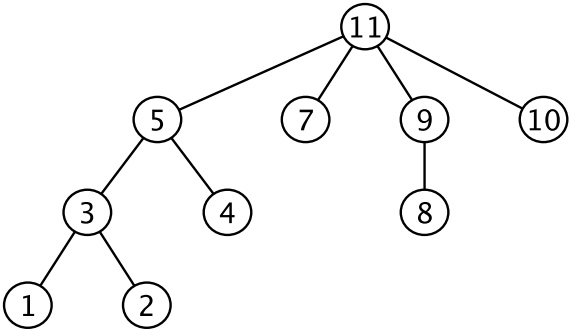
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.

Again: Is there a vertex cover of size at most 4?

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.

Again: Is there a vertex cover of size at most 4?

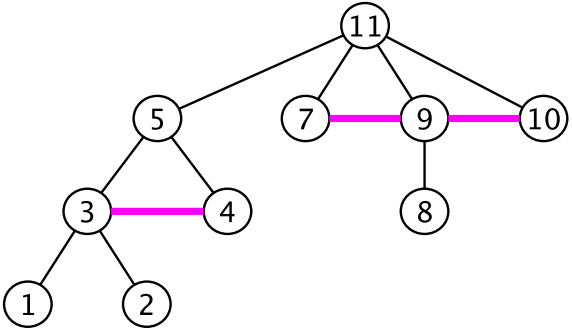


Old Graph (tree).

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.

Again: Is there a vertex cover of size at most 4?

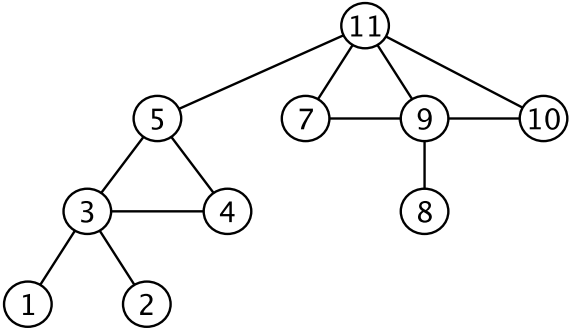


New Graph (close to the old tree, right?)

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.

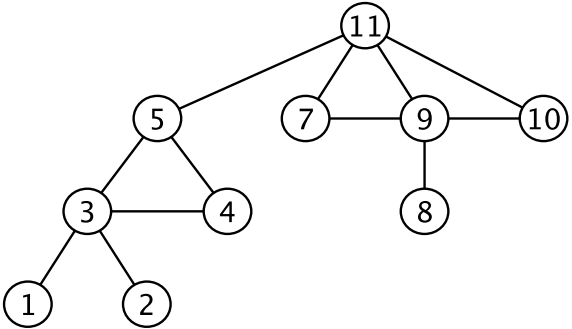
Again: Is there a vertex cover of size at most 4?



Let's test our previous set of vertices.

Example: Vertex Cover on Graphs

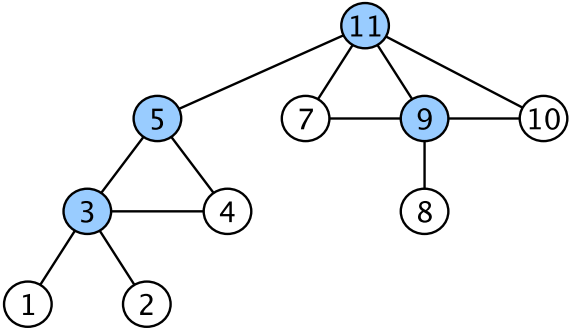
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Let's test our previous set of vertices.

Example: Vertex Cover on Graphs

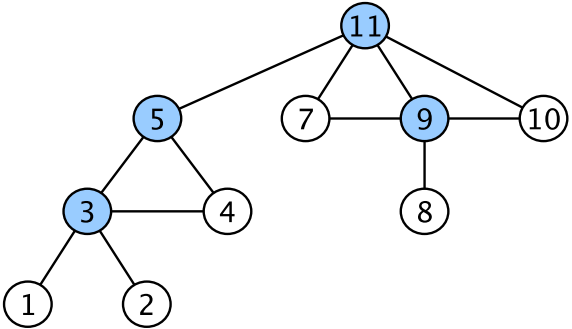
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Does the set of blue colored vertices form such a VC? YES!

Example: Vertex Cover on Graphs

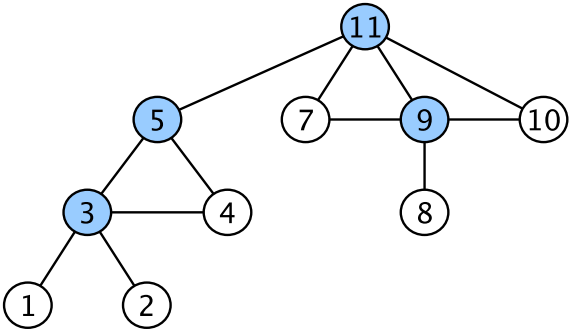
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Does the set of blue colored vertices form such a VC? YES!

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Does the set of blue colored vertices form such a VC? YES!

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?

Let's generalize the algorithm

(by considering the vertices that occur in a clique exhaustively at the same time.)

Clique. Huh?

Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?

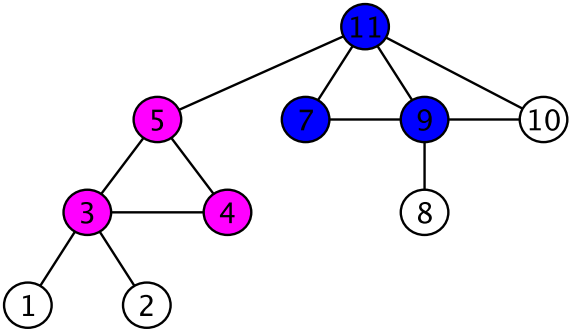
Let's generalize the algorithm

(by considering the vertices that occur in a clique exhaustively at the same time.)

Clique. Huh?

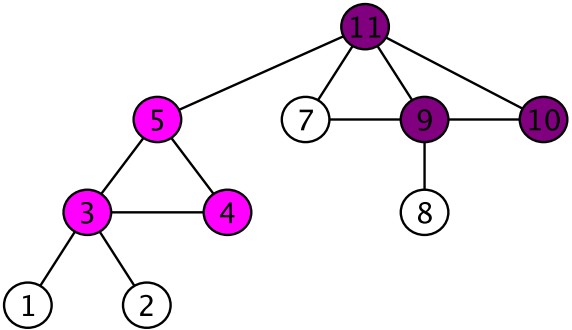
Example: Vertex Cover on Graphs

Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Example: Vertex Cover on Graphs

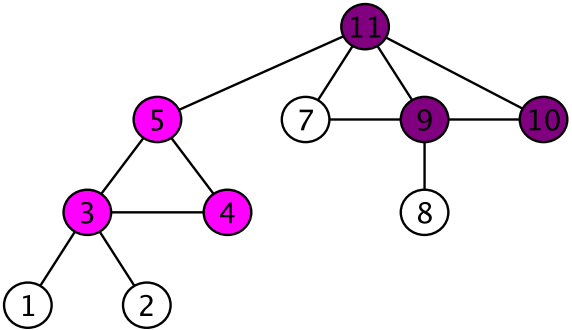
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Clique: Subset of the vertices such that every two distinct vertices are neighbors.

Example: Vertex Cover on Graphs

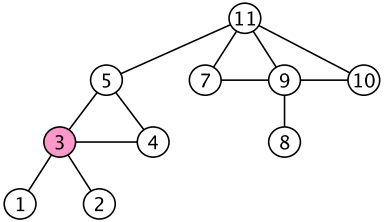
Now let's see whether we can solve it if the graph is not a tree, but close to a tree.
Again: Is there a vertex cover of size at most 4?



Clique: Subset of the vertices such that every two distinct vertices are neighbors.

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?

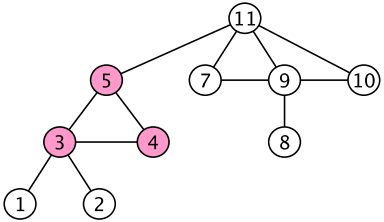


i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



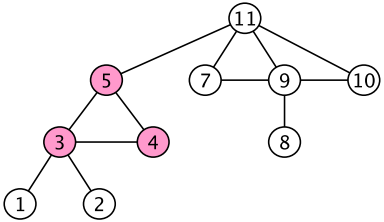
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.1	$\{\}$	0	✓	✗
3.2	$\{3\}$	1	✓	✗
3.3	$\{4\}$	1	✓	✗
3.4	$\{5\}$	1	✓	✗
3.5	$\{3,4\}$	2	✓	✗
3.6	$\{3,5\}$	2	✓	✗
3.7	$\{4,5\}$	2	✓	✗
3.8	$\{3,4,5\}$	3	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



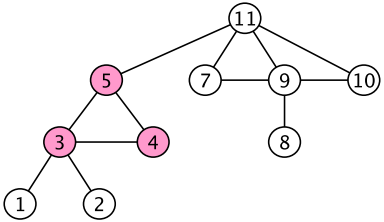
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.1	$\{\}$	0	✓	✗
3.2	$\{3\}$	1	✓	✗
3.3	$\{4\}$	1	✓	✗
3.4	$\{5\}$	1	✓	✗
3.5	$\{3,4\}$	2	✓	✗
3.6	$\{3,5\}$	2	✓	✗
3.7	$\{4,5\}$	2	✓	✗
3.8	$\{3,4,5\}$	3	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



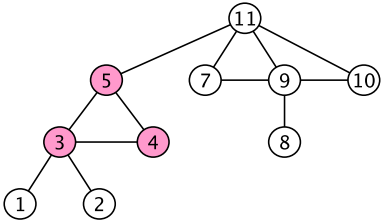
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.9	$\{1\}$	1	✓	✗
3.10	$\{1,3\}$	2	✓	✗
3.11	$\{1,4\}$	2	✓	✗
3.12	$\{1,5\}$	2	✓	✗
3.13	$\{1,3,4\}$	3	✓	✗
3.14	$\{1,3,5\}$	3	✓	✗
3.15	$\{1,4,5\}$	3	✓	✗
3.16	$\{1,3,4,5\}$	4	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



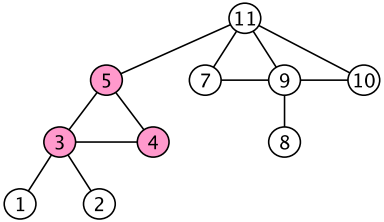
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.9	$\{1\}$	1	✓	✗
3.10	$\{1,3\}$	2	✓	✗
3.11	$\{1,4\}$	2	✓	✗
3.12	$\{1,5\}$	2	✓	✗
3.13	$\{1,3,4\}$	3	✓	✗
3.14	$\{1,3,5\}$	3	✓	✗
3.15	$\{1,4,5\}$	3	✓	✗
3.16	$\{1,3,4,5\}$	4	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



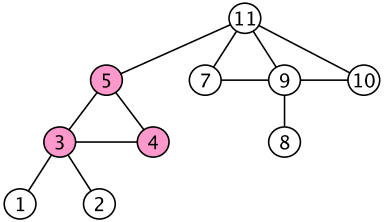
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.17	$\{2\}$	1	✓	✗
3.18	$\{2,3\}$	2	✓	✗
3.19	$\{2,4\}$	2	✓	✗
3.20	$\{2,5\}$	2	✓	✗
3.21	$\{2,3,4\}$	3	✓	✗
3.22	$\{2,3,5\}$	3	✓	✗
3.23	$\{2,4,5\}$	3	✓	✗
3.24	$\{2,3,4,5\}$	4	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



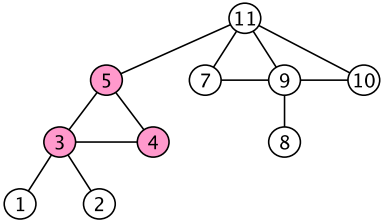
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.17	$\{2\}$	1	✓	✗
3.18	$\{2,3\}$	2	✓	✗
3.19	$\{2,4\}$	2	✓	✗
3.20	$\{2,5\}$	2	✓	✗
3.21	$\{2,3,4\}$	3	✓	✗
3.22	$\{2,3,5\}$	3	✓	✗
3.23	$\{2,4,5\}$	3	✓	✗
3.24	$\{2,3,4,5\}$	4	✓	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



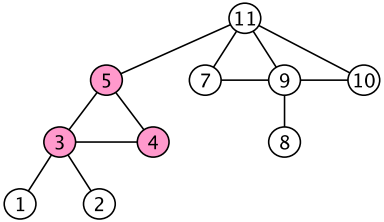
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.25	$\{1,2\}$	2	✓	✗
3.26	$\{1,2,3\}$	3	✓	✗
3.27	$\{1,2,4\}$	3	✓	✗
3.28	$\{1,2,5\}$	3	✓	✗
3.29	$\{1,2,3,4\}$	4	✓	✗
3.30	$\{1,2,3,5\}$	4	✓	✗
3.31	$\{1,2,4,5\}$	4	✓	✓
3.32	$\{1,2,3,4,5\}$	5	✗	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



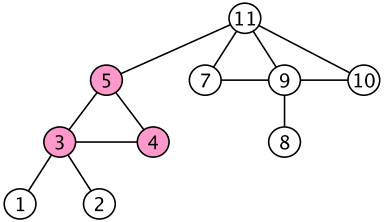
i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

i	S_i	c	Size	VC
3.25	$\{1,2\}$	2	✓	✗
3.26	$\{1,2,3\}$	3	✓	✗
3.27	$\{1,2,4\}$	3	✓	✗
3.28	$\{1,2,5\}$	3	✓	✗
3.29	$\{1,2,3,4\}$	4	✓	✗
3.30	$\{1,2,3,5\}$	4	✓	✗
3.31	$\{1,2,4,5\}$	4	✓	✓
3.32	$\{1,2,3,4,5\}$	5	✗	✓

Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



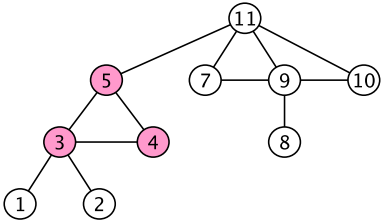
i	S_i	c	Size	VC
3.8	$\{3,4,5\}$	3	✓	✓
3.16	$\{1,3,4,5\}$	4	✓	✓
3.24	$\{2,3,4,5\}$	4	✓	✓
3.31	$\{1,2,4,5\}$	4	✓	✓

i	S_i	c
1.1	$\{1\}$	1
1.2	$\emptyset$	0

i	S_i	c
2.1	$\{1\}$	2
2.2	$\emptyset$	0

Example: Towards Vertex Cover on Graphs

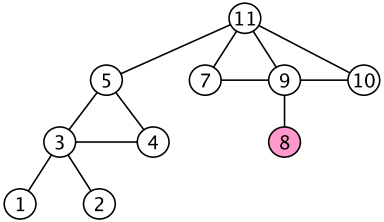
Again: Is there a vertex cover of size at most 4?



i	S_i	c	Size	VC
3.8	$\{3, 4, 5\}$	3	✓	✓

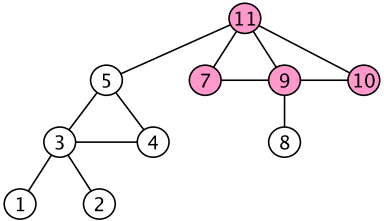
Example: Towards Vertex Cover on Graphs

Again: Is there a vertex cover of size at most 4?



i	$S_{8.x}$	c
8.1	$\{8\}$	1
8.2	$\emptyset$	0

Example: Towards Vertex Cover on Graphs



i	S_i	c
11.1	$\{\}$	1
11.2	$\{11\}$	1
11.3	$\{7\}$	1
11.4	$\{9\}$	1
11.5	$\{10\}$	1
...		

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?
   - Lesson: split graph into two, then discard one.

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?

   YES: decomposition of graphs into trees (tree decompositions)

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?

   YES: decomposition of graphs into trees (tree decompositions)

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?
   YES: decomposition of graphs into trees (tree decompositions)

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?
   YES: decomposition of graphs into trees (tree decompositions)

Example: Towards Vertex Cover on Graphs

Omit remaining part...

Take Home

1. If we consider the cliques exhaustively at the same time, should work.
2. What does it cost? 2^k
3. Issue: quite expensive right?
4. What if it's not a clique? Can we do better?
   YES: decomposition of graphs into trees (tree decompositions)

Introduction

From Trees to Graphs: Tree Decompositions

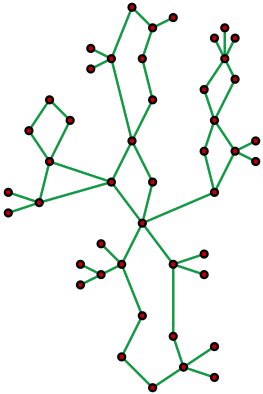
Treewidth

Motivation

- You've seen a problem, which can be solved efficiently on trees
- You've vague seen how to generalize things to graphs (based on cliques)
- There are many problems are (computationally) hard on arbitrary graphs but become much simpler on simple graphs such as trees.

From Trees to “tree-like” Graphs

- Is there a way to generalize from trees to more complex graphs and still retain the good properties of trees?
- In other words is there a way to capture how “tree-like” a graph is?

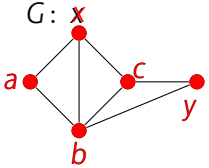


Tree Decompositions

Treewidth ▸ Definition & Example

- Most prominent graph invariant
- Small treewidth indicates tree-likeness and sparsity

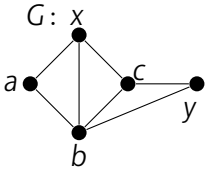
Tree Decompositions

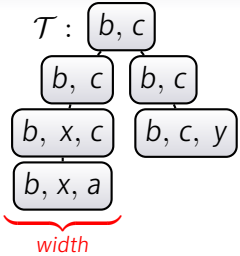


Treewidth ▸ Definition & Example

- Treewidth defined in terms of tree decompositions (TD)
- TD: arrangement of graph into a tree + bags s.t. ...
- Treewidth: width of a TD of smallest width

Tree Decompositions



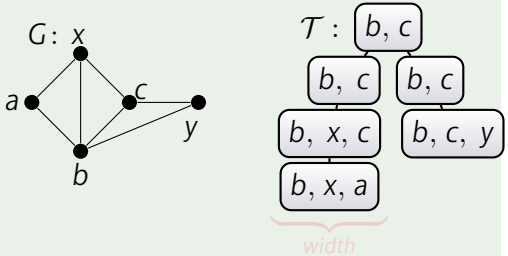


Treewidth ▸ Definition & Example

- Treewidth defined in terms of tree decompositions (TD)
- TD: arrangement of graph into a tree + bags s.t. ...
- Treewidth: width of a TD of smallest width

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



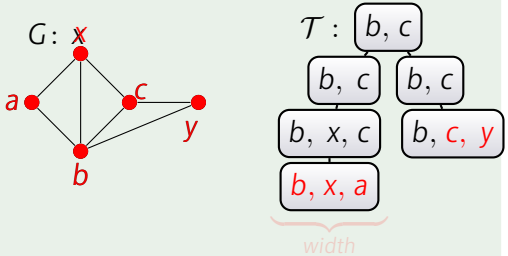
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



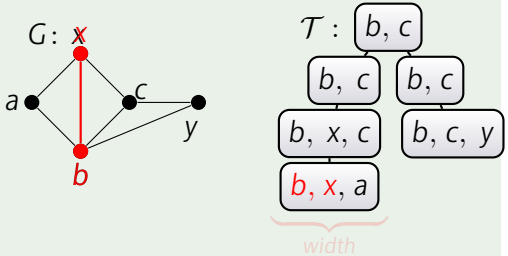
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



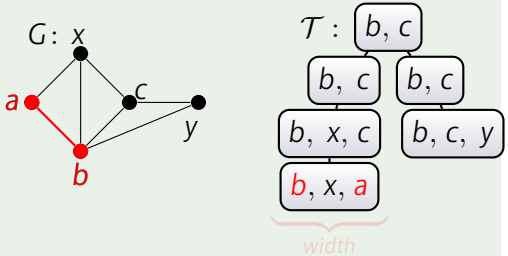
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



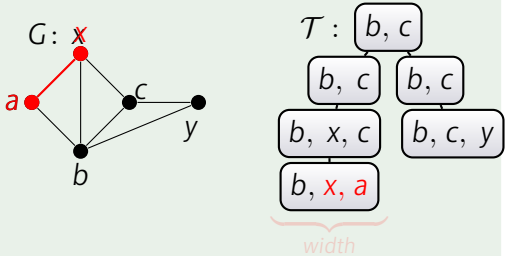
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



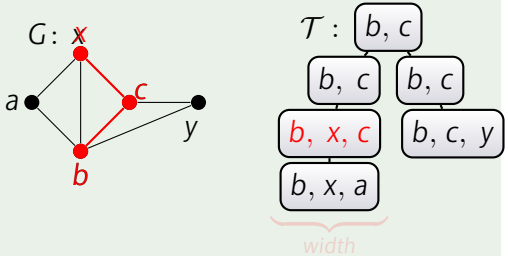
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



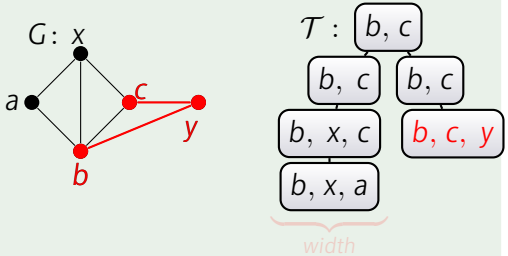
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



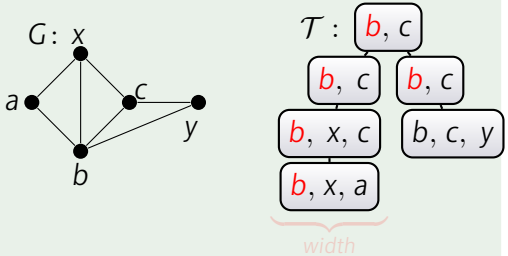
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



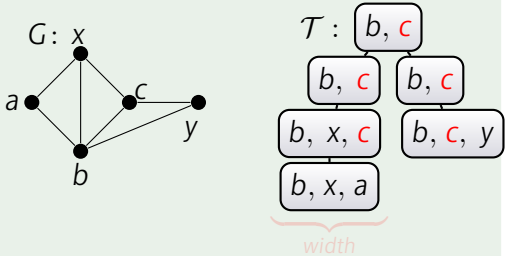
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



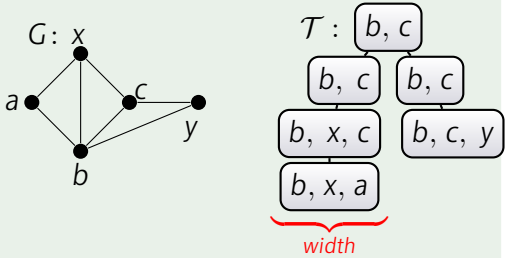
Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Tree Decompositions

Tree Decomposition $\mathcal{T}$ of G



Definition

A tree decomposition is a tree obtained from an arbitrary graph s.t.

1. Each vertex must occur in some bag
2. For each edge, there is a bag containing both endpoints
3. Connected: If vertex v appears in bags of nodes t_0 and t_1, then v is also in the bag of each node on the path between t_0 and t_1

Solving Problems on Tree Decompositions

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Seen the algorithm and just saying that we apply the algorithm in the same way as before is really hand waiving, right?

⇒ Usually, we make tree decompositions nice, then we define the algorithm on nice tree decompositions

Why: nice tree decompositions?

1. Keep the numbers of potential case distinctions very small and
2. it's again very close to a tree (so cases are simpler and in that case algorithms are very similar to the ones on trees)

However, nice tree decompositions requires us to copy more data in practice.
⇒ we often avoid nice tree decompositions in practice

Solving Problems on Tree Decompositions

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Seen the algorithm and just saying that we apply the algorithm in the same way as before is really hand waiving, right?

$\Rightarrow$ Usually, we make tree decompositions nice, then we define the algorithm on nice tree decompositions

Why: nice tree decompositions?

1. Keep the numbers of potential case distinctions very small and
2. it's again very close to a tree (so cases are simpler and in that case algorithms are very similar to the ones on trees)

However, nice tree decompositions requires us to copy more data in practice.
$\Rightarrow$ we often avoid nice tree decompositions in practice

Solving Problems on Tree Decompositions

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Seen the algorithm and just saying that we apply the algorithm in the same way as before is really hand waiving, right?

$\Rightarrow$ Usually, we make tree decompositions nice, then we define the algorithm on nice tree decompositions

Why: nice tree decompositions?

1. Keep the numbers of potential case distinctions very small and
2. it's again very close to a tree (so cases are simpler and in that case algorithms are very similar to the ones on trees)

However, nice tree decompositions requires us to copy more data in practice.
$\Rightarrow$ we often avoid nice tree decompositions in practice

Solving Problems on Tree Decompositions

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Seen the algorithm and just saying that we apply the algorithm in the same way as before is really hand waiving, right?

$\Rightarrow$ Usually, we make tree decompositions nice, then we define the algorithm on nice tree decompositions

Why: nice tree decompositions?

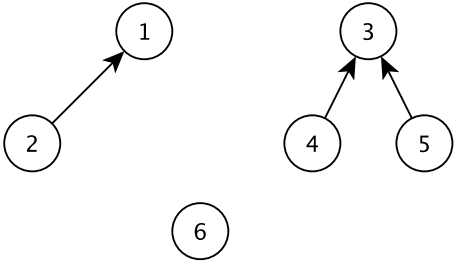
1. Keep the numbers of potential case distinctions very small and
2. it's again very close to a tree (so cases are simpler and in that case algorithms are very similar to the ones on trees)

However, nice tree decompositions requires us to copy more data in practice.
$\Rightarrow$ we often avoid nice tree decompositions in practice

Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

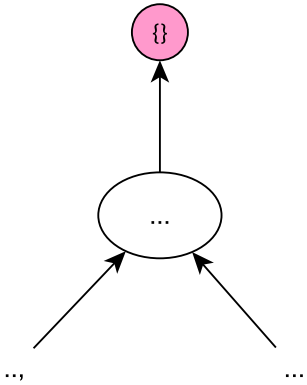
0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

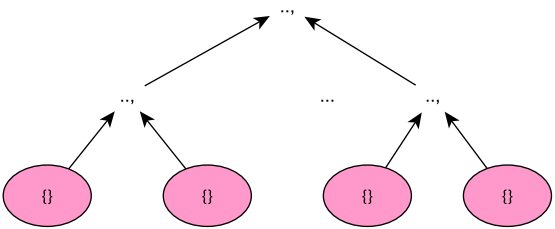
0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

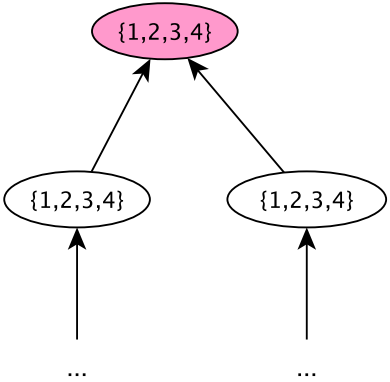
0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$) (0 children)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

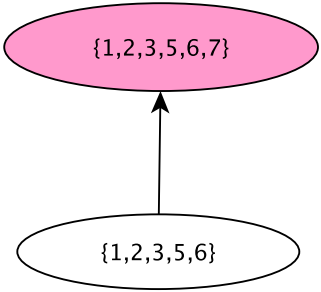
0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$) (2 children)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

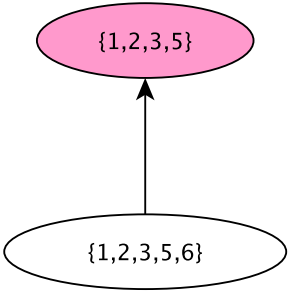
0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$) (1 child)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$) (1 child)



Nice Tree Decompositions

Cases (we restrict the content of the bags B_t at node t)

0. A node has 0, 1, or 2 children
1. Root is empty ($B_t = \{\}$)
2. Leafs are empty ($B_t = \{\}$)
3. Join nodes have the same content (children t_1,t_2, then $B_t = B_{t_1} = B_{t_2}$)
4. Introduce (child t_1, then $B_t = B_{t_1} \cup \{a\}$ for $a \in V$)
5. Forget (child t_1, then $B_t = B_{t_1} \setminus \{a\}$ for $a \in V$)

We can transform a tree decomposition into a nice tree decomposition in polynomial time.

Solving Problems on Tree Decompositions (cont.)

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Runtime

Gives a runtime result of $\mathcal{O}(2^w \cdot \text{poly}(n))$ for a given tree decomposition of width w and a number n of vertices

Other Problems

There are numerous problems that can be solved in runtime $\mathcal{O}(2^w \cdot \text{poly}(n))$ polynomial time if we can find a tree decomposition of width w

Solving Problems on Tree Decompositions (cont.)

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Runtime

Gives a runtime result of $\mathcal{O}(2^w \cdot \text{poly}(n))$ for a given tree decomposition of width w and a number n of vertices

Other Problems

There are numerous problems that can be solved in runtime $\mathcal{O}(2^w \cdot \text{poly}(n))$ polynomial time if we can find a tree decomposition of width w

Solving Problems on Tree Decompositions (cont.)

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Runtime

Gives a runtime result of $\mathcal{O}(2^w \cdot \text{poly}(n))$ for a given tree decomposition of width w and a number n of vertices

Other Problems

There are numerous problems that can be solved in runtime $\mathcal{O}(2^w \cdot \text{poly}(n))$ polynomial time if we can find a tree decomposition of width w

Solving Problems on Tree Decompositions (cont.)

Vertex Cover

Apply the algorithm in the same way as before. Construct the tables at each bag.

Runtime

Gives a runtime result of $\mathcal{O}(2^w \cdot \text{poly}(n))$ for a given tree decomposition of width w and a number n of vertices

Other Problems

There are numerous problems that can be solved in runtime $\mathcal{O}(2^w \cdot \text{poly}(n))$ polynomial time if we can find a tree decomposition of width w

How to find tree decompositions?

We briefly discuss it in one of the later lectures

(covering this topic in more details could easily fill an entire course)

General Techniques

- Find tree decompositions an minimize width
  (exact, best result, slow; NP-c)
- Find tree decompositions an try to minimize width
  (heuristic, fair result without guarantee, fast; polynomial)

Treewidth

Having the notion of a tree decomposition at hand.

- Observe that there are multiple tree decompositions possible

$\Rightarrow$ Define Treewidth

- Consider a tree decomposition of smallest width (minimize width)
- Treewidth of a graph is the width of a tree decomposition of smallest width

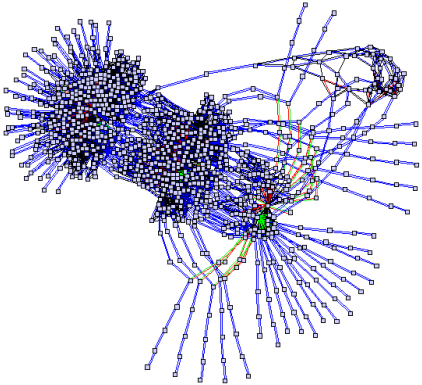
Solving Problems on Tree Decomposition (cont.)

So far

1. Notion treewidth at hand
2. Assuming that we can compute the treewidth of a graph

$\Rightarrow$ We can compute a tree decomposition of treewidth

3. Many algorithms run in polynomial-time for fixed width (Hence: efficient)
4. Treewidth gives us a structure to solve problems faster (if the structure is present)



Solving Problems of Bounded Treewidth

Courcelle's (Meta-)Theorem

Problems that can be expressed as a sentence φ in monadic second order logic can be solved in time $f(w) \cdot \text{poly}(n + m)$ where w is the width, $n := |V|$, and $m := |V|$ (MSO model checking, i.e., Does $G \models \varphi$ hold?)[a]

[a] Elberfeld et al.'11 improved it to linear

Monadic Second Order (MSO)

First order logic + Quantification over Sets of Vertices,
for example,
3-Col $\exists R, \exists G, \exists B : \forall x, y : Exy \rightarrow \neg(Rx \wedge Ry) \wedge \neg(Gx \wedge Gy) \wedge (Bx \wedge By)$;
Vertex Cover $\exists S : \forall x, y : Exy \rightarrow (Sx \vee Sy)$

Introduction

From Classical to Parameterized Complexity

Solving Problems on Tree Decomposition (cont.)

Question

Didn't we say that classical complexity rules out efficient solutions?

Ideas?

1. Polynomial for fixed width (BUT: we need small treewidth)
2. Classical complexity talks about negative results in the worst case for all possible inputs

Solving Problems on Tree Decomposition (cont.)

Question

Didn't we say that classical complexity rules out efficient solutions?

Ideas?

1. Polynomial for fixed width (BUT: we need small treewidth)
2. Classical complexity talks about negative results in the worst case for all possible inputs

From Classical to Parameterized Complexity

Drawbacks (classical complexity)

- But, efficient algorithms for real-world instances still possible

$\Rightarrow$ Insufficient as a theoretical framework to rule out that solutions can be found efficiently if we also talk about structure

- Reason: only size of the input considered (1-D)

From Classical to Parameterized Complexity

Drawbacks (classical complexity)

- But, efficient algorithms for real-world instances still possible

$\Rightarrow$ Insufficient as a theoretical framework to rule out that solutions can be found efficiently if we also talk about structure

- Reason: only size of the input considered (1-D)

Business as usual

- Whenever the approach does not fit, add another dimension
- 2-D fix the idea of an additional "hidden structure" (parameter)
- Exploit to solve problem faster

Classical/Parameterized Complexity

What do we measure?

- Runtime
- Space

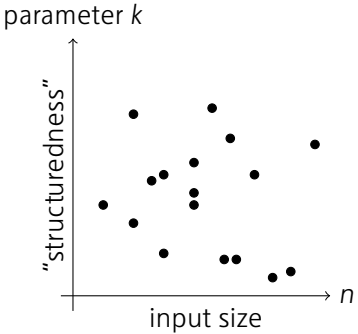
depending on the size of the input (actual length of the input instance).

What cases are interesting?

- Behavior on any input instance (hence: arbitrarily complicated)
- Best/worst case
  Why not average case? well...
  The worst case is already complicated enough (for now)
  Anyhow what's the average input?

How to capture hidden structures in theory?

2D Theoretical Model



Parameter

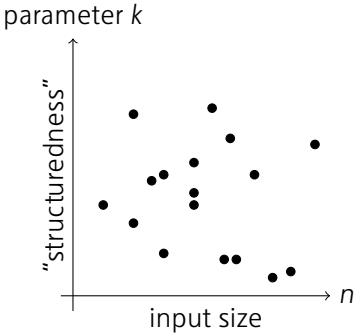
Can represent various structural aspects of input

Question

Suitable mathematical correlation between n and k?

How to capture hidden structures in theory?

2D Theoretical Model



Mathematical Notion

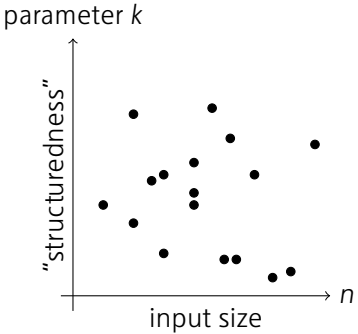
Parameterized Complexity Theory
[Downey and Fellows (99, 14)]

Key Concepts

- Control the explosion in the search space
- Confine the combinatorial explosion which is expected in the parameter k

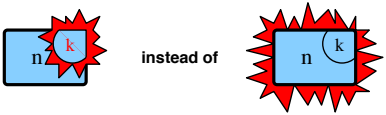
How to capture hidden structures in theory?

2D Theoretical Model



Key Concepts

- Control the explosion in the search space
- Confine the combinatorial explosion which is expected in the parameter k



Comment

There is not THE parameter.

Parameters to Parameterize in

- Treewidth
- Number of Vertices?
- Size of the Solution (e.g., number of vertices in the vertex cover)
- ...

It all depends on the parameter we choose.

In the following, k is the value the parameter may have.

Notions of Parameterized Complexity

Fixed-parameter tractability

- Parameterized counterpart of polynomial-time tractability
- Combinatorial explosion confined to the parameter k
- The running time is polynomial in n and
- The order of the polynomial is independent of k
- Runtime: $f(k) \cdot n^{\mathcal{O}(1)}$

Fpt reduction

- Parameterized counterpart of polynomial-time reduction
- Many-to-one reduction
- Runtime: $f(k) \cdot n^{\mathcal{O}(1)}$

Parameterized Reductions

Essentially, very similar to before we only add an integer (the parameter) that has to survive the reduction

Fpt reduction from L_1 to L_2

- finite alphabets Σ_1 and Σ_2
- $L_1 \subseteq \Sigma_1^* \times \mathbb{N}$, $L_2 \subseteq \Sigma_2^* \times \mathbb{N}$ parameterized decision problems
- many-to-one reduction from $\Sigma_1^* \times \mathbb{N}$ to $\Sigma_2^* \times \mathbb{N}$ s.t.
  - for all $I_1 \in \Sigma_1^*$ we have $(I_1, k_1) \in L_1$ if and only if $r(I_1, k_1) = (I_2, k_2) \in L_2$
  - $k_2 \leq g(k_1)$ for a fixed computable function $g : \mathbb{N} \to \mathbb{N}$
  - t.i. a computable function f and a constant c s.t. r is computable in time $\mathcal{O}(f(k_1)\|I_1\|^c)$

Introduction

Basic Classes

Basic Classes of Parameterized Problems

- FPT: the class of all parameterized problems that are fixed-parameter tractable
- XP: all parameterized problems that can be solved in polynomial time if the parameter is considered constant, that is, $(x, k) \in P$ can be decided in time $\mathcal{O}(|x|^{f(k)})$
- para-NP: all parameterized problems that are in NP if the parameter is considered as a constant, that is, $(x, k) \in P$ can be decided nondeterministically in time $\mathcal{O}(|x|^{f(k)})$

Discussion

- If f is a polynomial, then A runs in polynomial time.
- Hence, if the unparameterized version of P is NP-hard, then we will assume that f is super-polynomial.
- FPT $\subseteq$ XP (can be shown to be a proper subset)
- FPT $\subseteq$ para-NP
- XP and para-NP are incomparable (subject to complexity theoretic assumptions)
- Between FPT and XP $\cap$ para-NP are the W[i]-classes which we will discuss separately.

Notations (just that you heard about it)

Big $\mathcal{O}$-Star

- Denote the running time of an fpt algorithm as $\mathcal{O}^*(f(k))$
- Simply omit the polynomial factor poly(n)

Alternative style (Flum/Grohe)

- A parameterized decision problem P is a pair (L, κ) where $P \subseteq \Sigma^*$ and κ is a mapping from Σ^* to $\mathbb{N}^+$.
- Both definitions are basically equivalent.
- Informally we don't make a difference.

Additive vs. Multiplicative

$f(k)n^c$ vs. $f(k) + n^c$

In the definition of FPT we could have used the additive term. Let P be a parameterized problem and (x, k) and instance, $n = |x|$.

Idea.

- ($\Longrightarrow$) Assume there exists an algorithm A that solves the problem in time $f(k)n^c$. By the inequality $ab \leq a^2 + b^2$, this algorithm solves the problem in time $f^2(k) + n^{2c}$.
- ($\Longleftarrow$) Assume the problem has an algorithm B that solves the problem in time $f(k) + n^c$. By the inequality $a + b \leq a(b + 1)$ we have $f(k) + n^c \leq f(k)(n^c + 1) < f(k)n^{c+1}$.

$\square$

Introduction

Weft Hierarchy and Hardness

W-Hierarchy

- Consider a decision circuit consisting of large gates with unbounded fan-in, and small gates with bounded fan-in (the precise value of the bound is unimportant).
- The weft of the circuit is the largest number of large gates on any path from the inputs to the output of the circuit.
- The depth of the circuit is the largest number of gates of any size on any path from the inputs to the output of the circuit.

Weighted Weft t Depth h Circuit Satisfiability ($WCS(t, h)$):

- Instance: a weft t depth h circuit C and an integer k
- Parameter: k
- Question: has C a satisfying input of weight exactly k?

Weft Hierarchy (cont.)

Weft Hierarchy

- For each $t \geq 1$, the class W[t] is defined as the class of all parameterized problems that can be reduced to $WCS(t, h)$ for some constant h
- W[SAT] is the class of parameterized problems that can be reduced to $WCS(\infty, \infty)$, where all gates have fan-out 1
- W[P] is the class of parameterized problems that can be reduced to $WCS(\infty, \infty)$

$$\text{FPT} \subseteq \text{W[1]} \subseteq \text{W[2]} \subseteq \cdots \subseteq \text{W[SAT]} \subseteq \text{W[P]}$$

Logical Characterization of the W-Hierarchy

First-Order Model Checking $MC(\Phi)$

- Given: structure $\mathcal{S}$, formula $\varphi \in \Phi$
- Parameter: $|\varphi|$
- Question: Decide $\mathcal{S} \models \varphi$.
- $\Sigma_{t,u}$ = class of all FO formulas with
  - t quantifier blocks alternations, starting with $\exists$,
  - all blocks after the leading one consist of at most u quantifiers.
- Example: $\exists x_1, \ldots x_n \forall y_1, y_2 \exists z_1, z_2, z_3, z_4 \varphi$ belongs to $\Sigma_{3,4}$
- Theorem $MC(\Sigma_{t,u})$ is W[t]-complete, for every $u, t \geq 1$

Examples

Clique

$$\exists x_1, \ldots, x_k \bigwedge_{1 \leq i < j \leq k} E(x_i, x_j)$$

belongs to $\Sigma_{1,0}$ hence Clique belongs to W[1]

Dominating Set

$$\exists x_1, \ldots, x_k \forall y \Big(\bigvee_{i=1}^{k} y = x_i \vee E(y, x_i) \Big)$$

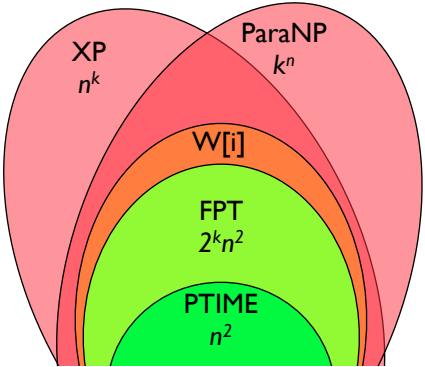
belongs to $\Sigma_{2,1}$ hence Dominating Set belongs to W[2]

Introduction

Brief Summary

Parameterized Complexity Zoo

Input x: $|x| = n$ Parameter: k



- FPT (parameterized P)

  All non-polynomial parts depend only on the parameter and not on the input: $f(k) \cdot n^{\mathcal{O}(1)}$
- Werft Hierarchy (W[P]): parameterized P + at most $h(k)\log(n)$ non-deterministic steps

  Decid in $f(k) \cdot n^{\mathcal{O}(1)}$ deterministic steps and at most $h(k)\log(n)$ non-deterministic steps
- XP (nonuniform parameterized P)

  Order of the polynomial depends on the parameter: $\mathcal{O}(n^{f(k)})$
- para-NP (parameterized NP)

  non-deterministic algorithm; All non-polynomial parts depend only on the parameter and not on the input: $f(k) \cdot n^{\mathcal{O}(1)}$

Summary

- Parameterized Complexity provides a framework for more detailed analysis
- In theory, differentiate between:
  - Easy (PTIME, FPT)
    Ex: Vertex Cover (treewidth, also solution size)
  - Hard (W[i])
    Ex: Clique, Dominating Set
  - Very Hard (XP)
  - Intractable (ParaNP)
    Ex: many problems in AI (will see some of them later)
- In practice: FPT is not enough (we need a small tower for the f)

Introduction

Propositional Satisfiability

Boolean Satisfiability Problem

SAT-Problem

Given:	Propositional formula F.
Question:	Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Input normal form

- Conjunctive normal form (CNF)
- Form:

  $F = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge \ldots \wedge (\ldots)$

  where ℓ_i either x or $\neg x$

Boolean Satisfiability Problem

SAT-Problem

Given:	Propositional formula F.
Question:	Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Input normal form

- Conjunctive normal form (CNF)
- Form:

  $F = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge \ldots \wedge (\ldots)$

  where ℓ_i either x or $\neg x$

Boolean Satisfiability Problem

SAT-Problem

Given:	Propositional formula F.
Question:	Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$(a \vee \neg b \vee d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge$$

$$(\neg a \vee \neg d) \wedge (b \vee c) \wedge (d \vee e) \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given:	Propositional formula F.
Question:	Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$(a \vee \neg b \vee d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge$$

$$(\neg a \vee \neg d) \wedge (b \vee c) \wedge (d \vee e) \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge (\neg a \vee \neg c \vee \neg d) \wedge$$

$$(\neg a \vee \neg d) \wedge (b \vee c) \wedge (d \vee e) \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \lor \neg b \lor d)} \land \cancelto{1}{(\neg a \lor \neg c \lor \neg d)} \land$$

$$(\neg a \lor \neg d) \land (b \lor c) \land (d \lor e) \land (\neg b \lor \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge \cancelto{1}{(\neg a \vee \neg c \vee \neg d)} \wedge$$

$$\cancelto{1}{(\neg a \vee \neg d)} \wedge (b \vee c) \wedge (d \vee e) \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge \cancelto{1}{(\neg a \vee \neg c \vee \neg d)} \wedge$$

$$\cancelto{1}{(\neg a \vee \neg d)} \wedge \cancelto{1}{(b \vee c)} \wedge (d \vee e) \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given:	Propositional formula F.
Question:	Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge \cancelto{1}{(\neg a \vee \neg c \vee \neg d)} \wedge$$

$$\cancelto{1}{(\neg a \vee \neg d)} \wedge \cancelto{1}{(b \vee c)} \wedge \cancelto{1}{(d \vee e)} \wedge (\neg b \vee \neg e)$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge \cancelto{1}{(\neg a \vee \neg c \vee \neg d)} \wedge$$

$$\cancelto{1}{(\neg a \vee \neg d)} \wedge \cancelto{1}{(b \vee c)} \wedge \cancelto{1}{(d \vee e)} \wedge \cancelto{1}{(\neg b \vee \neg e)}$$

Boolean Satisfiability Problem

SAT-Problem

Given: Propositional formula F.

Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Example

$$\cancelto{1}{(a \vee \neg b \vee d)} \wedge \cancelto{1}{(\neg a \vee \neg c \vee \neg d)} \wedge$$

$$\cancelto{1}{(\neg a \vee \neg d)} \wedge \cancelto{1}{(b \vee c)} \wedge \cancelto{1}{(d \vee e)} \wedge \cancelto{1}{(\neg b \vee \neg e)}$$

$\Rightarrow$ Satisfiable

Some Examples for Parameterizations in SAT

1. SAT(vars)
   - Instance: a CNF formula F.
   - Parameter: number of variables of F.
   - Question: is F satisfiable?
2. SAT(clause size)
   - Instance: a CNF formula F.
   - Parameter: the size of a largest clause of F.
   - Question: is F satisfiable?
3. Bounded-SAT(ones)
   - Instance: a CNF formula F and an integer k.
   - Parameter: k.
   - Question: is there a satisfying assignment for F that sets at mosty k variables to 1?

Can we locate the problems in FPT, XP, or para-NP?

1. FPT

   Because: well just branch on every variable
2. para-NP-complete (unlikely in FPT)

   Because: NP-c already for 3-CNF
3. XP (unlikely in FPT)

   Was definition of $W[i]$ hierarchy

Some Examples for Parameterizations in SAT

1. SAT(vars)
   - Instance: a CNF formula F.
   - Parameter: number of variables of F.
   - Question: is F satisfiable?
2. SAT(clause size)
   - Instance: a CNF formula F.
   - Parameter: the size of a largest clause of F.
   - Question: is F satisfiable?
3. Bounded-SAT(ones)
   - Instance: a CNF formula F and an integer k.
   - Parameter: k.
   - Question: is there a satisfying assignment for F that sets at mosty k variables to 1?

Can we locate the problems in FPT, XP, or para-NP?

1. FPT

   Because: well just branch on every variable
2. para-NP-complete (unlikely in FPT)

   Because: NP-c already for 3-CNF
3. XP (unlikely in FPT)

   Was definition of $W[i]$ hierarchy

Some Examples for Parameterizations in SAT

1. SAT(vars)
   - Instance: a CNF formula F.
   - Parameter: number of variables of F.
   - Question: is F satisfiable?
2. SAT(clause size)
   - Instance: a CNF formula F.
   - Parameter: the size of a largest clause of F.
   - Question: is F satisfiable?
3. Bounded-SAT(ones)
   - Instance: a CNF formula F and an integer k.
   - Parameter: k.
   - Question: is there a satisfying assignment for F that sets at mosty k variables to 1?

Can we locate the problems in FPT, XP, or para-NP?

1. FPT

   Because: well just branch on every variable
2. para-NP-complete (unlikely in FPT)

   Because: NP-c already for 3-CNF
3. XP (unlikely in FPT)

   Was definition of $W[i]$ hierarchy

Some Examples for Parameterizations in SAT

1. SAT(vars)
   - Instance: a CNF formula F.
   - Parameter: number of variables of F.
   - Question: is F satisfiable?
2. SAT(clause size)
   - Instance: a CNF formula F.
   - Parameter: the size of a largest clause of F.
   - Question: is F satisfiable?
3. Bounded-SAT(ones)
   - Instance: a CNF formula F and an integer k.
   - Parameter: k.
   - Question: is there a satisfying assignment for F that sets at mosty k variables to 1?

Can we locate the problems in FPT, XP, or para-NP?

1. FPT

   Because: well just branch on every variable
2. para-NP-complete (unlikely in FPT)

   Because: NP-c already for 3-CNF
3. XP (unlikely in FPT)

   Was definition of $W[i]$ hierarchy

SAT and Treewidth

More about this soon...
when we talk about counting

Computing Tree Decompositions

Purpose of this Chapter

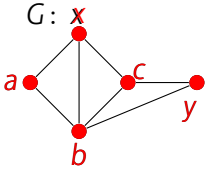
1. Get familiar with tree decompositions
2. Introduce techniques to compute TDs (SAT encoding, Heuristic, Hybrid)
3. Get the chance to try out tools

Recall Treewidth

Treewidth

- Most prominent graph invariant
- Small treewidth indicates tree-likeness and sparsity
- Treewidth defined in terms of tree decompositions (TD)
- TD: arrangement of graph into a tree + bags s.t. ...
- Treewidth: TD of smallest width

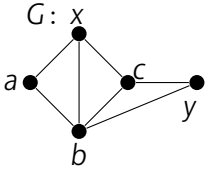
Recall Treewidth

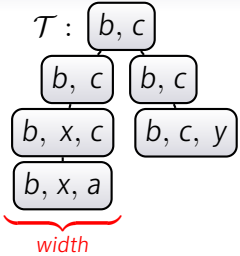


Treewidth

- Most prominent graph invariant
- Small treewidth indicates tree-likeness and sparsity
- Treewidth defined in terms of tree decompositions (TD)
- TD: arrangement of graph into a tree + bags s.t. ...
- Treewidth: TD of smallest width

Recall Treewidth





Treewidth

- Most prominent graph invariant
- Small treewidth indicates tree-likeness and sparsity
- Treewidth defined in terms of tree decompositions (TD)
- TD: arrangement of graph into a tree + bags s.t. ...
- Treewidth: TD of smallest width

Motivation

Exploit TDs of Small Width

- Many combinatorial hard problems can be solved in polynomial-time for graphs of small bounded treewidth
- Applies to variety of fields
  (discrete algorithms, CSP, KR, biology, probabilistic networks)
- Why small width?
  Runtime of DP algorithms usually exponential in width

$\Rightarrow$ "small reduction in width (say by even by 1 or 2) can amount to one or two orders of magnitude reduction in inference time" (Kask et al.'15)

Computing Tree Decompositions

Intuition Tree Decompositions: Cops and Robber Games

The following slides (Cops and Robber Games) are from
Sebastian Ordyniak (Univ. Sheffield, UK)
on “Structural Decompositions and Algorithms”

Computing Tree Decompositions

Computation: Overview (other than C&R)

How to Compute Treewidth?

A) Exact

Find tree decomposition of smallest width

1. NP-c task
2. limited to rather small graphs
3. slow

   SAT and combinatorial methods and implementations available;

   domain-specific preprocessing very important

B) Heuristics

Find some tree decomposition (preferably of small width)

1. decompositions often far from optimal
2. quite fast, scale up to large graphs

C) Hybrid

Find tree decompositions of (1) small width (2) fast

Exact (A)

Finding Tree Decompositions of Smallest Width

1. Hisao Tamaki (Combinatorial Decomposer)
2. Marko Samer & Helmut Veith (SAT-Encoding based)

Following, we talk only about (2), since (1) is quite involved [see Hisao's slides]

Exact (A)

Finding Tree Decompositions of Smallest Width

1. Hisao Tamaki (Combinatorial Decomposer)
2. Marko Samer & Helmut Veith (SAT-Encoding based)

Following, we talk only about (2), since (1) is quite involved [see Hisao's slides]

Computing Tree Decompositions

A SAT-Encoding

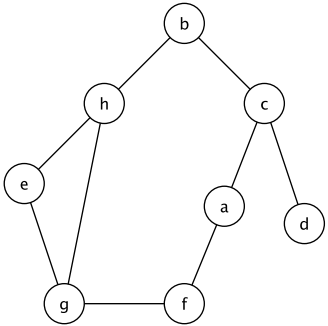
Treewidth in Terms of Elimination Orderings

Characterization [Bodlaender'05, Dechter'06]

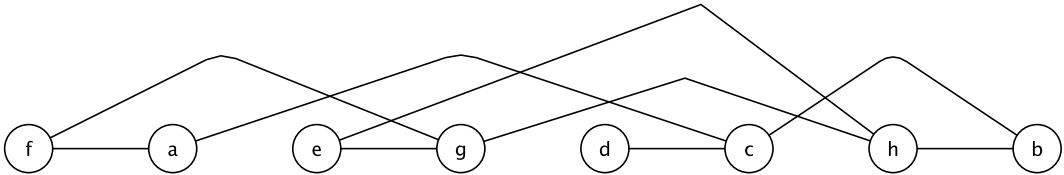
1. For a graph G: there is a linear ordering of its vertices from which one can construct a TD whose width equals the $\mathrm{tw}(G)$.
2. Linear ordering gives rise to a triangulation; from which width can be directly read off and from which the TD can be constructed
3. Then sometimes also called induced treewidth

Treewidth in Terms of Elimination Orderings (cont.)

- Given: Graph $G = (V, E)$
  Linear ordering $v_1, v_2, \ldots, v_n$ of the vertice
- v_j predecessor of v_i if $\{v_i, v_j\} \in E$ and $j < i$
- v_j successor of v_i if $\{v_i, v_j\} \in E$ and $j > i$

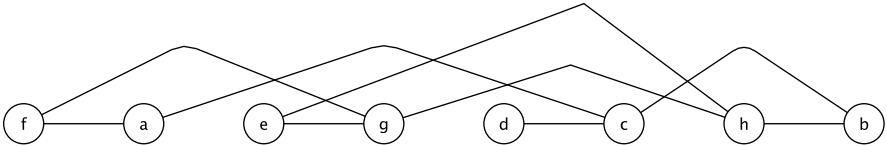


$f < a < e < g < d < c < h < b$

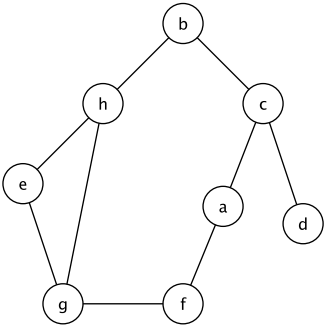


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

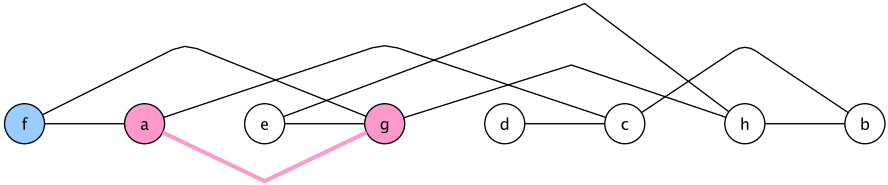


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

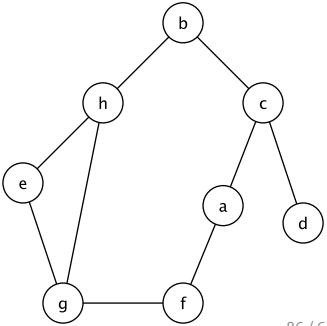


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

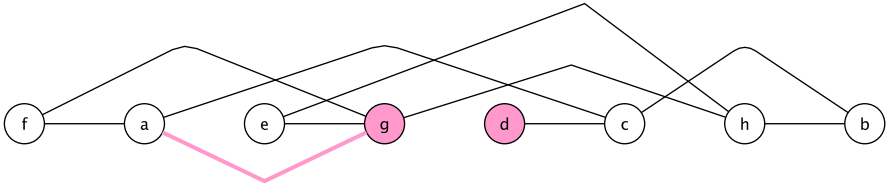


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

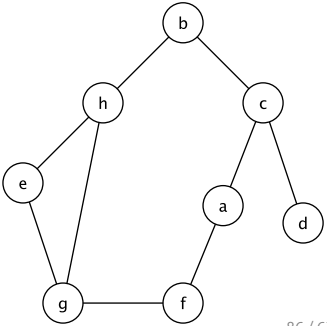


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

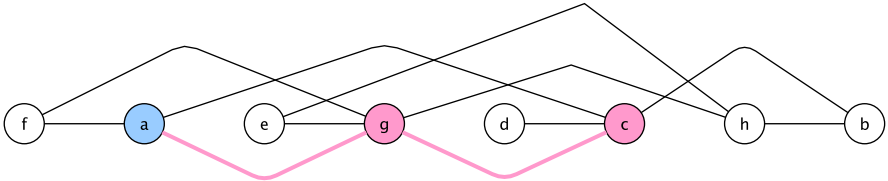


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

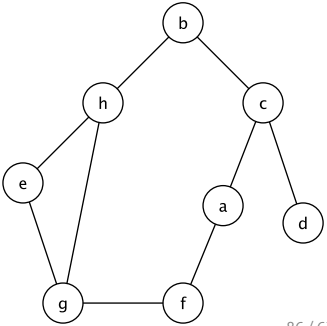


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

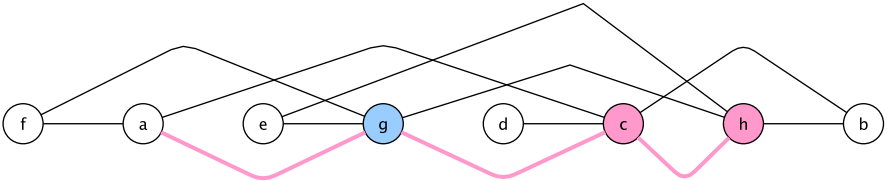


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

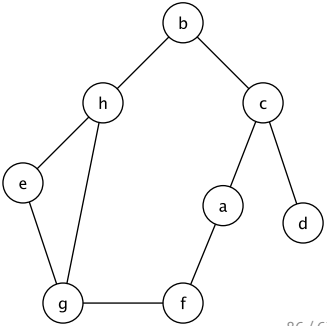


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

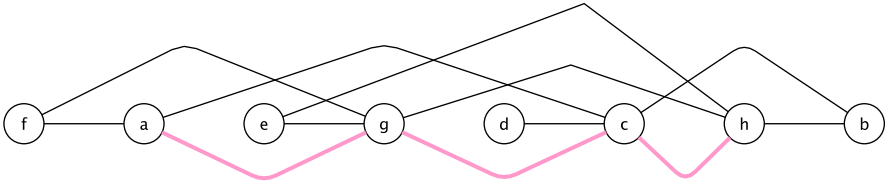


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

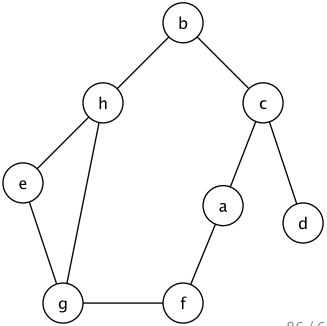


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

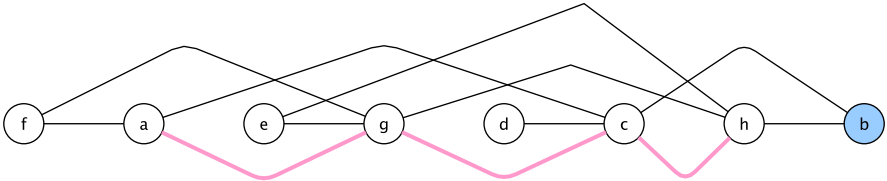


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.

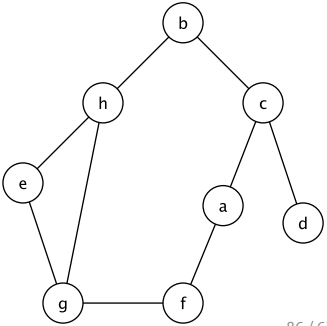


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

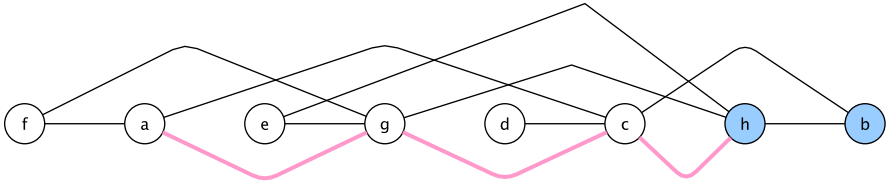


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex b has 0 successors

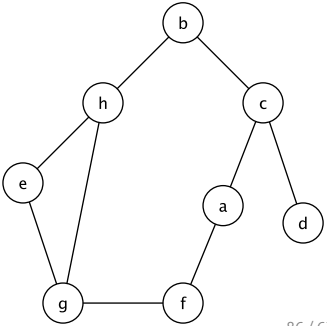


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

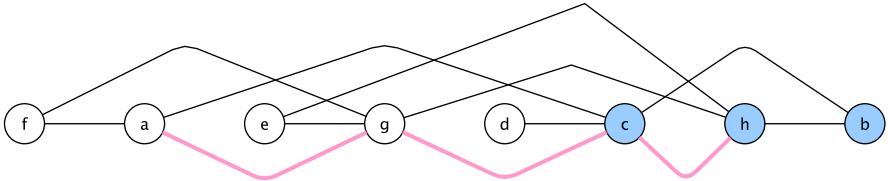


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex h has 1 successor

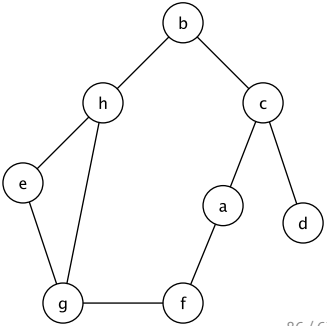


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

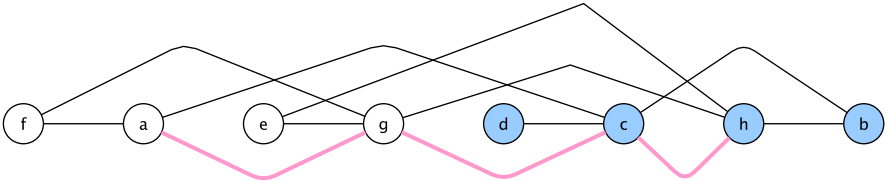


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex c has 2 successors

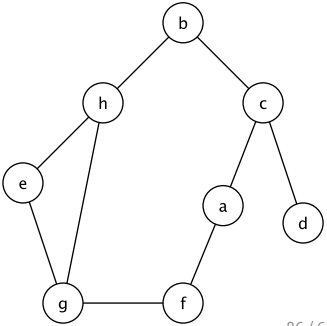


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

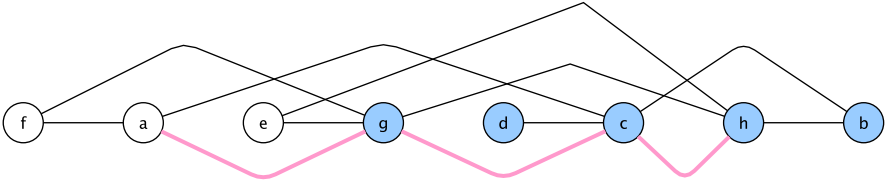


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex d has 1 successor

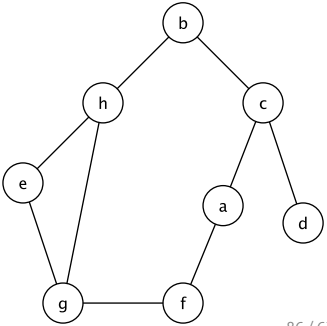


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

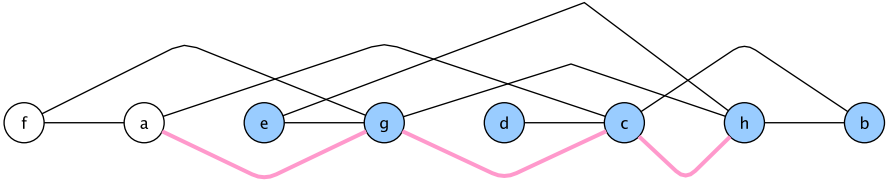


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex g has 2 successors

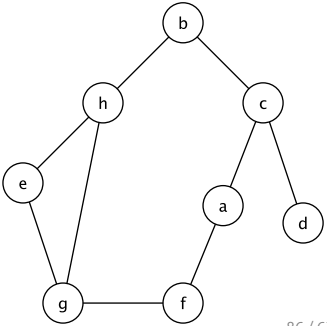


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

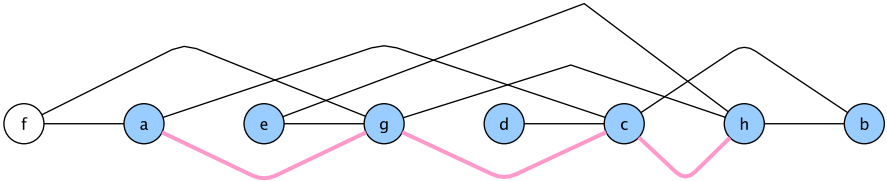


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex e has 2 successors

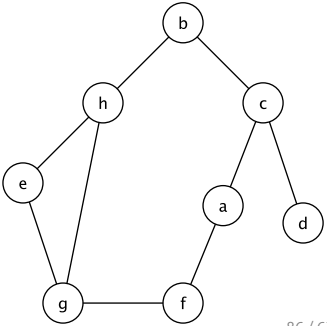


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

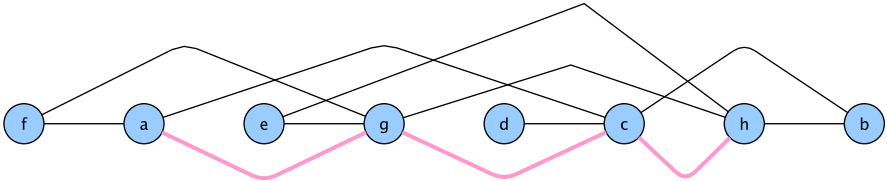


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex a has 2 successors

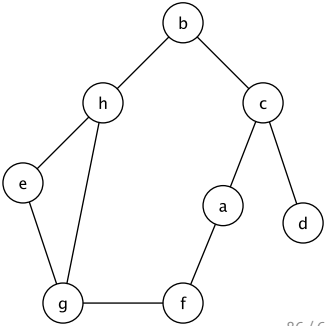


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

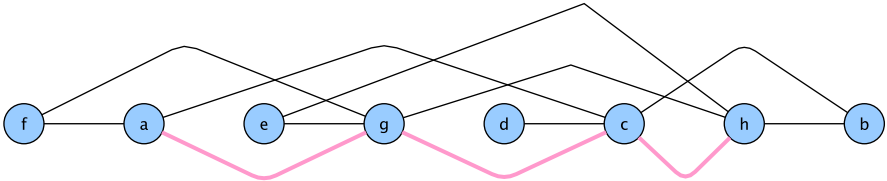


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Vertex f has 2 successors

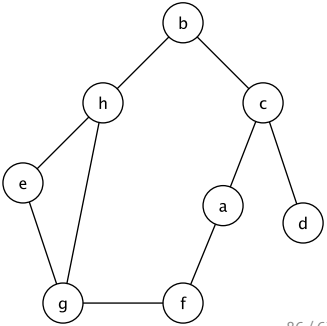


Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$

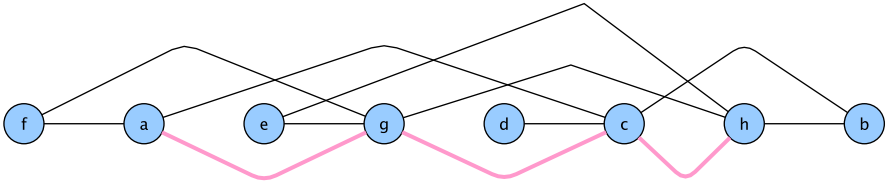


1. For each pair of non-neighbors v_i and v_j that have a common predecessor, add an edge $\{v_i, v_j\}$ to E
2. Repeat (1) as long as new edges can be added
3. The width is the maximum number of successors overall vertices, i.e., $\max_{v_i \in V} |\{v_i v_j \in E : j > i\}|$.
   Maximum is 2
   $\Rightarrow$ treewidth is 2

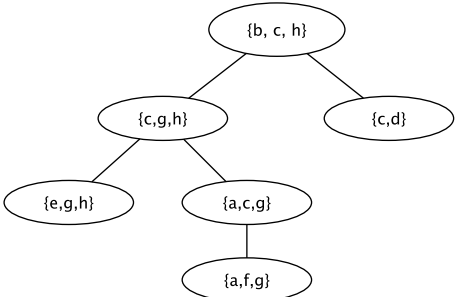


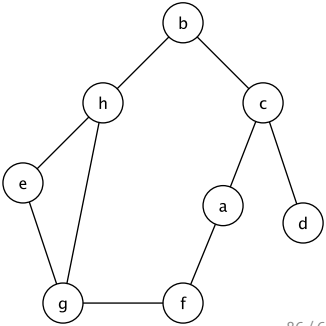
Treewidth in Terms of Elimination Orderings (cont.)

$f < a < e < g < d < c < h < b$



Tree decomposition:





From Elimination Orderings to a SAT Encoding

- $G = (V, E)$; $n = |V|$, $m = |E|$
- For simplicity $V = \{1, 2 \ldots, n\}$
- Linear Ordering
  - $\text{ord}_{i,j}$:
    - $\text{ord}_{i,j}$ true if and only if vertex $v_i < v_j$
    - $n \cdot (n-1)/2$ variables with $1 \leq i < j \leq n$
  - Clauses to enforce transitivity

    $$(\text{ord}^*_{i,j} \wedge \text{ord}^*_{j,l}) \rightarrow \text{ord}^*_{i,l} \text{ where } \text{ord}^*_{p,q} = \begin{cases} \text{ord}_{p,q} & \text{if } p < q \\ \neg ord_{q,p} & \text{otherwise} \end{cases}$$

    - $n \cdot (n-1) \cdot (n-2)$

From Elimination Orderings to a SAT Encoding

- $G = (V, E)$; $n = |V|$, $m = |E|$
- For simplicity $V = \{1, 2 \ldots, n\}$
- Successor relation induced by the ordering
  - Linear ordering gives a direction to all edges $\{v_i, v_j\} \in E$ (from v_i to v_j) iff $v_j > v_i$
  - $\text{arc}_{i,j}$ with $1 \leq i, j \leq n$
    - n^2 variables
    - $\text{arc}_{i,j}$ is true if vertex $v_j > v_i$
    - arc variable active if it is assigned true

From Elimination Orderings to a SAT Encoding

- $G = (V, E)$; $n = |V|$, $m = |E|$
- For simplicity $V = \{1, 2 \ldots, n\}$
- Successor relation induced by the ordering
  - Linear ordering gives a direction to all edges $\{v_i, v_j\} \in E$ (from v_i to v_j) iff $v_j > v_i$
  - $\text{arc}_{i,j}$ with $1 \leq i, j \leq n$
    - n^2 variables
    - $\text{arc}_{i,j}$ is true if vertex $v_j > v_i$
    - arc variable active if it is assigned true
- Edges of Input Graph
  - add $2 \cdot m$ clauses of the form
    $\text{ord}_{i,j} \rightarrow \text{arc}_{i,j}$ and $\neg\, \text{ord}_{i,j} \rightarrow \text{arc}_{j,i}$ for $\{v_i, v_j\} \in E$ and $i < j$

From Elimination Orderings to a SAT Encoding

- $G = (V, E)$; $n = |V|$, $m = |E|$
- For simplicity $V = \{1, 2 \ldots, n\}$
- Enforce Additional Edges (caused by linear ordering)
  - $(\text{arc}_{i,j} \wedge \text{arc}_{i,l} \wedge \text{ord}_{j,l}) \rightarrow \text{arc}_{j,l}$ and $(\text{arc}_{i,j} \wedge \text{arc}_{i,l} \wedge \neg\, \text{ord}_{j,l}) \rightarrow \text{arc}_{l,j}$ for $1 \leq i, j, l \leq n$, $i \neq j$, $i \neq l$, and $j < l$
  - $n \cdot (n-1) \cdot (n-2)$ clauses
- Redundant Clauses (speeds up the search; common trick)
  - $\neg\, \text{arc}_{i,j} \vee \neg\, \text{arc}_{i,l} \vee \text{arc}_{j,l} \vee \text{arc}_{l,j}$
  - $n \cdot (n-1) \cdot (n-2)/2$ clauses
- Unit Clauses (self loops can be neglected)
  - $\neg\, \text{arc}_{i,i}$ for $1 \leq i \leq n$
  - n unit clauses of the form
- At most k constraint for the outgoing edges
  - Guarantee that the instance is satisfiable if and only if there exists a tree decomposition of width at most k
  - Put a cardinality constraint that at most k outgoing arcs of each vertex are active

Heuristics (B)

Finding Some Tree Decomposition Preferably of Small Width

1. Ben Strasser (flow-cutter)
2. Hisao Tamaki (tcs-meiji)
3. Michael Abseher (htd: well-known bucket elimination, minfill)

htd: Bucket Elimination

Input: Graph (or hypergraph) + Elimination ordering $\vec{O}$
Result: A tree decomposition $\mathcal{T} = (T, \chi)$ of $G = (V, E)$

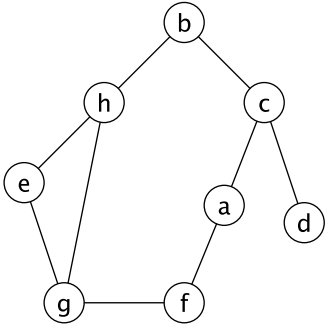
1. $E = \emptyset$
2. Create an empty (bucket) bag $\chi(v)$ for each vertex $v \in \vec{O}$
3. For $vw \in E$:
   Let $v < w$ by $\vec{O}$, then $\chi(t_v) := \chi(t_v) \cup \{w\}$
4. For $i \in \{1, \ldots, n\}$ and $\vec{O} = (v_1, v_2, \ldots, v_n)$ do
   /* Create temporary set A of vertices based on bag $\chi(t_{v_i})$ */
   (1) Let $A := \chi(t_{v_i}) \setminus \{v_i\}$
   /* Determine lowest ranked vertex in A based on $\vec{O}$ */
   (2) $v_j \in A$ s.t. $v_i < v_j$ and no $v_\ell \in A$ s.t. $v_i < v_\ell < v_j$
   (3) $\chi(B_{v_j}) := \chi(t_{v_j}) \cup A$
   (4) $E = E \cup \{(t_{v_i}, t_{v_j})\}$
5. Return $((N, E), \chi)$ where $N = \{t_1, \ldots, t_n\}$

How do we get the elimination ordering. For a heuristic just guess one.

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$

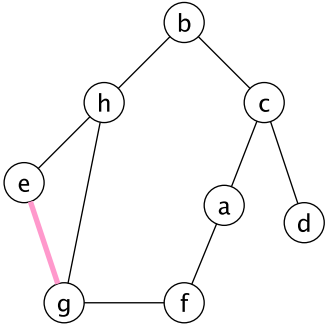


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $eg \in E$; $e < g$; $\chi(t_e) = \chi(t_e) \cup \{g\}$

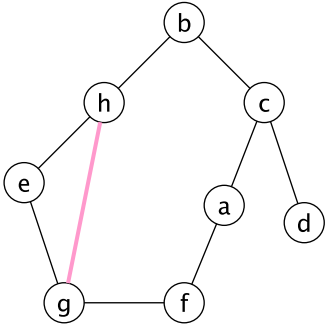


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $hg \in E$; $g < h$; $\chi(t_g) = \chi(t_g) \cup \{h\}$

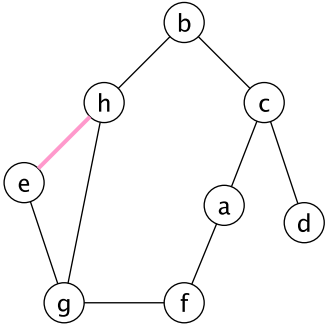


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $eh \in E$; $e < h$; $\chi(t_e) = \chi(t_e) \cup \{h\}$

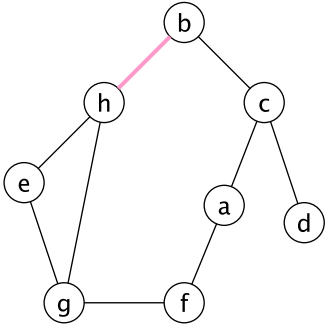


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $hb \in E$; $h < b$; $\chi(t_h) = \chi(t_h) \cup \{b\}$

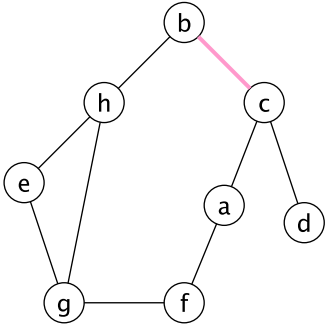


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $bc \in E$; $c < b$; $\chi(t_c) = \chi(t_c) \cup \{b\}$

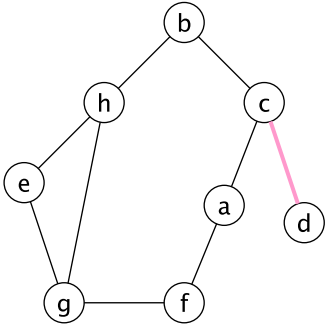


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $cd \in E$; $d < c$; $\chi(t_d) = \chi(t_d) \cup \{c\}$

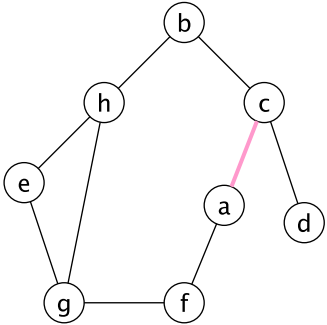


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $ca \in E$; $a < c$; $\chi(t_a) = \chi(t_a) \cup \{c\}$

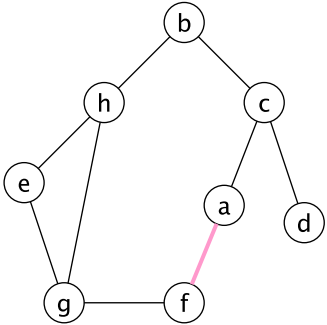


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $af \in E$; $f < a$; $\chi(t_f) = \chi(t_f) \cup \{a\}$

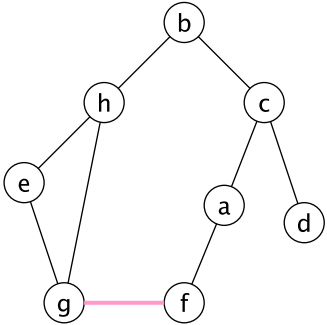


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $fg \in E$; $f < g$; $\chi(t_f) = \chi(t_f) \cup \{g\}$

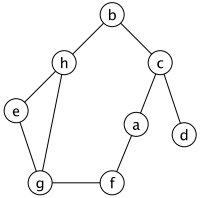


Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and
   $\chi(t_n) = \emptyset$ for $n \in N$
   $\chi(t_e) = \{g, h\}$, $\chi(t_g) = \{h\}$, $\chi(t_h) = \{b\}$,
   $\chi(t_c) = \{b\}$, $\chi(t_d) = \{c\}$, $\chi(t_a) = \{c\}$,
   $\chi(t_f) = \{a, g\}$, $\chi(t_b) = \{\}$



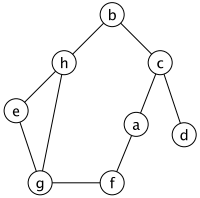
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



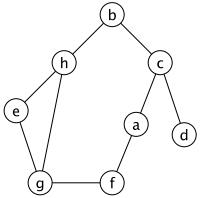
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



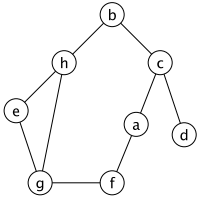
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



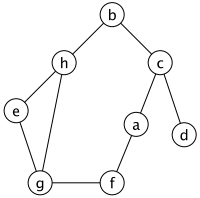
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



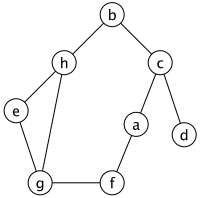
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



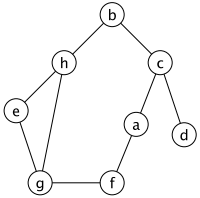
$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: Bucket Elimination Example

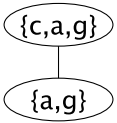
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 1$:
   (1) $A = \chi(t_{v_1}) \setminus \{v_1\} = \chi(t_f) \setminus \{f\} = \{a, g\}$
   (2) $v_j = a$
   (3) $\chi(t_a) = \chi(t_a) \cup A = \{c\} \cup \{a, g\}$
   (4) $E = E \cup \{(t_f, t_a)\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\}$

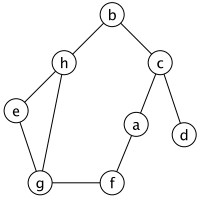


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

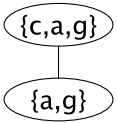
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\}$

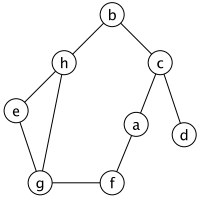


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

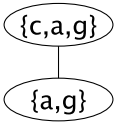
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\}$

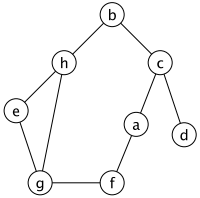


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

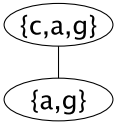
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\}$

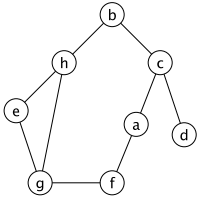


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

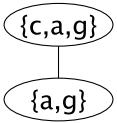
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h\}$,
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\}$

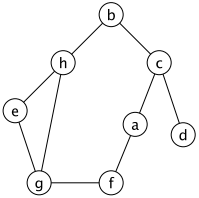


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

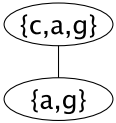
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h, c, g\}$
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a)\} E = \{(t_f, t_a), (t_a, t_g)\}$

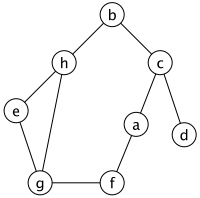


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

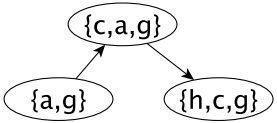
Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$
3. $i = 3$: ...



$\chi(t_e) = \{g, h\}$,
$\chi(t_g) = \{h, c, g\}$
$\chi(t_h) = \{b\}$,
$\chi(t_c) = \{b\}$,
$\chi(t_d) = \{c\}$,
$\chi(t_a) = \{c, a, g\}$,
$\chi(t_f) = \{a, g\}$,
$\chi(t_b) = \{\}$

$E = \{(t_f, t_a), (t_a, t_g)\}$

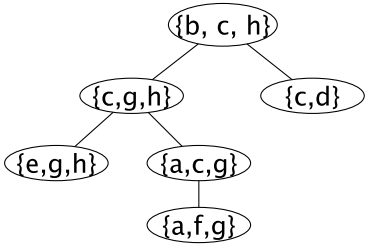


Thats also how we construct a TD from a linear ordering that was obtained by a

htd: Bucket Elimination Example

Input G (right); Ordering
$f < a < e < g < d < c < h < b$

1. $E := \emptyset$
2. $N = \{t_f, t_a, t_e, t_g, t_d, t_c, t_h, t_b\}$ and $\chi(t_n) = \emptyset$ for $n \in N$
3. $i = 2$:
   (1) $A = \chi(t_{v_2}) \setminus \{v_2\} = \chi(t_a) \setminus \{a\} = \{c, g\}$
   (2) $v_j = g$
   (3) $\chi(t_g) = \chi(t_g) \cup A = \{h\} \cup \{c, g\}$
   (4) $E = E \cup \{t_a t_g\}$
3. $i = 3$: ...



(+ some simplification)

Thats also how we construct a TD from a linear ordering that was obtained by a SAT solver using the SAT encoding from above

htd: MinFill

- Greedy Figure taken from Michael Abseher's PhD Thesis [Abseher'17, Alg 4.8]

Algorithm 4.2: Min-Fill (Verbose Pseudo-Code)

Input: A (constraint) hypergraph $\mathcal{H} = (V, H)$

Result: A vertex elimination ordering $\sigma = \{\sigma_1 \dots \sigma_n\}$ of the vertices in V

1 Let $\mathcal{G} = (V, E)$ be the primal graph of $\mathcal{H}$.

2 $\sigma \leftarrow []$; // List σ is initially empty.

3 $fill \leftarrow []$; // The dictionary for the fill value of each vertex.

4 $pool \leftarrow \emptyset$; // The set of vertices with minimum fill value.

5 $minfill \leftarrow \infty$; // The minimum fill value.

6 Initialize $fill$, $minfill$ and $pool$ based on $\mathcal{G}$.

htd: MinFill

- Greedy Figure taken from Michael Abseher's PhD Thesis [Abseher'17, Alg 4.8]

```
while G is not empty do
    if pool == ∅ then
        Initialize minfill and pool based on fill.
    end
    Select randomly a vertex v_x ∈ pool;
    if fill[v_x] == 0 then
        for v ∈ N_1(v_x) \ {v_x} where fill[v] > 0 do
            fill[v] ← fill[v] − |N_1(v) \ N_1(v_x)|;
        end
    else
```

htd: MinFill

- Greedy Figure taken from Michael Abseher's PhD Thesis [Abseher'17, Alg 4.8]

```
else
    for v ∈ N_1(v_x) \ {v_x} do
        fill[v] ← updateNeighbor(G, v_x, v, fill[v]);
    end
    for v ∈ N_2(v_x) \ ({v_x} ∪ {u|u ∈ N_1(v_x), N_u(u) == ∅ || N_a(u) == ∅}) do
        for u ∈ N_e(v) do
            fill[v] ← fill[v] − |{t|t ∈ N_e(v), t > u} ∩ N_a(u)|;
        end
    end
end
```

htd: MinFill

- Greedy Figure taken from Michael Abseher's PhD Thesis [Abseher'17, Alg 4.8]

```
    Add necessary fill-in edges;
    Remove v_x from G;
    Append v_x to σ;
end
return σ;
```

Computing Tree Decompositions

Hybrid: Local Improvement

How to compute TD with SAT?

SAT encoding by Samer & Veith (09)

- Construction uses elimination orderings (alternative characterization of TDs)
- #variables: $\mathcal{O}(k|V|^2)$ and #clauses: $\mathcal{O}(|V|^3)$
- Works for at most 100 vertices

▸ Details on Encoding

Incremental SAT approach by Berg & Järvisalo (14)

Research Question

Can we efficiently find TDs of small width for larger instances?

Local Improvement [FLS'17]

Overall Idea

1. Compute a TD heuristically (global solver)
2. Select parts of the TD, improve with another method (local solver)

$\Rightarrow$ SAT-based exact methods are particularly well-suited for local solver

Related Work

- Idea of local improvement for branchwidth [Lodha, Ordyniak, Szeider SAT'16]

Overview

Given: Graph G and global decomposition $\mathcal{T}$

What do we need for local improvement?

Method to:

- extract parts of global decomposition $\mathcal{T}$,
- construct "local graph" from which we compute "local TD"
- merge "local TD" back into the TD.

Overview

Given: Graph G and global decomposition $\mathcal{T}$

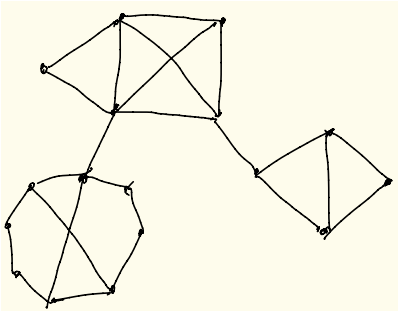
What do we need for local improvement?

Method to:

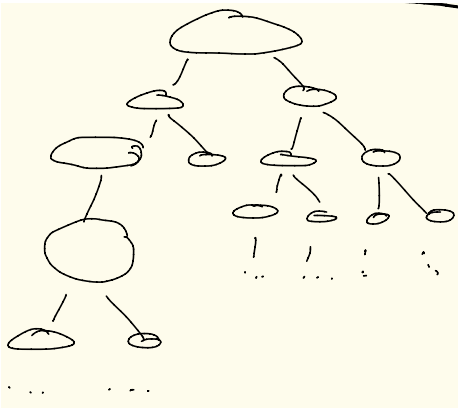
- extract parts of global decomposition $\mathcal{T}$,
- construct "local graph" from which we compute "local TD"
- merge "local TD" back into the TD.

Local Improvement (Extraction & Merge)

Input Graph G



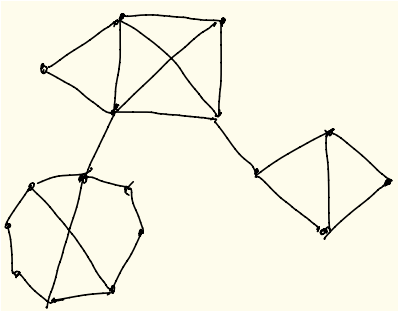
Tree Decomposition of G



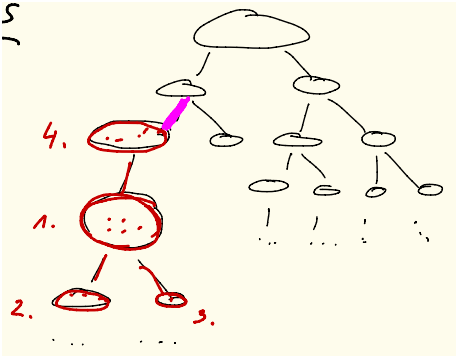
- Select largest bag
- Run BFS selecting bags as long as we are in given limit
- Construct subgraph of G wrt. bags selected by BFS

Local Improvement (Extraction & Merge)

Input Graph G



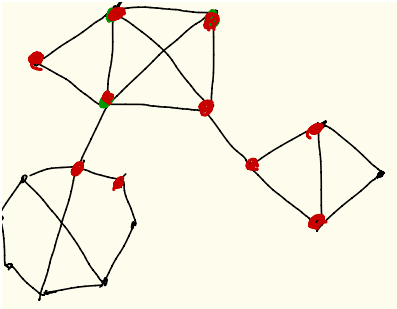
Tree Decomposition of G



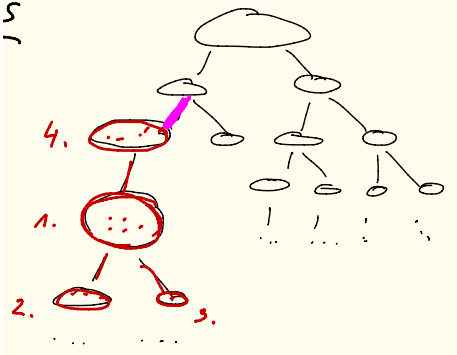
- Select largest bag
- Run BFS selecting bags as long as we are in given limit
- Construct subgraph of G wrt. bags selected by BFS

Local Improvement (Extraction & Merge)

$G \Rightarrow$ local graph G_S

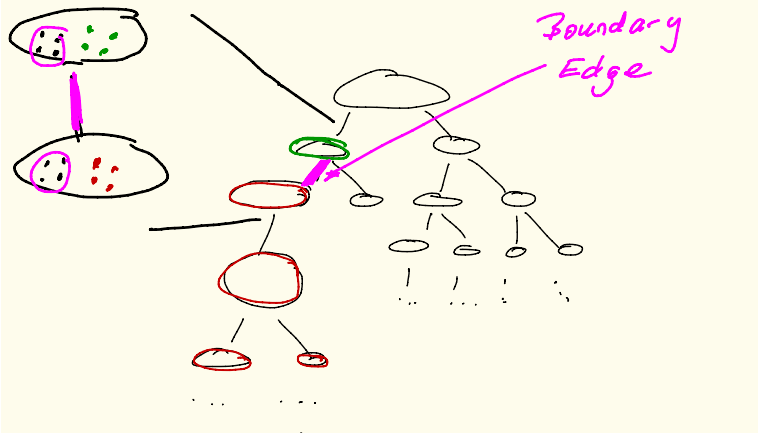


Tree Decomposition of G



- Select largest bag
- Run BFS selecting bags as long as we are in given limit
- Construct subgraph of G wrt. bags selected by BFS

Local Improvement (Extraction & Merge)



- By BFS selected part (red) and not-selected part

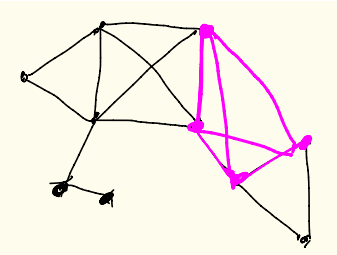
⇒ Obtain cuts (intersection of “boundary bags”)

⇒ Construct a “marker clique” on cuts in “local graph”

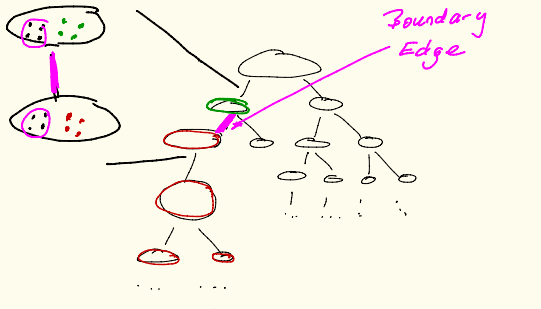
- Compute TD of the local graph

Local Improvement (Extraction & Merge)

Local Graph G_S



TD Cuts



- By BFS selected part (red) and not-selected part

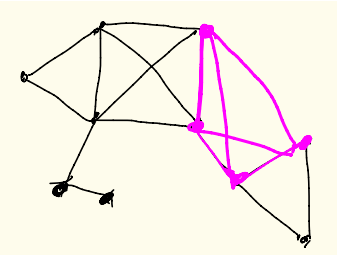
⇒ Obtain cuts (intersection of "boundary bags")

⇒ Construct a "marker clique" on cuts in "local graph"

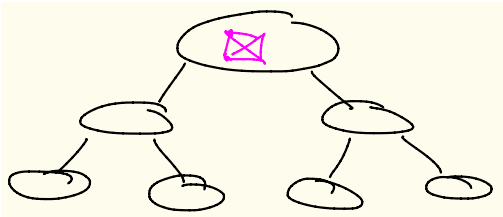
- Compute TD of the local graph
- Marker clique ensures in "local TD" to find the bag, which we have to attach

Local Improvement (Extraction & Merge)

Local Graph G_S



TD of local graph



- By BFS selected part (red) and not-selected part

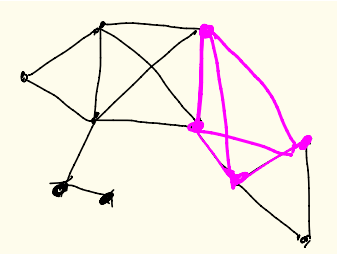
⇒ Obtain cuts (intersection of "boundary bags")

⇒ Construct a "marker clique" on cuts in "local graph"

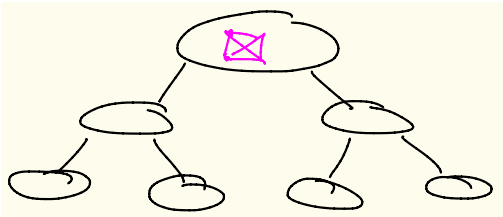
- Compute TD of the local graph
- Marker clique ensures in "local TD" to find the bag, which we have to attach to the new "global" TD

Local Improvement (Extraction & Merge)

Local Graph G_S



TD of local graph



- By BFS selected part (red) and not-selected part

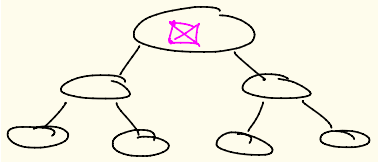
⇒ Obtain cuts (intersection of "boundary bags")

⇒ Construct a "marker clique" on cuts in "local graph"

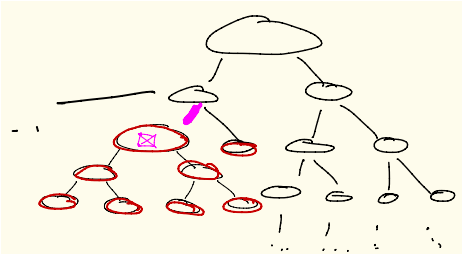
- Compute TD of the local graph
- Marker clique ensures in "local TD" to find the bag, which we have to attach to the new "global" TD

Local Improvement (Extraction & Merge)

TD of G_S



New TD



- Merge local TD into global TD $\mathcal{T}$

$\Rightarrow$ Obtain new TD $\mathcal{T}_\infty$

$\Rightarrow$ Observation: $\text{width}(\mathcal{T}_\infty) \leq \text{width}(\mathcal{T})$

Extended Version

Experimental Work (Setting)

Graphs

- Initial selection of overall 3168 graphs
- Publicly available graph sets
  - TreewidthLIB graphs (Broek & Bodlaender'10)
  - UAI competition instances (Dechter'13)
  - PACE 2016 treewidth challenge graphs (Dell'16)
  - Transit graphs from GTFS-transit feeds (Fichte'16)

Restriction

- Interest: larger graphs
- Requirement: more than 100 vertices

$\Rightarrow$ 1946 graphs in total

Experimental Work: Solvers and Tuning

Solvers

- heur: htd [Abseher'16]
- sat: Jdrasil [Bannach et. al'16];
  Samer & Veith'09 encoding; Glucose + PBLib for cardinalities
- comb: tcs-mejin [Tamaki'16];
  Arnborg et. al'87 combinatorial algorithm;
- global: heur
- local: heur, sat, comb

Parameters (positive integers) for the loop

1. the local budget **lb**,
2. the local timeout **lt**: 1800 sec (900 per SAT call),
3. the global timeout **gt**,
4. the number of no-improvement rounds **ni**: 10

Experimental Work: Results

Summary of treewidth improvements

#	sum	max	solver	**lb**
647	**2015**	13	sat	100
584	1984	16	sat	125
630	1805	15	comb	100
493	1676	**20**	sat	150
631	1548	12	comb	075
609	1460	12	comb	075
447	1077	19	comb	125
368	822	14	comb	150
325	538	9	heur	150
258	421	8	heur	100

- Heuristic: improved
- #: difference small
- SAT-based: best (# + sum + max)
- Local budget **lb** $= 100$: more solved instances

#... number of improved instances

sum... cumulative sum of improvements over all instances; max... maximum improvement

Experimental Work: Results (cont.)

Some of the best and notable improvements

instance	$\|V\|$	$\|E\|$	graphs	itw	htw	diff	solver	**lb**
or_chain_224.fg	1638	3255	networks	75	95	**20**	sat	150
or_chain_54.fg	1404	2757	networks	65	84	19	comb	125
1bkr_graph	107	1340	twlib	44	56	12	comb	075
dimacs_fpsol2.i.1-pp	191	4418	pace2016	61	72	11	sat	150
Promedus_38	668	1235	networks	16	23	7	sat	150

Overall

- Best results: SAT-based solver as local solver

⇒ Immediately better upper bounds by improved encoding or new techniques

- VBS: +200 instances than best SAT-based configuration

⇒ Portfolio solver: improve more of instances

Experimental Work: Results (cont.)

Some of the best and notable improvements

instance	$\|V\|$	$\|E\|$	graphs	itw	htw	diff	solver	**lb**
or_chain_224.fg	1638	3255	networks	75	95	**20**	sat	150
or_chain_54.fg	1404	2757	networks	65	84	19	comb	125
1bkr_graph	107	1340	twlib	44	56	12	comb	075
dimacs_fpsol2.i.1-pp	191	4418	pace2016	61	72	11	sat	150
Promedus_38	668	1235	networks	16	23	7	sat	150

Overall

- Best results: SAT-based solver as local solver

⇒ Immediately better upper bounds by improved encoding or new techniques

- VBS: +200 instances than best SAT-based configuration

⇒ Portfolio solver: improve more of instances

Summary (Local Improvement)

Hybrid Approach

- SAT-based approach to find TD of small width
- Employs local improvement
- Marker cliques allow splitting and merging
- Cross-over between heuristical methods and exact methods
  - Global solver: heuristic
  - Local solver: SAT or combinatorial solver

  $\Rightarrow$ SAT outperforms combinatorial solver (even though not the best exact solver)
- Seen experimental work

Interesting for the Future

Future work:

1. Improve current setup by:
   - Upgrading method for selecting local TD (BFS)
   - Tuning and optimizing the SAT-based local solver (ML?)
   - Improve just by one instead of optimal local TD
2. Customized tree decompositions
   (Additional constraints to speed-up DP algorithms)
   easy for SAT but difficult for combinatorial methods
3. Automated algorithm configuration and parameter tuning

Interesting for the Future

Future work:

1. Improve current setup by:
   - Upgrading method for selecting local TD (BFS)
   - Tuning and optimizing the SAT-based local solver (ML?)
   - Improve just by one instead of optimal local TD
2. Customized tree decompositions
   (Additional constraints to speed-up DP algorithms)
   easy for SAT but difficult for combinatorial methods
3. Automated algorithm configuration and parameter tuning

Interesting for the Future

Future work:

1. Improve current setup by:
   - Upgrading method for selecting local TD (BFS)
   - Tuning and optimizing the SAT-based local solver (ML?)
   - Improve just by one instead of optimal local TD
2. Customized tree decompositions
   (Additional constraints to speed-up DP algorithms)
   easy for SAT but difficult for combinatorial methods
3. Automated algorithm configuration and parameter tuning

What's next?

Would suggest to do a hands-on session.

Where we get into some of those tools. You install them and maybe construct a tree decomposition yourself with one of the tools.

What's next?

Would suggest to do a hands-on session.

Where we get into some of those tools. You install them and maybe construct a tree decomposition yourself with one of the tools.

What's next?
Would suggest to do a hands-on session.

Where we get into some of those tools. You install them and maybe construct a tree decomposition yourself with one of the tools.

Computing Tree Decompositions

Backup Slides

Treewidth Basics (cont.)

Folklore, e.g., Robertson & Seymour (1986)

Let G be a graph and $C_1, \ldots, C_l$ its connected components, then $\mathrm{tw}(G) = \max\{\, \mathrm{tw}(C_j) : 1 \leq i \leq l \,\}$.

Folklore, e.g., Bodlaender & Koster (2008)

Let G be a graph. If G has a feedback vertex set size at most k, then $\mathrm{tw}(G) \leq k + 1$.

Triangulation and Treewidth

TODO

How to compute TD with SAT?

SAT encoding by Samer & Veith (09)

- In: Graph G, integer k
- Out: CNF formula $\Phi(G, k)$
- Such that: satisfiable iff G has a TD of width $\leq k$
- Construction of $\Phi(G, k)$ uses elimination orderings (alternative characterization of TDs)
- $\mathcal{O}(k|V|^2)$ variables and $\mathcal{O}(|V|^3)$ clauses
- Works for at most 100 vertices

▸ Details on Encoding

Incremental SAT approach by Berg & Järvisalo (14)

Samer & Veith Encoding (09)

Elimination ordering

- Linear ordering of vertices in G guessed
- Based on the ordering "fill-in edges" added to the graph

$\Rightarrow$ Triangulation of G

Encoding

- Ordering represented by Boolean variables
- One variable for every pair uv of vertices
- Truth value indicate relative ordering of uv
- Transitivity of the ordering (suitable clauses)
- For each vertex v:
  - v has at most k neighbors right to v in ordering
  - using cardinality constraints (see Sinz'05)
- Exact treewidth:
  - systematically calling solver for heuristic upper bound u
  - $\Phi(G, k)$ for $k = u, u-1, u-2, \ldots$ and until $\Phi(G, k)$ UNSAT
- Satisfying assignment $\Rightarrow$ efficiently obtain a TD of width k

Overview

Given: Graph G

What do we need for local improvement?

Method to extract parts of a decomposition, construct a "local graph", compute TD of the "local graph", and merge back.

1. Global:
   Extract parts of TD of width h
2. Local Loop:
   a) Extract "local graph"
      (depending on parts of TD and G)
   b) SAT encoding for local graph
      Q: Local graph has TD of size at most h?
   c) Y: Replace parts of (global) TD with new TD.

- Stop after a predefined number of no-improvement rounds **ni** or overall timeout **gt**,

Local Improvement (more detailed)

Input: G graph

Main Loop

1. Compute TD $\mathcal{T} = (T, \chi)$ of G by heuristic (global solver)
2. Select node t from T with largest bag size
3. Obtain "local graph" (modified BFS)
   Next: get TD, width small as possible (start $k = \text{width}(\mathcal{T}) - 1$)
4. SAT encoding $\Phi(G_S, k)$ of TD of local graph G_S
5. SAT check (local solver)
   - Yes: decrease k and proceed with Step 4
   - No or local timeout (**lt**): Insert new TD into $\mathcal{T}$ and proceed with Step 2

- Stop after a predefined number of no-improvement rounds **ni** or overall timeout **gt**,
- k in the local loop $\Rightarrow$ treewidth of G_S at most $k + 1$

Local Improvement (more detailed)

Input: G graph

Main Loop

1. Compute TD $\mathcal{T} = (T, \chi)$ of G by heuristic (global solver)
2. Select node t from T with largest bag size
3. Obtain "local graph" (modified BFS)
   Next: get TD, width small as possible (start $k = \text{width}(\mathcal{T}) - 1$)
4. SAT encoding $\Phi(G_S, k)$ of TD of local graph G_S
5. SAT check (local solver)
   - Yes: decrease k and proceed with Step 4
   - No or local timeout (**lt**): Insert new TD into $\mathcal{T}$ and proceed with Step 2

- Stop after a predefined number of no-improvement rounds **ni** or overall timeout **gt**,
- k in the local loop $\Rightarrow$ treewidth of G_S at most $k + 1$

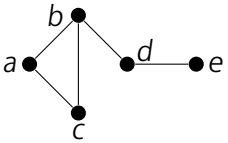
How to patch things back in? (cont.)

Construct New TD $\mathcal{T}_1 = (T_1, \chi_1)$ of G

- $T_1, \ldots, T_r$ connected components of $T - S$ (each T_i is a tree)
- Local tree decomposition $\mathcal{T}_i = (T_i, \chi_i)$ from T_i
- Tree:
  - Nodes: old nodes without those where we improved bags + nodes of new decomposition
  - Edges: old edges + new edges + connectors ($t_i s_i^1$)

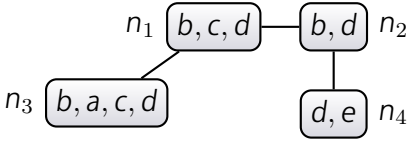
1) Select Parts of TD

Graph G



$\mathcal{T} = (T, \chi)$ tree decomposition

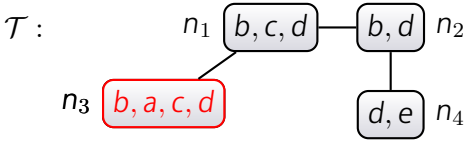
$\mathcal{T}$:



Limit max. #vertices (of G) to consider (lb)

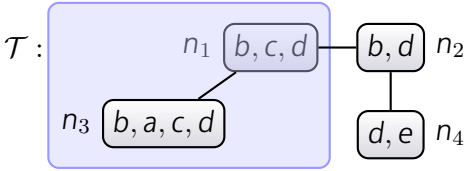
Example: 4

1) Select Parts of TD



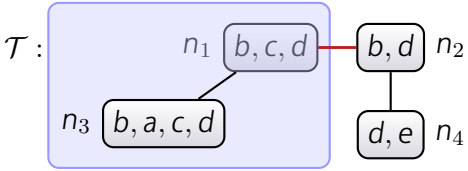
Select node t with
largest bag size

1) Select Parts of TD



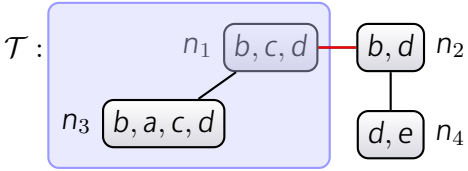
Run BFS as long we are in the limit (#vertices ≤ 4)
(but stop here for demonstration purposes)

1) Select Parts of TD



Run BFS as long we are in the limit (#vertices ≤ 4)
(but stop here for demonstration purposes)

1) Select Parts of TD

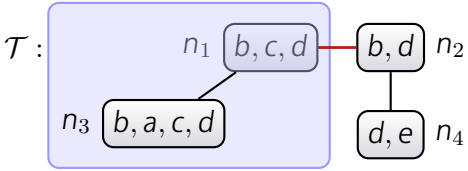


Boundary edge $n_1 n_2$.

Next:

- Extract bag content from the input graph
  ($\Rightarrow$ obtain a "local graph") such that
- we can reinsert the resulting TD of the local graph back

1) Select Parts of TD

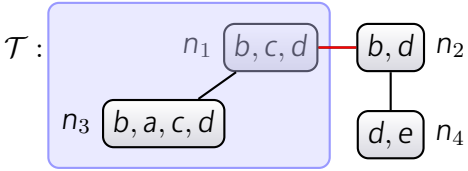


Extract "local graph" such that resulting TD can go back in

Idea:

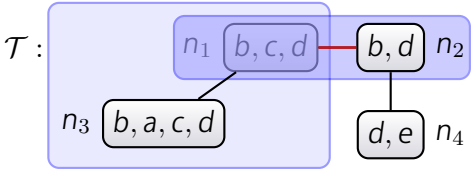
Marker Cliques

1) Select Parts of TD



Only boundary edge $n_1 n_2$

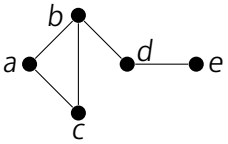
1) Select Parts of TD (Idea: Marker Clique)



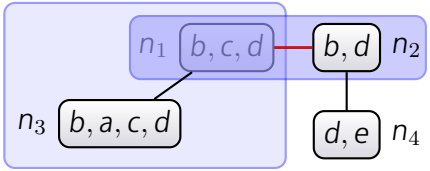
Only boundary edge $n_1 n_2$
$\Rightarrow$ Obtain cut sets
(over bags) from boundary edges
Here: $\lambda_{\mathcal{T}}(n_1 n_2) =$
$\{b, c, d\} \cap \{c, d\} = \{b, d\}$

2a) Local Graph (Idea: Marker Clique)

Graph G



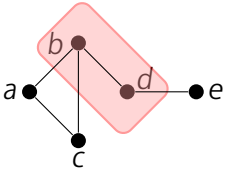
$\mathcal{T}$:



Cut set: $\lambda_{\mathcal{T}}(n_1 n_2) = \{b, d\}$

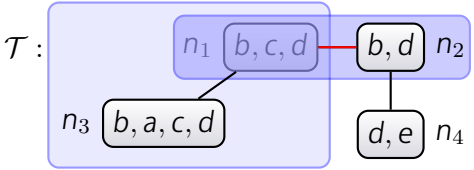
2a) Local Graph (Idea: Marker Clique)

Graph G



- Here: only on set $\{b, d\}$

$\Rightarrow$ Same as before

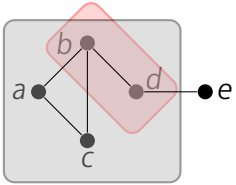


Cut set: $\lambda_{\mathcal{T}}(n_1 n_2) = \{b, d\}$

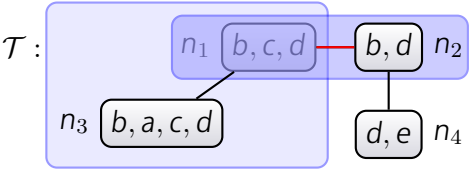
I) Cliques on vertices of cut sets
(associated with boundary edges)

2a) Local Graph (Idea: Marker Clique)

Graph G



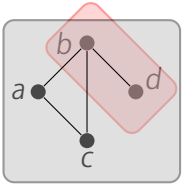
- Vertices considered by BFS: $\{a, b, c, d\}$



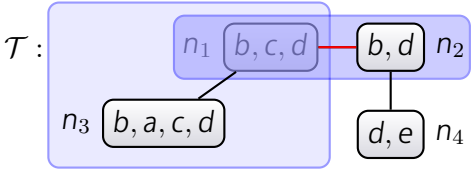
I) Cliques on vertices of cut sets (associated with boundary edges) +
II) Restrict to vertices occurring in bags considered by BFS

2a) Local Graph (Idea: Marker Clique)

Graph G_S

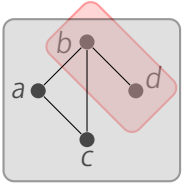


- Take local graph G_S
- Compute a TD of G_S
- Preferably of smaller width than before

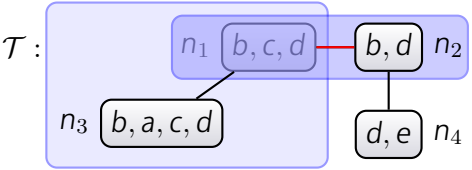


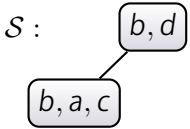
2b) New TD (Idea: Marker Clique)

Graph G_S



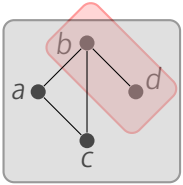
- Take local graph G_S
- Compute a TD of G_S
- Preferably of smaller width than before

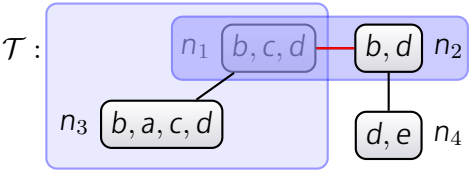


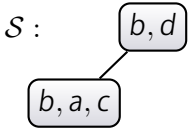


2c) Update Global TD (Idea: Marker Clique)

Graph G_S



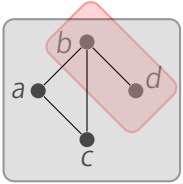


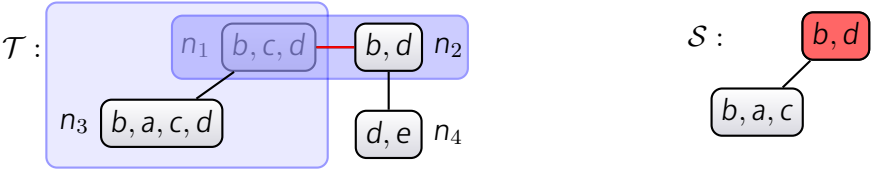


Merge local TD $\mathcal{S}$ back into the global TD $\mathcal{T}$

2c) Update Global TD (Idea: Marker Clique)

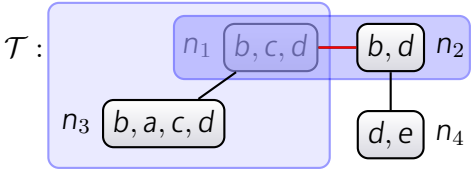
Graph G_S

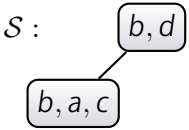


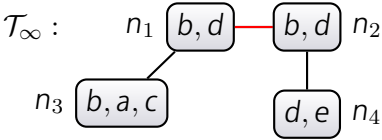


Basic Property of TDs: cliques occur together in one bag
$\Rightarrow$ Marker clique ensures that we can patch things back in

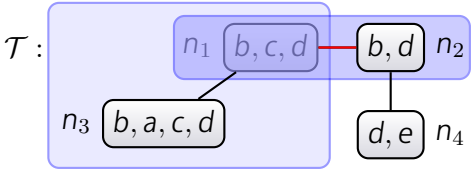
2c) Update Global TD (Idea: Marker Clique)

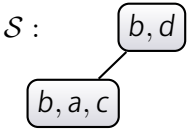


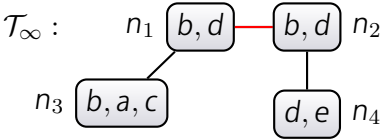




2c) Update Global TD (Idea: Marker Clique)







Observation

New TD $\mathcal{T}_1$ is a TD of G and width$(\mathcal{T}_1) \leq$ width$(\mathcal{T})$

How to patch things back in? (cont.)

Construct New TD $\mathcal{T}_{=}(T_1, \chi_1)$ of G (cont.)

- Connectors:
  - t_i leaf of T_i incident with a boundary edge $e_i = t_i s_i$ in T
  - Boundary edge e_i responsible for marker clique on vertices in cut set of e_i
  - s_i^1 of S^* s.t. vertices of marker clique coincide with cut set

Construct New TD $\mathcal{T}_1 = (T_1, \chi_1)$ of G (cont.)

- Bags take new bags + remaining old bags More Details

Observation

$\mathcal{T}_1$ is a TD of of G and $\text{width}(\mathcal{T}_1) \leq \text{width}(\mathcal{T})$

More Detailed:
$\text{width}(\mathcal{T}_1) = \max(\text{width}(\mathcal{T}^1), \ldots, \text{width}(\mathcal{T}^r), \text{width}(\mathcal{S}^*)) \leq \max(\text{width}(\mathcal{T}), \text{width}(\mathcal{S}^*)) \leq \max(\text{width}(\mathcal{T}), \text{width}(\mathcal{S})) \leq \text{wi}$

New Tree Decomposition (Definitions)

- Tree T_1:
  - Vertices: $V(S^*) \cup \bigcup_{i=1}^{r} V(T_i) = V(S^*) \cup V(T) \setminus V(S)$
  - Edges: $E(S^*) \cup \bigcup_{i=1}^{r} E(T_i) \cup \{t_1 s_1^1, \ldots, t_r s_r^1\}$.
- Bags χ_1:
  - For $t \in V(T_i)$ we define $\chi_1(t) = \chi(t)$
  - and for $s \in V(S^*)$ we define $\chi_1(s) = \chi^*(s)$

Augmented Local Graph (Definitions)

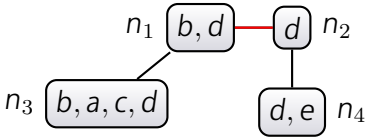
Definition (Augmented local graph $G^*_{\mathcal{S}}$)

- Vertices of G that appear in a bag of $\mathcal{S}$
- Edges: $u, v \in V^*$ such that $uv \in E(G)$ or $u, v \in \lambda_{\mathcal{T}}(e)$ for a boundary edge e of T

Select Parts of TD

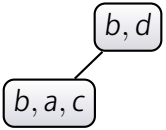
Graph G, $\mathcal{T} = (T, \chi)$ tree decomposition

$\mathcal{T}$:

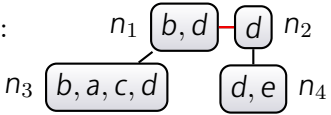


Limit max. #bagsize (lb)
Example: 4

$\mathcal{S}^*$:

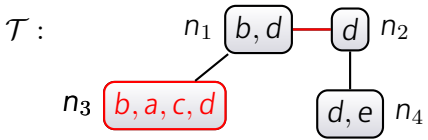


$\mathcal{T}$:



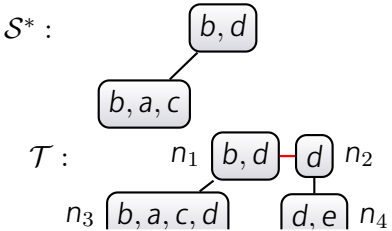
Definition

Select Parts of TD



Select node t with largest bag size

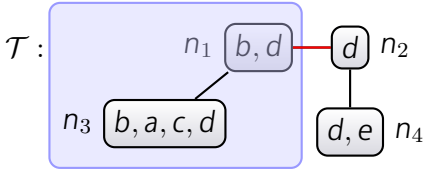
$\Rightarrow$



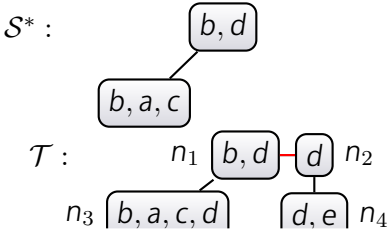
Definition

- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ to nodes of S

Select Parts of TD



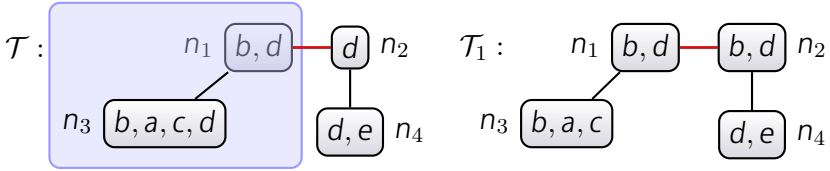
Run BFS until we reach the limit

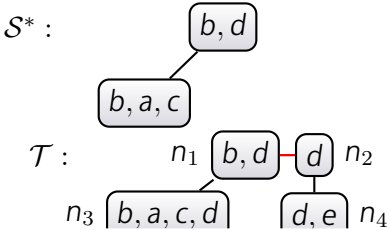


Definition

- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ

Select Parts of TD





Definition

- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ

Local Tree Decomposition

Graph G, $\mathcal{T} = (T, \chi)$ tree decomposition, S subtree of T

Definition

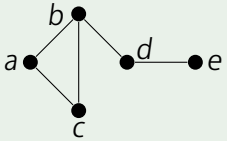
- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ to nodes of S
- $G_{\mathcal{S}} \subseteq G$ induced by vertices of G appearing in bags of $\mathcal{S}$

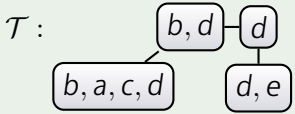
Local Tree Decomposition

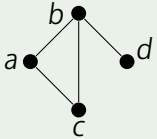
Definition

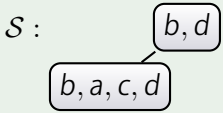
- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ to nodes of S
- $G_{\mathcal{S}} \subseteq G$ induced by vertices of G appearing in bags of $\mathcal{S}$

Tree Decomposition of Graph G







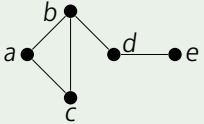


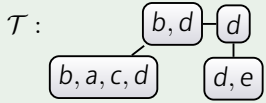
Local Tree Decomposition

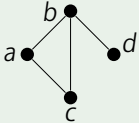
Definition

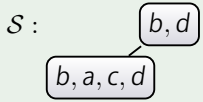
- $\mathcal{S} = (S, \chi_S)$ local tree decomposition of $\mathcal{T}$ (induced by S), χ_S restriction of χ to nodes of S
- $G_{\mathcal{S}} \subseteq G$ induced by vertices of G appearing in bags of $\mathcal{S}$

Tree Decomposition of Graph G









Observation

$\mathcal{S}$ is a tree decomposition of $G_{\mathcal{S}}$ and $\text{width}(G_{\mathcal{S}}) \leq \text{width}(\mathcal{T})$.

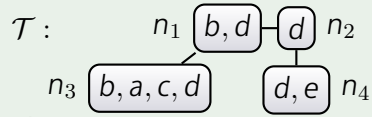
Cut Set and Boundary Edge

Definition

- Edge st of $\mathcal{T}$, cut set $\lambda_{\mathcal{T}}(st) = \chi(s) \cap \chi(t)$
- Edge st of T, boundary edge if $s \in V(S)$ and $t \notin V(S)$

Cut set and Boundary Edge

$\mathcal{T}$:



Edge: $n_1 n_3$

Cut set:

$\lambda_{\mathcal{T}}(n_1 n_3) = \{b, d\} \cap \{b, a, c, d\} = \{b, d\}$

$\mathcal{T}$:

$\mathcal{S}$:

Boundary edge: $n_1 n_2$

$s \in V(S)$ and $t \notin V(S)$

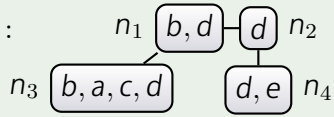
Cut Set and Boundary Edge

Definition

- Edge st of $\mathcal{T}$, cut set $\lambda_{\mathcal{T}}(st) = \chi(s) \cap \chi(t)$
- Edge st of T, boundary edge if $s \in V(S)$ and $t \notin V(S)$

Cut set and Boundary Edge

$\mathcal{T}$:



Edge: $n_1 n_3$

Cut set:

$\lambda_{\mathcal{T}}(n_1 n_3) = \{b, d\} \cap \{b, a, c, d\} = \{b, d\}$

$\mathcal{T}$:

$\mathcal{S}$:

Boundary edge: $n_1 n_2$

$s \in V(S)$ and $t \notin V(S)$

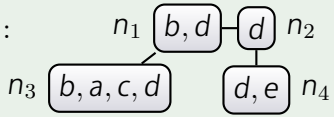
Cut Set and Boundary Edge

Definition

- Edge st of $\mathcal{T}$, cut set $\lambda_{\mathcal{T}}(st) = \chi(s) \cap \chi(t)$
- Edge st of T, boundary edge if $s \in V(S)$ and $t \notin V(S)$

Cut set and Boundary Edge

$\mathcal{T}$:

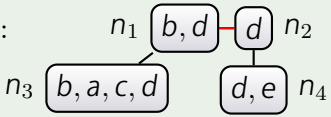


Edge: $n_1 n_3$

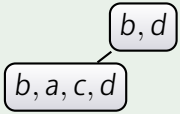
Cut set:

$\lambda_{\mathcal{T}}(n_1 n_3) = \{b, d\} \cap \{b, a, c, d\} = \{b, d\}$

$\mathcal{T}$:



$\mathcal{S}$:



Boundary edge: $n_1 n_2$

$s \in V(S)$ and $t \notin V(S)$

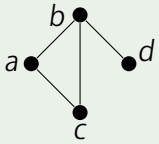
Augmented Local Graph

Definition

Obtained from $G_{\mathcal{S}}$ by forming cliques over cut sets associated with boundary edges ▸ Details

Augmented Local Graph $G^*_{\mathcal{S}}$

$G_{\mathcal{S}}$:



Observation

$\mathcal{S}$ is a tree decomposition of $G^*_{\mathcal{S}}$ and $\text{width}(\mathcal{S}) < \text{width}(\mathcal{T})$.

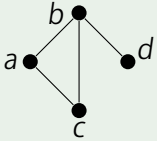
Augmented Local Graph

Definition

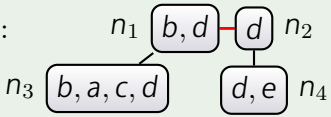
Obtained from $G_{\mathcal{S}}$ by forming cliques over cut sets associated with boundary edges ▸ Details

Augmented Local Graph $G^*_{\mathcal{S}}$

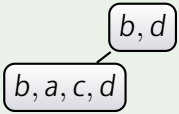
$G_{\mathcal{S}}$:



$\mathcal{T}$:



$\mathcal{S}$:



Boundary Edge: $n_1 n_2$
Cut set: $\{b, d\}$

Observation

$\mathcal{S}$ is a tree decomposition of G^* and width$(\mathcal{S}) \leq$ width$(\mathcal{T})$

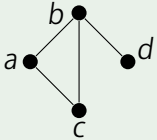
Augmented Local Graph

Definition

Obtained from $G_{\mathcal{S}}$ by forming cliques over cut sets associated with boundary edges ▸ Details

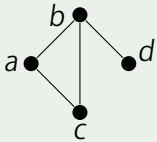
Augmented Local Graph $G^*_{\mathcal{S}}$

$G_{\mathcal{S}}$:



$\Rightarrow$

$G^*_{\mathcal{S}}$:



Boundary Edge: $n_1 n_2$
Cut set: $\{b, d\}$

Here: same; there is already an edge between b and d

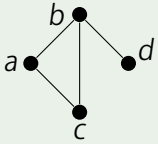
Observation

$\mathcal{S}$ is a tree decomposition of G^* and $\text{width}(\mathcal{S}) \leq \text{width}(\mathcal{T})$

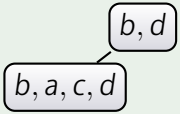
How to patch things back in?

Compute TD of augmented local graph.

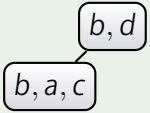
$G^*_{\mathcal{S}}$:



$\mathcal{S}$:



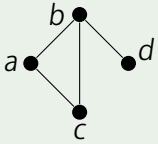
$\mathcal{S}^*$:



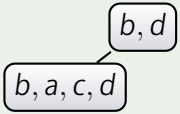
How to patch things back in?

Compute TD of augmented local graph.

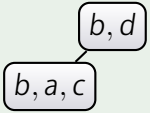
$G^*_{\mathcal{S}}$:



$\mathcal{S}$:



$\mathcal{S}^*$:

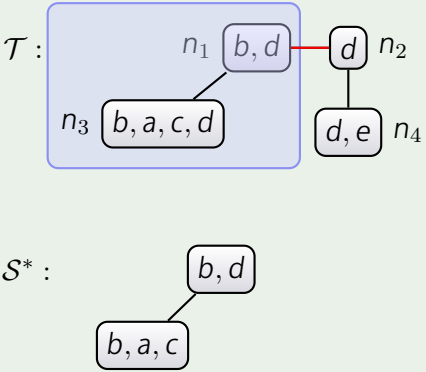


"Improved TD":

- $\mathcal{S}^* = (S^*, \chi^*)$ TD of $G^*_{\mathcal{S}}$ with $\text{width}(\mathcal{S}^*) \leq \text{width}(\mathcal{S})$

How to patch things back in?

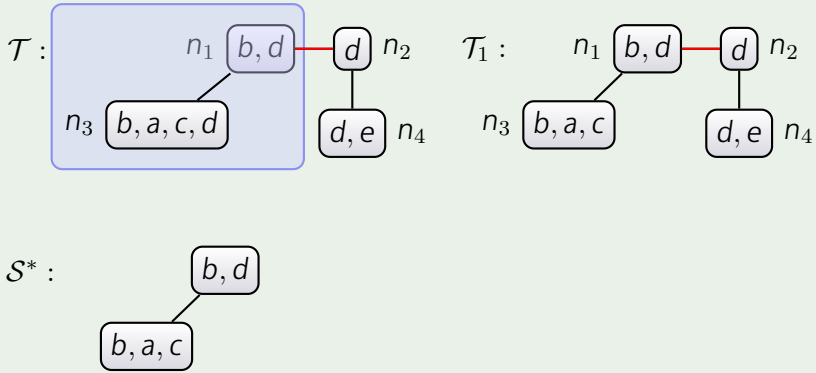
Merge In.



- $T_1, \ldots, T_r$ connected components of $T - S$ (each T_i is a tree)
- Boundary edge responsible for marker clique
- Marker clique coincides with cut set ▸ Details

How to patch things back in?

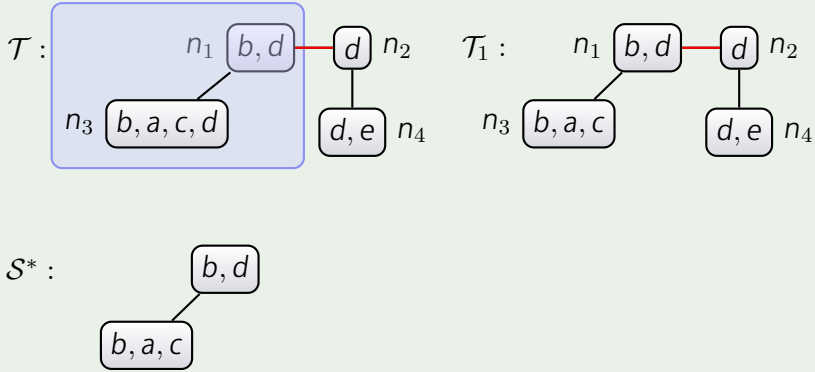
Merge In.



- $T_1, \ldots, T_r$ connected components of $T - S$ (each T_i is a tree)
- Boundary edge responsible for marker clique
- Marker clique coincides with cut set ▸ Details

How to patch things back in?

Merge In.



Observation

$\mathcal{T}_1$ is a TD of of G and $\text{width}(\mathcal{T}_1) \leq \text{width}(\mathcal{T})$

Local Improvement (cont.)

BFS (breadth-first-search)

Given: Local budget **lb** (positive integer)

1. Auxiliary variables $L = \chi(t)$
2. Visit t_1: add elements of $\chi(t_1)$ to L
   - If node t_1 visited via an edge e, neighbor t_2 of t_1 only visited if cut set smaller than for e ($|\lambda_{\mathcal{T}}(t_1 t_2)| < |\lambda_{\mathcal{T}}(e)|$)
3. Stop: if visiting another node yields $|L| >$ **lb**

- Visited nodes induce subtree S of T

$\Rightarrow$ Yields a local tree decomposition $\mathcal{S} = (S, \chi_S)$ of $\mathcal{T}$

- L contains vertices of local graph $G_{\mathcal{S}}$ (or augmented local graph $G^*_{\mathcal{S}}$)

$\Rightarrow$ At most **lb** many vertices

Propositional Model Counting

Propositional Model Counting

Motivation

Purpose of this Chapter

1. Get familiar with Model Counting
2. Solve Model Counting for Instances of Low Treewidth
3. Give some insights into GPU-based solving

Motivation

Model Counting (#SAT)

- Generalizes Boolean satisfiability problem (SAT)
- #SAT: output the number of satisfying assignments
- Various applications in AI and reasoning, e.g.,
  - Bayesian reasoning [Sang et al.'05]
  - Learning preference distributions [Choi et al.'15]
  - Infrastructure reliability [Meel et al.17]
- Computational complexity: #P-hard [Roth'96]

Problem of Interest

SAT-Problem (Boolean Satisfiability Problem)

Given: Propositional formula F.
Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Skip Example

Input normal form

- Conjunctive normal form (CNF)
- Form: $F = (\ell_1 \vee \ell_2 \vee \ell_3) \wedge \ldots \wedge (\ldots)$ where ℓ_i either x or $\neg x$

#SAT (Number SAT)

- Number of of satisfying truth assignments to F.

Problem of Interest

SAT-Problem (Boolean Satisfiability Problem)

Given: Propositional formula F.
Question: Is there a truth assignment τ to the variables in F such that F_τ evaluates to 1 (satisfiable).

Skip Example

Example

$$(a \vee \neg b \vee d) \wedge (\neg a \vee \neg c \vee \neg d) \wedge$$
$$(\neg a \vee \neg d) \wedge (b \vee c) \wedge (d \vee e) \wedge (b \vee \neg e)$$

#SAT (Number SAT)

- Number of of satisfying truth assignments to F.

The Problem of Interest

#SAT (Number SAT)

Given: Propositional formula F.

Output: Number of truth assignments τ such that the formula F_τ under the truth assignment evaluates to 1 (satisfiable).

Problem of Interest cont.

Weighted Model Counting (WMC)

- Generalizes #SAT
- Given Boolean formula, e.g., $F = (x \vee y) \wedge (\neg x \vee \neg y)$

  Function that maps literals to reals between 0 and 1, e.g.,

  $x \mapsto 0.4$, $\neg x \mapsto 0.6$, $y \mapsto 0.7$, $\neg y \mapsto 0.3$
- Weight of an assignment α is the product over the weights of its literals, i.e., $w(\alpha) := \Pi_{v \in \alpha^{-1}(1)} w(v) \cdot \Pi_{v \in \alpha^{-1}(0)} w(\neg v)$,

  e.g., $\alpha(x) = 1$, $\alpha(y) = 0 \Rightarrow w(\alpha) = 0.4 \cdot 0.3 = 0.16$
- Weighted model count (WMC) of formula is the sum of weights over all its satisfying assignments, e.g.,

  $w(F) = w(\alpha_1) + w(\alpha_2) = 0.12 + 0.42 = 0.54$

Problem of Interest cont.

Weighted Model Counting (WMC)

- Generalizes #SAT
- Given Boolean formula, e.g., $F = (x \vee y) \wedge (\neg x \vee \neg y)$

  Function that maps literals to reals between 0 and 1, e.g.,

  $x \mapsto 0.4$, $\neg x \mapsto 0.6$, $y \mapsto 0.7$, $\neg y \mapsto 0.3$
- Weight of an assignment α is the product over the weights of its literals, i.e., $w(\alpha) := \Pi_{v \in \alpha^{-1}(1)} w(v) \cdot \Pi_{v \in \alpha^{-1}(0)} w(\neg v)$,

  e.g., $\alpha(x) = 1$, $\alpha(y) = 0 \Rightarrow w(\alpha) = 0.4 \cdot 0.3 = 0.16$
- Weighted model count (WMC) of formula is the sum of weights over all its satisfying assignments, e.g.,

  $w(F) = w(\alpha_1) + w(\alpha_2) = 0.12 + 0.42 = 0.54$

The Problem of Interest

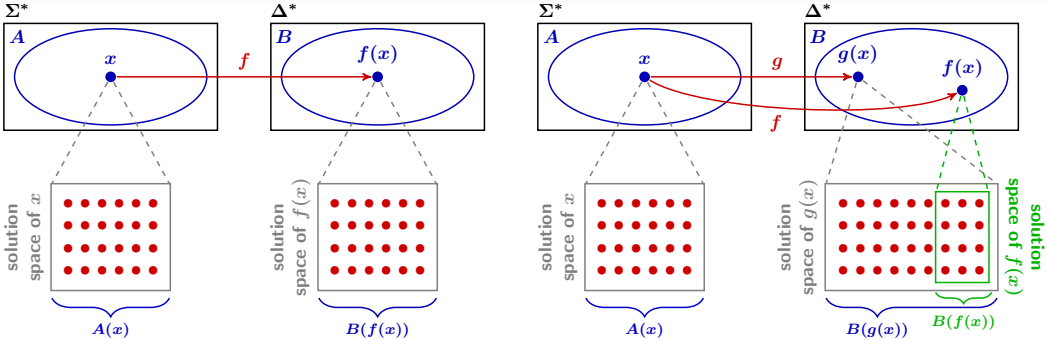
#SAT (Number SAT)

Given: Propositional formula F.
Output: Number of truth assignments τ such that the formula F_τ under the truth assignment evaluates to 1 (satisfiable).

WMC (Number SAT)

Given: Propositional formula F.
Output: Number of truth assignments τ such that the formula F_τ under the truth assignment evaluates to 1 (satisfiable).

Counting Complexity



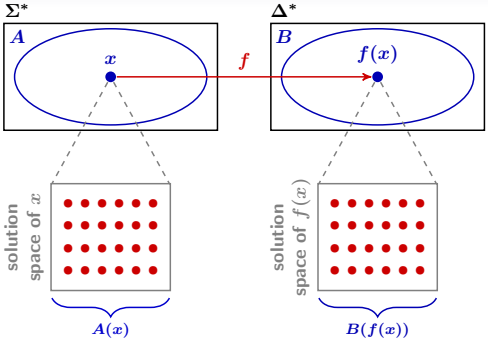
parsimonious reductions maintain cardinalities of the solution spaces, so $|A(x)| = |B(f(x))|$

subtractive reductions recalculate cardinalities: $|A(x)| = |B(g(x))| - |B(f(x))|$

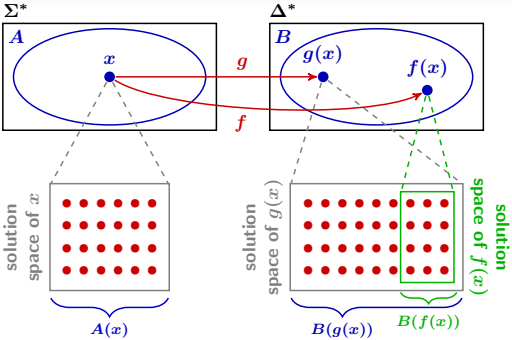
Now we talk about problems that output a solution (recall: NP was about decision problems).

1. #P: class of problems that count the number of accepting

Counting Complexity



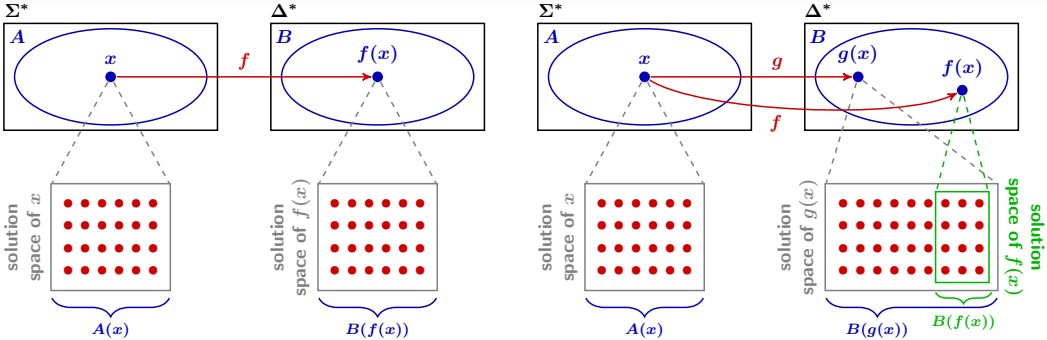
parsimonious reductions maintain cardinalities of the solution spaces, so $|A(x)| = |B(f(x))|$



subtractive reductions recalculate cardinalities: $|A(x)| = |B(g(x))| - |B(f(x))|$

1. #P: class of problems that count the number of accepting paths of a polynomial-time non-deterministic Turing machine (NP)
2. #P-hard: problem has a Turing reduction or parsimonious reduction, which is more detailed (subtractive reductions often helpful for problems beyond #P)

Counting Complexity



parsimonious reductions maintain cardinalities of the solution spaces, so $|A(x)| = |B(f(x))|$

subtractive reductions recalculate cardinalities: $|A(x)| = |B(g(x))| - |B(f(x))|$

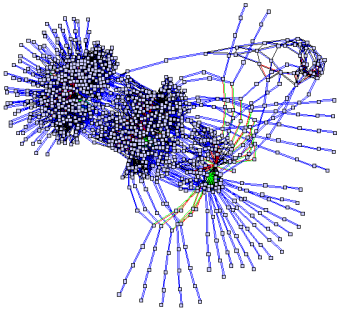
1. #P: class of problems that count the number of accepting paths of a polynomial-time non-deterministic Turing machine (NP)
2. #P-hard: problem has a Turing reduction or parsimonious reduction, which is more detailed (subtractive reductions often helpful for problems beyond #P)

Motivation: What's the issue?

Theory:
SAT cannot be solved faster than $2^{o(n)}$ steps! (ETH)

Idea:

- Practical instances are usually highly structured
- Structure can be exploited by algorithms



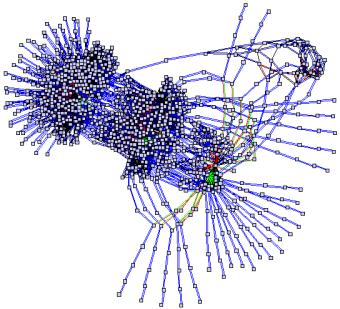
$\Rightarrow$ Various Approaches in Solvers to exploit Structure

Motivation: What's the issue?

Theory:
SAT cannot be solved faster than $2^{o(n)}$ steps! (ETH)

Idea:

- Practical instances are usually highly structured
- Structure can be exploited by algorithms



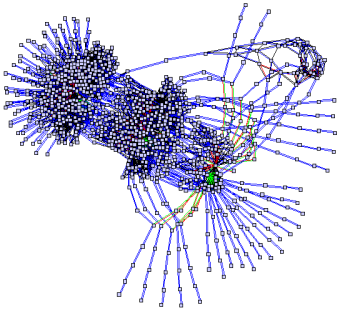
$\Rightarrow$ Various Approaches in Solvers to exploit Structure

Motivation: What's the issue?

Theory:
SAT cannot be solved faster than $2^{o(n)}$ steps! (ETH)

Idea:

- Practical instances are usually highly structured
- Structure can be exploited by algorithms



$\Rightarrow$ Various Approaches in Solvers to exploit Structure

Motivation: A somewhat different approach.

#SAT/WMC Solving

- There are already various solvers based on various techniques: approximate (Meel) / CDCL (Baccus/Thurley) / knowledge compilation based (Darwiche et al.)

Parameterized Algorithms

- Lots a theoretical work over last 20 years and various algorithms for #SAT

Research Question

Are (theoretical) algorithms from parameterized complexity even useful for implementations in #SAT/WMC solving?

Motivation: A somewhat different approach.

#SAT/WMC Solving

- There are already various solvers based on various techniques:
  approximate (Meel) / CDCL (Baccus/Thurley) /
  knowledge compilation based (Darwiche et al.)

Parameterized Algorithms

- Lots a theoretical work over last 20 years and various algorithms for #SAT

Research Question

Are (theoretical) algorithms from parameterized complexity even useful for implementations in #SAT/WMC solving?

Parameterized Algorithmics

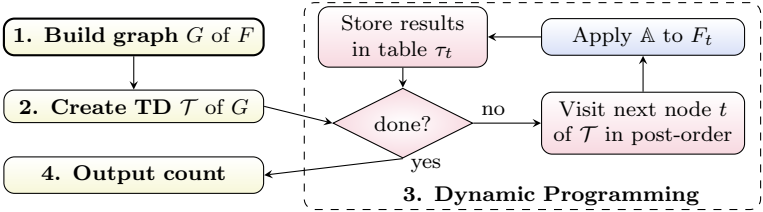
Topic of the Talk

Solve #SAT/WMC by means of an implementation of a parameterized algorithm that exploits small treewidth.

Propositional Model Counting

Outline

Outline



Part:

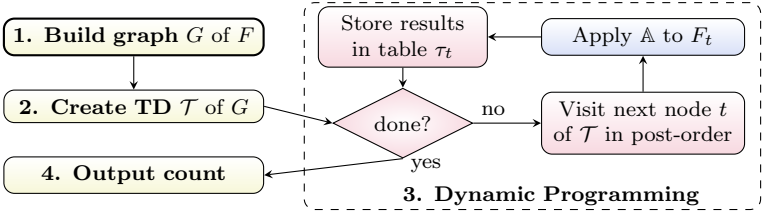
A) Background & Basic Concepts

Treewidth, Graph Representation (1) + Dynamic Programming (3) [Samer & Szeider JDA'10]

B) Finding TDs (2)

C) Dynamic Programming (3) on the GPU

Outline



Part:

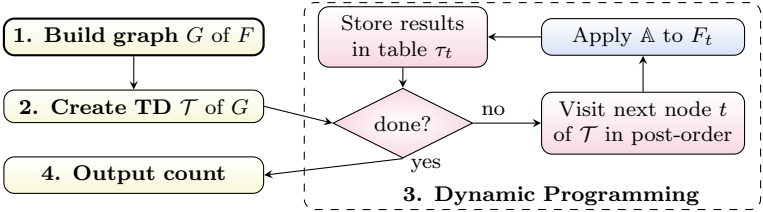
A) Background & Basic Concepts

Treewidth, Graph Representation (1) + Dynamic Programming (3) [Samer & Szeider JDA'10]

B) Finding TDs (2)

C) Dynamic Programming (3) on the GPU

Outline



Part:

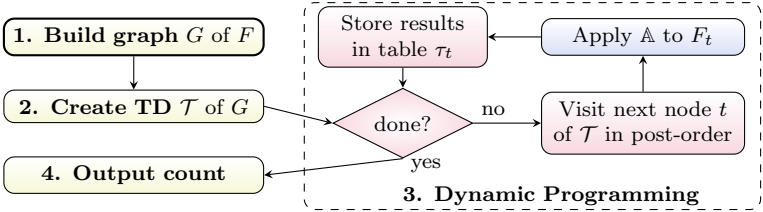
A) Background & Basic Concepts

Treewidth, Graph Representation (1) + Dynamic Programming (3) [Samer & Szeider JDA'10]

B) Finding TDs (2)

C) Dynamic Programming (3) on the GPU

Outline



Part:

A) Background & Basic Concepts

Treewidth, Graph Representation (1) + Dynamic Programming (3) [Samer & Szeider JDA'10]

B) Finding TDs (2)

C) Dynamic Programming (3) on the GPU

Propositional Model Counting

Use TDs to solve #SAT/WMC

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

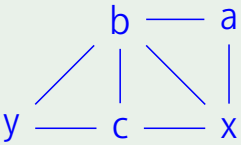
$$\mathbf{Mod}(\varphi) = \{ \{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\} \}$$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions

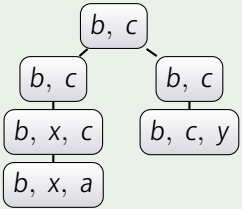


"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

LEAF.: Put empty set and counter 1

INTR.: Guess truth value and check satisfiability

REMOVE: Remove a from each assignment (row) in the table and sum up the counters if we get multiple assignments with the same data

JOIN: Match rows with the same assignment and multiply the counters

Algorithm for Primal Graph

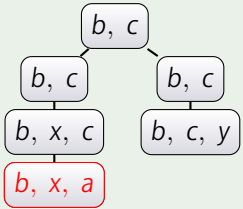
In: Node t, bag χ_t, clauses F_t, sequence C of tables.
Out: Table tab_t

if $\mathrm{type}(t) = \mathrm{leaf}$ **then**
  $\mathrm{tab}_t \leftarrow \{\emptyset\}$
else if $\mathrm{type}(t) = \mathrm{intr}$ *and* $a \in \chi_t \setminus \chi_{t'}$, **then**
  $\mathrm{tab}_t \leftarrow \left\{\tau \cup \{a\} \quad \middle| \ \tau \in \mathrm{tab}'', \tau \cup \{a\} \models F_t\right\} \cup$
  $\left\{\tau \quad \middle| \ \tau \in \mathrm{tab}'', \tau \models F_t\right\}$
else if $\mathrm{type}(t) = \mathrm{rem}$ *and* $a \in \chi_{t'} \setminus \chi_t$ **then**
  $\mathrm{tab}_t \leftarrow \left\{\tau \setminus \{a\} \quad \middle| \ \tau \in \mathrm{tab}''\right\}$
else if $\mathrm{type}(t) = \mathrm{join}$ **then**
  $\mathrm{tab}_t \leftarrow \left\{\tau \quad \middle| \ \tau \in \mathrm{tab}'', \tau \in \mathrm{tab}''\right\}$
return tab_t

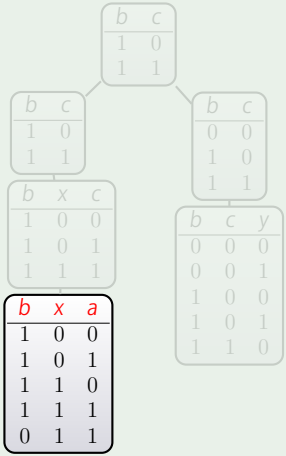
Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



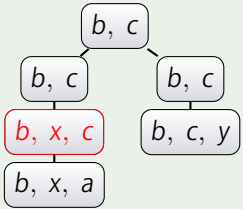
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

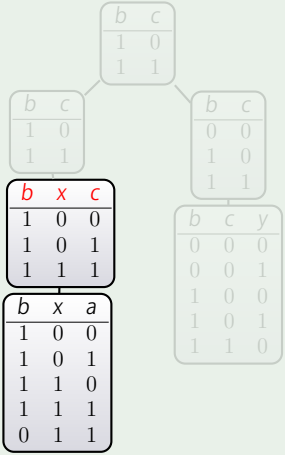


Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions





b	c
1	0
1	1

b	c
1	0
1	1

b	c
0	0
1	0
1	1

b	x	c
1	0	0
1	0	1
1	1	1

b	c	y
0	0	0
0	0	1
1	0	0
1	0	1
1	1	0

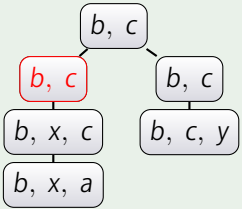
b	x	a
1	0	0
1	0	1
1	1	0
1	1	1
0	1	1

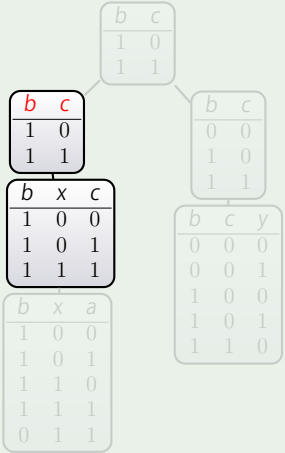
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



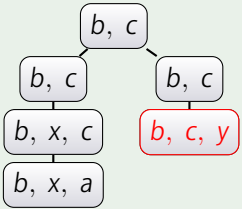


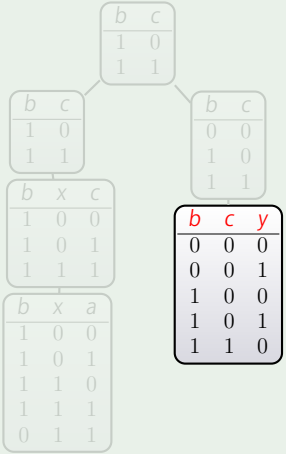
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



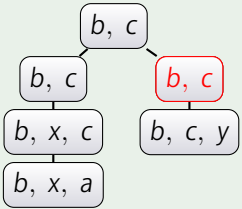


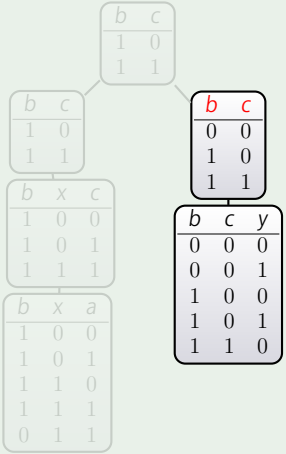
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



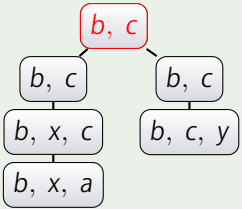


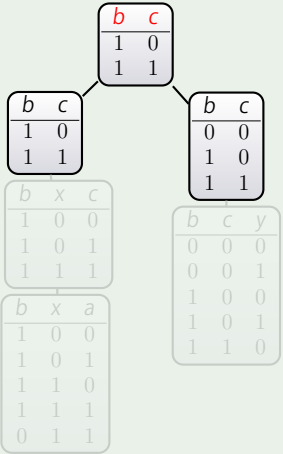
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



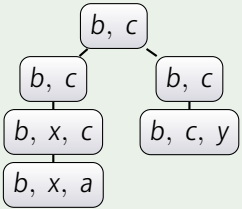


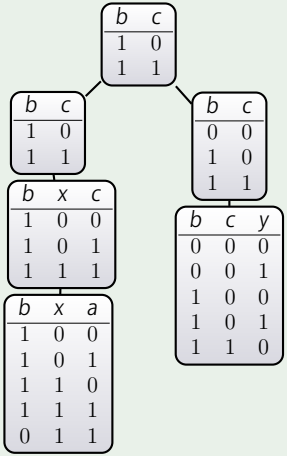
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



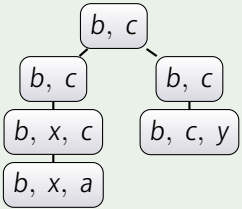


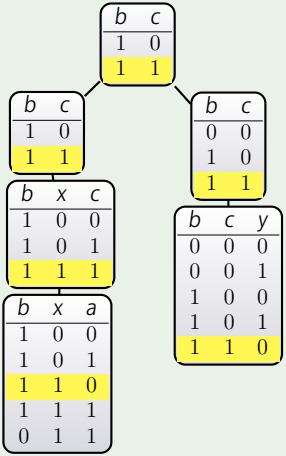
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



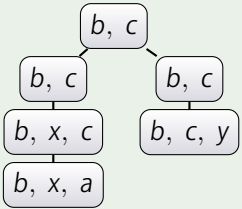


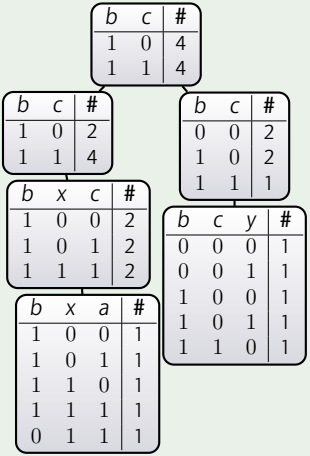
"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions





b	c	#
1	0	4
1	1	4

b	c	#
1	0	2
1	1	4

b	c	#
0	0	2
1	0	2
1	1	1

b	x	c	#
1	0	0	2
1	0	1	2
1	1	1	2

b	c	y	#
0	0	0	1
0	0	1	1
1	0	0	1
1	0	1	1
1	1	0	1

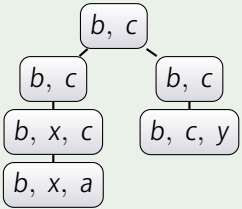
b	x	a	#
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1
0	1	1	1

"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Solving #SAT [SamerSzeider10]

$\varphi = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Create graph representation
2. Decompose graph
3. Solve problems via $\mathcal{S}$
4. Combine solutions



b	c	#
1	0	4
1	1	4

b	c	#
1	0	2
1	1	4

b	c	#
0	0	2
1	0	2
1	1	1

b	x	c	#
1	0	0	2
1	0	1	2
1	1	1	2

b	c	y	#
0	0	0	1
0	0	1	1
1	0	0	1
1	0	1	1
1	1	0	1

b	x	a	#
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1
0	1	1	1

"Local formula" F_t clauses whose variables are contained in the bag (colored in red above)

Runtime: $2^{O(tw)} \cdot \text{poly}(|\varphi|)$

“Find” tree decompositions of small width?

Works well even for relatively large instances.

Thanks to the Parameterized Algorithms and Computational Experiments Challenge (PACE) ’16/’17!!!

“Find” tree decompositions of small width?

Works well even for relatively large instances.

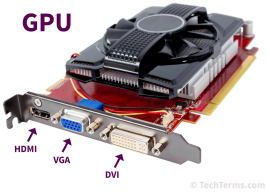
Thanks to the Parameterized Algorithms and Computational Experiments Challenge (PACE) ’16/’17!!!

Propositional Model Counting

DP goes Parallel

A GPU-based #SAT/WMC-solver

OR how to go parallel?



Dynamic Programming on the GPU

How to parallelize DP?

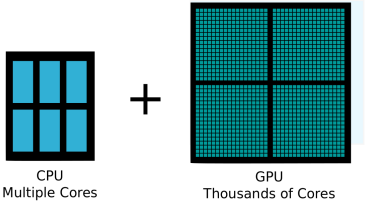
1. Compute tables for multiple nodes in parallel

⇒ Does not allow for immediate massive parallelization due to dependencies to children

2. Distribute computation of rows among different computation units

⇒ Allows with right hindsight for massive parallelization

Why: computation of rows are independent



Dynamic Programming on the GPU

How to parallelize DP?

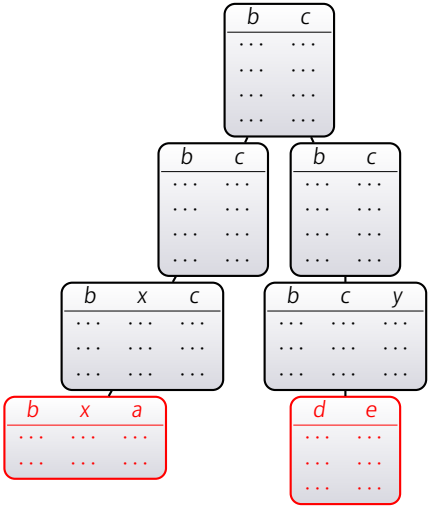
1. Compute tables for multiple nodes in parallel

⇒ Does not allow for immediate massive parallelization due to dependencies to children

2. Distribute computation of rows among different computation units

⇒ Allows with right hindsight for massive parallelization

Why: computation of rows are independent



Dynamic Programming on the GPU

How to parallelize DP?

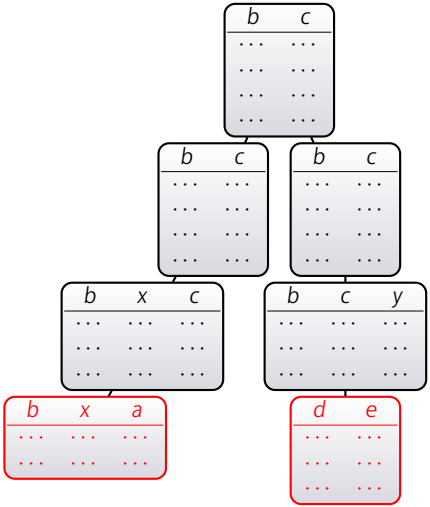
1. Compute tables for multiple nodes in parallel

⇒ Does not allow for immediate massive parallelization due to dependencies to children

2. Distribute computation of rows among different computation units

⇒ Allows with right hindsight for massive parallelization

Why: computation of rows are independent



Dynamic Programming on the GPU

How to parallelize DP?

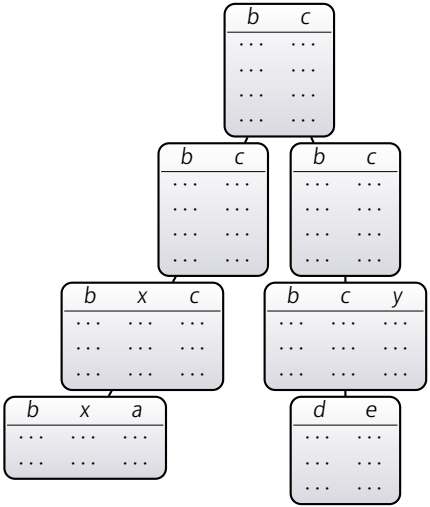
1. Compute tables for multiple nodes in parallel

⇒ Does not allow for immediate massive parallelization due to dependencies to children

2. Distribute computation of rows among different computation units

⇒ Allows with right hindsight for massive parallelization

Why: computation of rows are independent



Dynamic Programming on the GPU

How to parallelize DP?

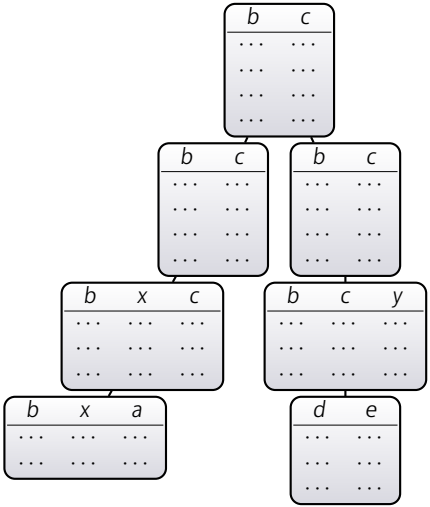
1. Compute tables for multiple nodes in parallel

⇒ Does not allow for immediate massive parallelization due to dependencies to children

2. Distribute computation of rows among different computation units

⇒ Allows with right hindsight for massive parallelization

Why: computation of rows are independent



Implementation (gpuSAT1 [FHWZ18])

Right hindsight?

- Data structures: a "pixel" represents #solutions (store data as array)
- Table merging: merge small bags (< 14)
- Table splitting: split large tables
- Weight: avoid double overflows by factor for #SAT
  (so it's actually WMC with uniform weights).

Implementation

- OpenCL: vendor and hardware independent computation framework; C++11
- Works for two graph types: primal and incidence graph
- Supports weighted model counting (WMC)

Propositional Model Counting

Some Experiments

Experimental Work (Naive Implementation)

Instances

- 2585 instances from public benchmarks
- #SAT and WMC

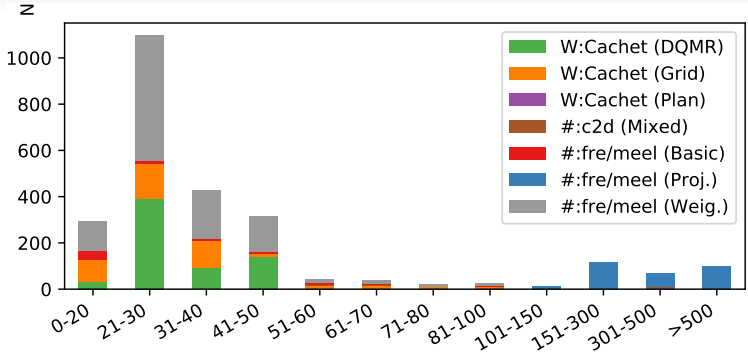
Limits

- Cannot expect to solve instances of high treewidth.

Hardware

- non-GPU solving: cluster of 9 nodes; each E5-2650 CPUs(12cores) 2.2 GHz, 256 GB RAM; disabled HT, kernel 4.4
- GPU-solving: i3-3245 3.4 GHz; 16 GB RAM; GPU: Sapphire Pulse ITX Radeon RX 570 GPU; 1.24 GHz with 32 compute units, 2048 shader units, 4GB VRAM

Distribution of Primal Width



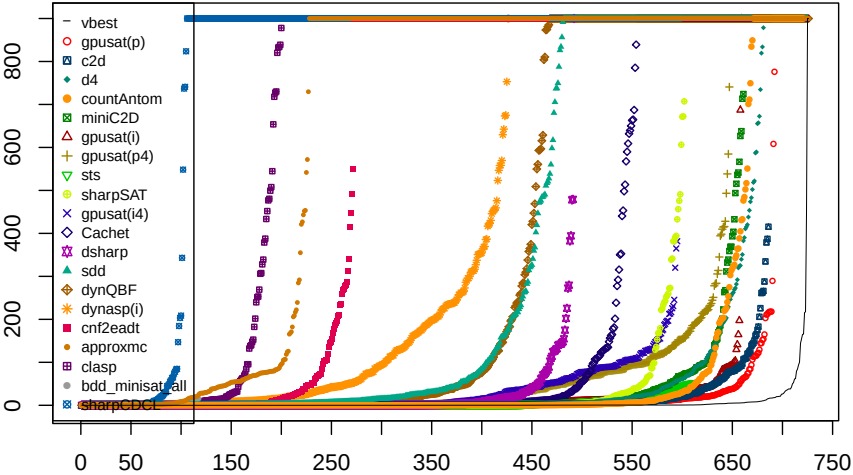
Decomposition Heuristic:

- Runtime well below a second (max. 2.5) 0–40
- Timeout (900s) on 41 instances

$\Rightarrow$ 54% primal treewidth below 30; 70% below 40

Parameterized Algorithms might work...

Solving (Width: 0–30): #SAT



kc/cdcl: c2d, d4, dsharp
dp: gpusat, dynQBF, dynasp
parallel: countAntom, gpusat
cdcl: Cachet, sharpSAT, clasp
bdd: sdd
approx: approxmc, sts

Solving: #SAT

solver	0-20	21-30	31-40	41-50	51-60	>60	best	$\sum$	rank
c2d	164	519	**175**	116	20	118	120	1112	2
Cachet	133	421	91	109	8	58	13	820	7
d4	**169**	510	156	119	**23**	**162**	191	**1139**	1
gpusat(p)	**169**	**523**	79	104	0	0	88	875	6
miniC2D	167	491	137	103	8	67	2	973	4
sharpSAT	136	465	136	112	11	124	**483**	984	3
sts	162	448	101	**146**	10	45	252	912	5

Table: Number of counting instances solved by solver and interval.

Empirical Work (first approach)

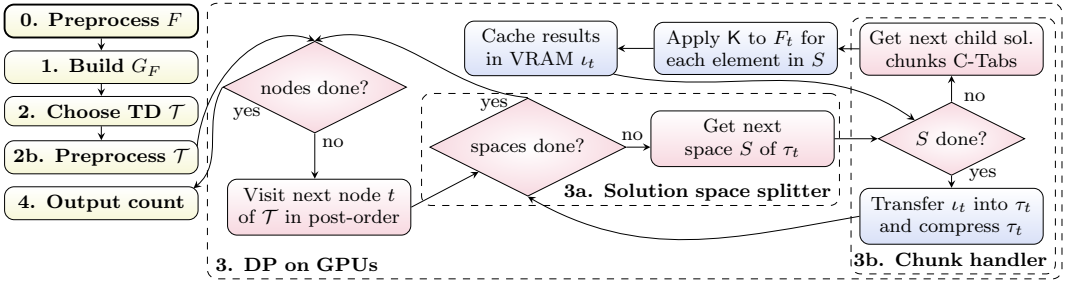
Observations

- Implementation is fairly naive
- Still: competitive up to width 30
- Requirement: obtain decompositions fast
- Width was surprisingly small (different for SAT)

Propositional Model Counting

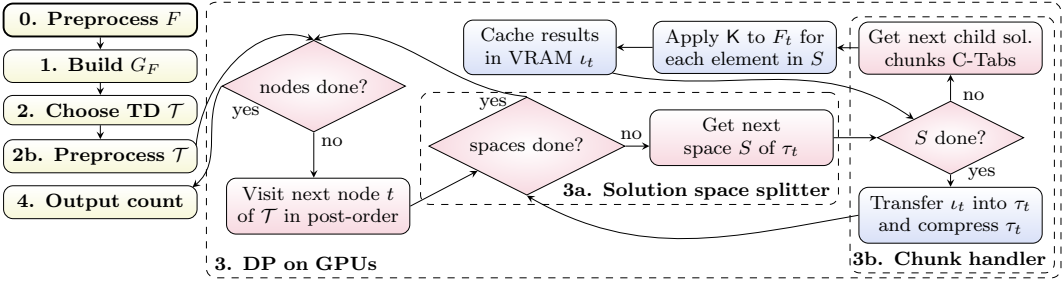
New Architecture

New Architecture [FHZ19]



0. Instance Preprocessing
2. Customized Tree Decompositions
3a. Solution Space Splitting
3b. Execute a small GPU-program in a GPU thread (kernel) for each element in S
Compress the data and store it in the VRAM (separate GPU-programs)
After all chunks are processed memory regions are merged

New Architecture [FHZ19]

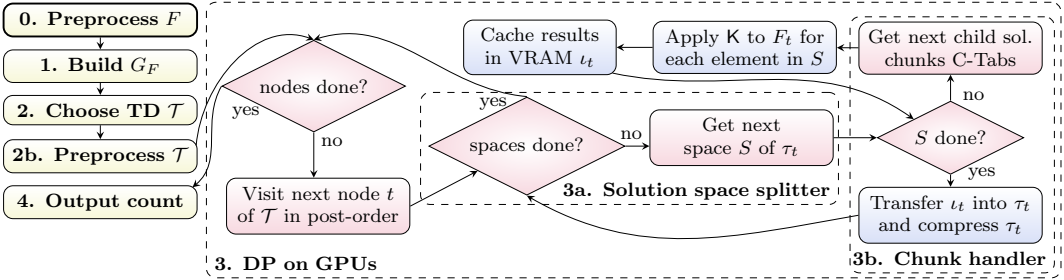


0. Instance Preprocessing
2. Customized Tree Decompositions
   (#30; minimize max. card. of intersection of bags at node and its children)

3a. Solution Space Splitting

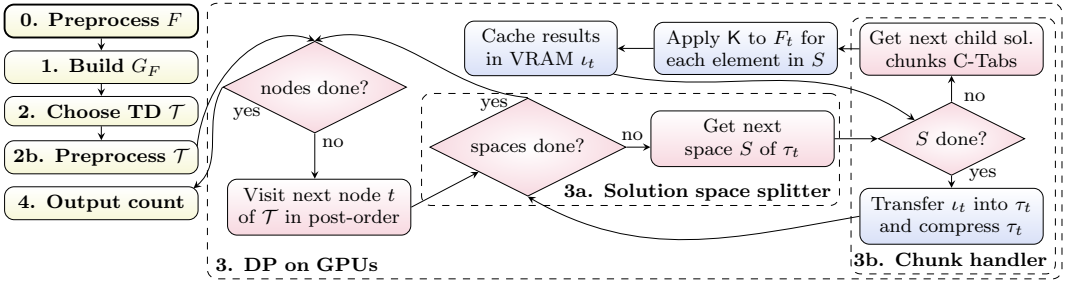
3b. Execute a small GPU-program in a GPU thread (kernel) for each element in S
   Compress the data and store it in the VRAM (separate GPU-programs)
   After all chunks are processed memory regions are merged

New Architecture [FHZ19]



0. Instance Preprocessing
2. Customized Tree Decompositions
3a. Solution Space Splitting
(Split larger solutions into smaller portions ⇒ avoid OOM)
3b. Execute a small GPU-program in a GPU thread (kernel) for each element in S
Compress the data and store it in the VRAM (separate GPU-programs)
After all chunks are processed memory regions are merged

New Architecture [FHZ19]



- 0. Instance Preprocessing
- 2. Customized Tree Decompositions
- 3a. Solution Space Splitting
- 3b. Execute a small GPU-program in a GPU thread (kernel) for each element in S
  Compress the data and store it in the VRAM (separate GPU-programs)
  After all chunks are processed memory regions are merged

Data Structures (Save Space)

Disclaimer for theorists: you need to get your hands dirty
(essentially: bit fiddling)

Data Structures (Save Space)

1. Store compressed partial assignments
   (only where # $\neq 0$, simulate a BST in an array)
2. Use logarithmic counters

Data Structures (Save Space)

1. Store compressed partial assignments
   (only where # $\neq 0$, simulate a BST in an array)
2. Use logarithmic counters

1) Assignment Compression (BST in an Array)

- Continuous sequence 64-bit unsigned integers (cells)
- Cell: empty, index, and value (counter)
- index cells: lower 32 bits index to the next cell (variable $\rightarrow$ 0), upper bits (1)
- Handle Sync by keeping track of the current size
  (number of allocated cells; prevent to allocate cell again)

Data Structures (Save Space)

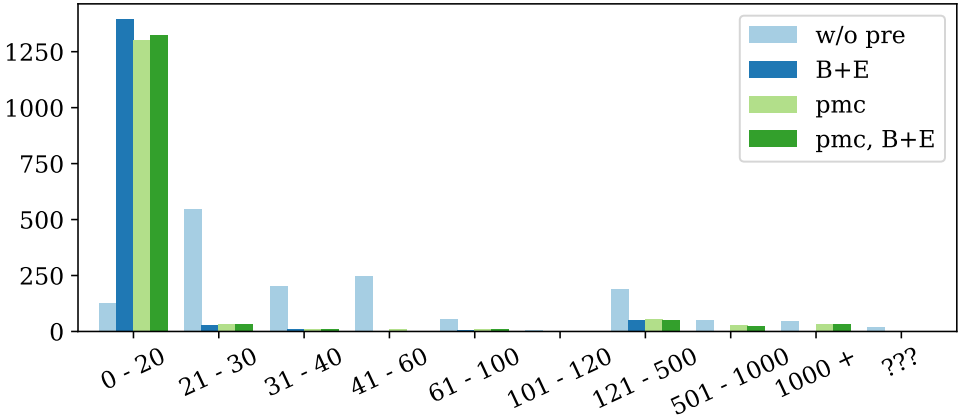
1. Store compressed partial assignments
   (only where # $\neq 0$, simulate a BST in an array)
2. Use logarithmic counters

2) Data Type Precision

- Store floating log-counters
- Numbers stored in relation to exponent 2^e (largest exponent)
- Dynamically change exponent (keep highest possible precision)

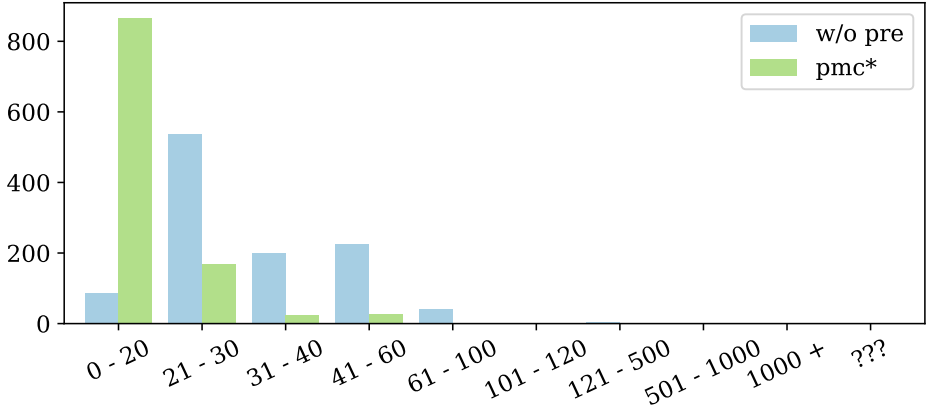
Where are we at with the new architecture?

#SAT: Width Comparison (w/o Preprocessing)



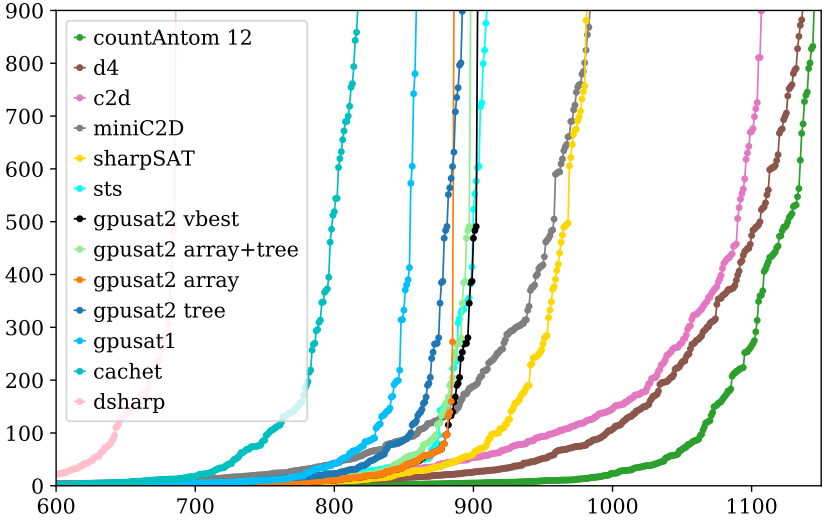
⇒ Produce decompositions of significantly smaller width

WMC: Width Comparison (w/o Preprocessing)

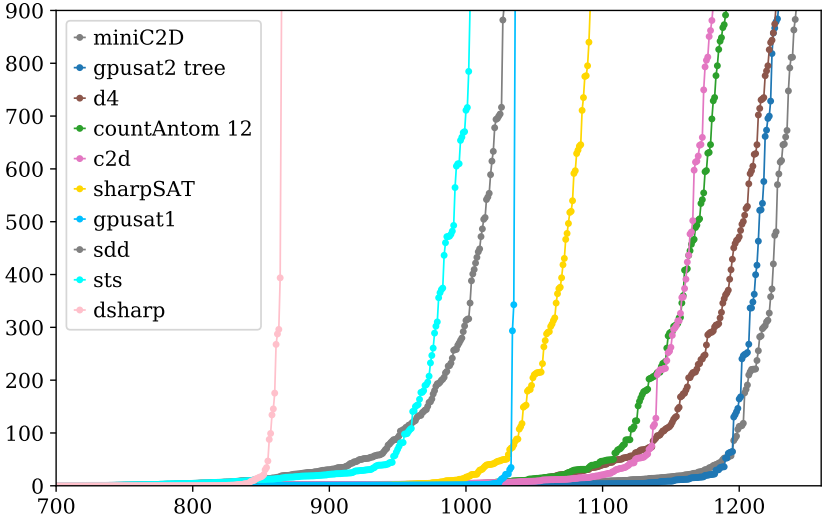


⇒ Produce decompositions of significantly smaller width

#SAT: Runtime Results (wo. Preprocessing)



#SAT: Runtime Results (w. Preprocessing)



⇒ Techniques pay off after preprocessing

Summary

Contributions

- Established Architecture for DP on the GPU
- Competitive Implementation for #SAT/WMC solving

Benchmark: Comparing apples and oranges

BUT: you compare parallel and sequential solvers.

1. We run on cheap consumer hardware (200 EUR).
2. Cannot measure speedup due to OpenCL limitations
   $\Rightarrow$ migrate to cuda

Summary contd.

Take Home Messages

1. Parameterized Algorithms can actually work
   (Preprocessing is key; some techniques pay only off with right preprocessing)
2. Does it work for SAT? $\Rightarrow$ we don't expect so.

Future Work

- Improve current setup by:
  Portfolio solving; Parallel Usage of GPUs; Alternative Frameworks
- Parameters (pswidth)

Summary contd.

Take Home Messages

1. Parameterized Algorithms can actually work
   (Preprocessing is key; some techniques pay only off with right preprocessing)
2. Does it work for SAT? $\Rightarrow$ we don't expect so.

Future Work

- Improve current setup by:
  Portfolio solving; Parallel Usage of GPUs; Alternative Frameworks
- Parameters (pswidth)

Advertisement:
PACE-2020 (?)
GitHub:daajoe/{benchmark-tool,fhtd,trellis}

Projected Model Counting

It gets slightly combinatorial now.
(Following are theoretical results.)

Projected Model Counting

Motivation

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

How many models?

$$|\mathbf{Mod}(F)| = |\{\{b\}, \{a, b\}, \{b, c\}, \{a, b, c\}, \{b, c, x\}, \{a, b, c, x\}, \{b, y\}, \{a, b, y\}\}|$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

How many models?

$$|\mathbf{Mod}(F)| = |\{\{b\}, \{a, b\}, \{b, c\}, \{a, b, c\}, \{b, c, x\}, \{a, b, c, x\}, \{b, y\}, \{a, b, y\}\}|$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

How many models?

$$|\mathbf{Mod}(F)| = |\{\{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\}\}| = 8$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

How many models?

$$|\mathbf{Mod}(F)| = |\{\{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\}\}| = 8$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$|\mathbf{Mod}(F)| = |\{\{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\}\}| = 8$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$|\mathbf{Mod}(F)_{|\{x,y\}|}| = |\{\{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\}\}|$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$|\mathbf{Mod}(F)_{|\{x,y\}|}| = |\{\{b\}, \{a,b\}, \{b,c\}, \{a,b,c\}, \{b,c,x\}, \{a,b,c,x\}, \{b,y\}, \{a,b,y\}\}|$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$\mathbf{Mod}(F)_{|\{x,y\}|} = \{\{\}, \{\}, \{\}, \{\}, \{x\}, \{x\}, \{y\}, \{y\}\}$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$\mathbf{Mod}(F)_{|\{x,y\}|} = \{\{\}, \{x\}, \{y\}\} \ = 3$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$\textbf{Mod}(F)_{|\{x,y\}|} = \{\{\}, \{x\}, \{y\}\} \ = 3$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

Counting where we are interested only in solutions wrt certain variables is called projected model counting.

From Model Counting to Projected Model Counting

Recall our Model Counting Example

$$F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$$

$$\mathbf{Mod}(F)_{|\{x,y\}|} = \{\{\}, \{x\}, \{y\}\} = 3$$

Hide: a, b, c Interested: x, y

Like "count * in (select x,y from solutions_to_F)"

Remove all variables other than x, y from the solutions.

Problem

Given: Formula F

Set $P \subseteq \mathbf{var}(F)$ of projection variables

Task: Compute $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

Note: Identical models (restricted to P) of F count as one

Classical Complexity

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of
occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Classical Complexity

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Classical Complexity

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Classical Complexity

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Parameterized Complexity (wpb Treewidth)

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Parameterized Complexity (wpb Treewidth)

SAT
$P = \emptyset$
NP-complete
[Cook'79]

PMC
P arbitrary
#·NP-complete
[DurandHK05]

#SAT
$P = \mathbf{var}(F)$
#-P-complete
[Valiant'79, Roth'96]

FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$
[BacchhusEtAl'03]

$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(|F|))$
d..maximum number of occurrences over all variables
[BacchhusEtAl'03,SamerSzeider10]

Parameterized Complexity (wpb Treewidth)

SAT	PMC	#SAT
$P = \emptyset$	P arbitrary	$P = \mathbf{var}(F)$
NP-complete	#·NP-complete	#-P-complete
[Cook'79]	[DurandHK05]	[Valiant'79, Roth'96]
FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$ [BacchhusEtAl'03]	?	$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(\lvert F \rvert))$ d..maximum number of occurrences over all variables [BacchhusEtAl'03,SamerSzeider10]

Parameterized Complexity (wpb Treewidth)

SAT	PMC	#SAT
$P = \emptyset$	P arbitrary	$P = \mathbf{var}(F)$
NP-complete	#·NP-complete	#-P-complete
[Cook'79]	[DurandHK05]	[Valiant'79, Roth'96]
FPT / Runtime: $\mathcal{O}(2^{\mathrm{tw}(F)})$ [BacchhusEtAl'03]	$2^{2^{\mathcal{O}(tw)}} \cdot \mathrm{poly}(\lvert F\rvert)$ Lower Bound (ETH): no algorithm for $2^{2^{o(tw)}} \cdot 2^{o(\lvert F\rvert)}$ [FichteHecherEtAl'18]	$\mathcal{O}(2^{\mathrm{tw}(F)} \cdot \mathrm{tw}(F) \cdot d \cdot \mathrm{poly}(\lvert F\rvert))$ d..maximum number of occurrences over all variables [BacchhusEtAl'03,SamerSzeider10]

Purpose of this Chapter

1. Get familiar with Projected Model Counting (PMC)
2. Show how we can solve PMC wpb treewidth
3. Explain the Lower Bound and Implications

Based on latest joint research [FHecherPfandler'19], [FHecherMorakWoltran'18].

Projected Model Counting

Lower Bound

Solving PMC using $\mathcal{S}$?

Let $\mathcal{DP}_S$ be algorithm to solve #SAT
where $\mathcal{S}$ "local" algorithm used to compute the tables

Can we trivially extend $\mathcal{S}$ to solve PMC?

Short answer: NO!

Theorem

Unless ETH fails, there is no algorithm for PMC running in time $2^{2^{o(tw)}} \cdot \text{poly}(n)$ where n is the number of variables of F.

Proof.

- Solve closed QBF $\forall X.\exists Y.F$ by checking whether $2^{|X|}$ solves PMC of $(F, P = X)$
- Unless ETH fails, 2-QSAT can not be solved [LampisM17] in time $2^{2^{o(tw)}} \cdot 2^{o(n)}$ □

Solving PMC using $\mathcal{S}$?

Let $\mathcal{DP}_S$ be algorithm to solve #SAT
where $\mathcal{S}$ "local" algorithm used to compute the tables

Can we trivially extend $\mathcal{S}$ to solve PMC?

Short answer: NO!

Theorem

Unless ETH fails, there is no algorithm for PMC running in time $2^{2^{o(tw)}} \cdot \text{poly}(n)$ where n is the number of variables of F.

Proof.

- Solve closed QBF $\forall X.\exists Y.F$ by checking whether $2^{|X|}$ solves PMC of $(F, P = X)$
- Unless ETH fails, 2-QSAT can not be solved [LampisM17] in time $2^{2^{o(tw)}} \cdot 2^{o(n)}$ □

Solving PMC using $\mathcal{S}$?

Let $\mathcal{DP}_S$ be algorithm to solve #SAT
where $\mathcal{S}$ "local" algorithm used to compute the tables

Can we trivially extend $\mathcal{S}$ to solve PMC?

Short answer: NO!

Theorem

Unless ETH fails, there is no algorithm for PMC running in time $2^{2^{o(tw)}} \cdot \text{poly}(n)$ where n is the number of variables of F.

Proof.

- Solve closed QBF $\forall X.\exists Y.F$ by checking whether $2^{|X|}$ solves PMC of $(F, P = X)$
- Unless ETH fails, 2-QSAT can not be solved [LampisM17] in time $2^{2^{o(tw)}} \cdot 2^{o(n)}$ □

Solving PMC using $\mathcal{S}$?

Let $\mathcal{DP}_S$ be algorithm to solve #SAT
where $\mathcal{S}$ "local" algorithm used to compute the tables

Can we trivially extend $\mathcal{S}$ to solve PMC?

Short answer: NO!

Theorem

Unless ETH fails, there is no algorithm for PMC running in time $2^{2^{o(tw)}} \cdot \text{poly}(n)$ where n is the number of variables of F.

Proof.

- Solve closed QBF $\forall X.\exists Y.F$ by checking whether $2^{|X|}$ solves PMC of $(F, P = X)$
- Unless ETH fails, 2-QSAT can not be solved [LampisM17] in time $2^{2^{o(tw)}} \cdot 2^{o(n)}$ □

Solving PMC using $\mathcal{S}$?

Let $\mathcal{DP}_S$ be algorithm to solve #SAT
where $\mathcal{S}$ "local" algorithm used to compute the tables

Can we trivially extend $\mathcal{S}$ to solve PMC?

Short answer: NO!

Theorem

Unless ETH fails, there is no algorithm for PMC running in time $2^{2^{o(tw)}} \cdot \text{poly}(n)$ where n is the number of variables of F.

Proof.

- Solve closed QBF $\forall X.\exists Y.F$ by checking whether $2^{|X|}$ solves PMC of $(F, P = X)$
- Unless ETH fails, 2-QSAT can not be solved [LampisM17] in time $2^{2^{o(tw)}} \cdot 2^{o(n)}$

$\square$

Quick Excursion (Notations)

Asymptotic Notation: Small o (Small oh)

Let f, g be functions $f, g : \mathbb{N} \to \mathbb{N}$.
$f \in o(g)$: means f is dominated by g asymptotically

formally, $\forall k > 0 : \exists n_0 : \forall n > n_0 : f(n) < k \cdot g(n)$

ETH (Exponential Time Hypothesis)

- Lower bound (Assumption) on how fast SAT can be solved
- ETH states that there is some real $s > 0$ such that we cannot decide satisfiability of a given 3-CNF formula in time $2^{s \cdot n} \cdot (n + m)^{\mathcal{O}(1)}$ where n is the number of variables and m number of clauses
- Assuming ETH (and a Theorem by Dell et al.).

  Implication: one cannot solve 3-SAT in time $2^{o(n)}$

Quantified Boolean Formula (QBF)

QBF instance $\mathcal{I}$ of the form

$$Q = Q_1 X_1 Q_2 X_2 \ldots Q_k X_k . F$$

- $Q_i \in \{\exists, \forall\}$ quantifier prefix
- X_i quantified variables (bounded by exactly one quantifier)
- F is a formula in CNF (same as for SAT) q

Depth: number of quantifier alternations

Question (QSAT): is the formula valid

QSAT

$\mathcal{I} = Q.\psi$ with

- $Q = \exists ab\ \forall cd\ \exists ef$
- $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$

$\mathcal{I}$ is satisfiable (for $a \mapsto 1$, $b \mapsto 0$).

Quantified Boolean Formula (QBF)

QBF instance $\mathcal{I}$ of the form

$$Q = Q_1 X_1 Q_2 X_2 \ldots Q_k X_k . F$$

- $Q_i \in \{\exists, \forall\}$ quantifier prefix
- X_i quantified variables (bounded by exactly one quantifier)
- F is a formula in CNF (same as for SAT) q

Depth: number of quantifier alternations
Question (QSAT): is the formula valid

QSAT

$\mathcal{I} = Q.\psi$ with

- $Q = \exists ab\ \forall cd\ \exists ef$
- $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$

$\mathcal{I}$ is satisfiable (for $a \mapsto 1$, $b \mapsto 0$).

Quantified Boolean Formula (QBF)

QBF instance $\mathcal{I}$ of the form

$$Q = Q_1 X_1 Q_2 X_2 \ldots Q_k X_k . F$$

- $Q_i \in \{\exists, \forall\}$ quantifier prefix
- X_i quantified variables (bounded by exactly one quantifier)
- F is a formula in CNF (same as for SAT) q

Depth: number of quantifier alternations
Question (QSAT): is the formula valid

QSAT

$\mathcal{I} = Q.\psi$ with

- $Q = \exists ab\ \forall cd\ \exists ef$
- $\psi = (a \vee c \vee e) \wedge (\neg b \vee d) \wedge (e \vee f) \wedge (c \vee \neg e) \wedge (\neg d \vee f)$

$\mathcal{I}$ is satisfiable (for $a \mapsto 1$, $b \mapsto 0$).

Side Question: Does it get simpler eventually?

$\#\Sigma_\ell$: "PMC" for QBFs

- Given: QBF $\psi = \exists X_1.\forall X_2.\ldots QX_\ell.F$, where set P is the set of free variables of ψ
- Task: Compute $\mathrm{PMCQ}_P(\psi) := |\{I \mid I \subseteq P, \psi[I] \text{ valid}\}|$

Theorem [FHecherPfandler19]

Unless ETH fails, there is no algorithm for $\#\Sigma_\ell$ running in time $\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$.

Proof.

- Argument similar to before, boils down to:
- Unless ETH fails, $(\ell+1)$-QSAT can not be solved [FHecherPfandler19] in time

$$\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$$

□

Side Question: Does it get simpler eventually?

#Σ_ℓ: "PMC" for QBFs

- Given: QBF $\psi = \exists X_1.\forall X_2.\ldots QX_\ell.F$, where set P is the set of free variables of ψ
- Task: Compute $\mathrm{PMCQ}_P(\psi) := |\{I \mid I \subseteq P, \psi[I] \text{ valid}\}|$

Theorem [FHecherPfandler19]

Unless ETH fails, there is no algorithm for #Σ_ℓ running in time $\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$.

Proof.

- Argument similar to before, boils down to:
- Unless ETH fails, $(\ell+1)$-QSAT can not be solved [FHecherPfandler19] in time

$$\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$$

□

Side Question: Does it get simpler eventually?

#Σ_ℓ: "PMC" for QBFs

- Given: QBF $\psi = \exists X_1.\forall X_2.\ldots QX_\ell.F$, where set P is the set of free variables of ψ
- Task: Compute $\mathrm{PMCQ}_P(\psi) := |\{I \mid I \subseteq P, \psi[I] \text{ valid}\}|$

Theorem [FHecherPfandler19]

Unless ETH fails, there is no algorithm for #Σ_ℓ running in time $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{o(tw)}}}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$.

Proof.

- Argument similar to before, boils down to:
- Unless ETH fails, $(\ell+1)$-QSAT can not be solved [FHecherPfandler19] in time

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{o(tw)}}}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$$

□

Side Question: Does it get simpler eventually?

#Σ_ℓ: "PMC" for QBFs

- Given: QBF $\psi = \exists X_1.\forall X_2.\ldots QX_\ell.F$, where set P is the set of free variables of ψ
- Task: Compute $\mathrm{PMCQ}_P(\psi) := |\{I \mid I \subseteq P, \psi[I] \text{ valid}\}|$

Theorem [FHecherPfandler19]

Unless ETH fails, there is no algorithm for #Σ_ℓ running in time $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{o(tw)}}}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$.

Proof.

- Argument similar to before, boils down to:
- Unless ETH fails, $(\ell+1)$-QSAT can not be solved [FHecherPfandler19] in time $\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^{o(tw)}}}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$

□

Side Question: Does it get simpler eventually?

$\#\Sigma_\ell$: "PMC" for QBFs

- Given: QBF $\psi = \exists X_1.\forall X_2.\ldots QX_\ell.F$, where set P is the set of free variables of ψ
- Task: Compute $\mathrm{PMCQ}_P(\psi) := |\{I \mid I \subseteq P, \psi[I] \text{ valid}\}|$

Theorem [FHecherPfandler19]

Unless ETH fails, there is no algorithm for $\#\Sigma_\ell$ running in time $\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$.

Proof.

- Argument similar to before, boils down to:
- Unless ETH fails, $(\ell+1)$-QSAT can not be solved [FHecherPfandler19] in time

$$\underbrace{2^{2^{\cdots^{2^{o(tw)}}}}}_{\text{height } \ell+1} \cdot 2^{o(|F|)}$$

□

Projected Model Counting

A DP-Algorithm to Solve PMC

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Solving PMC using $\mathcal{S}$?

Wanted

- $\mathrm{PMC}_P(F) := |\{I \cap P \mid I \subseteq \mathbf{var}(F), I \models F\}|$

$\rightsquigarrow$ "Directly" computing PMC_P: too expensive (exponentially many models)

Approach: Compute numbers "locally"

$\rightsquigarrow$ Still use algorithm from the previous Ch. to compute counts

PMC in three steps

1. Run algorithm $\mathcal{DP}_{\mathcal{S}}$ (seen before)
2. Purge non-solutions (go again over the tables and check top-down[a] whether we can construct a solution from the row; if not remove)
3. Projected counting via new algorithm $\mathcal{DP}_{\mathcal{P}}$

[a] pre-order traversal for each row

Step 3: $\rightsquigarrow$ Assign counter to a sets of rows

Dynamic Programming via Table algorithm $\mathcal{P}$ for PMC

Ingredients for given PMC instance (P, F)

- Equivalence Classes (Partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map set σ of rows to counter ipmc (intersection PMC)
- Maintain counter using principle of inclusion and exclusion

Ingredients for given PMC instance (P, F)

- Intersection PMC c for each set σ of rows:

  $\mathrm{PMC}(\sigma)_{\leq t} = |\bigcap_{u \in \sigma} \{J \cap P \mid I(u) \subseteq J \subseteq \mathbf{var}_{\leq t}, J \models F_{\leq t}\}|$

- Computing c

Dynamic Programming via Table algorithm $\mathcal{P}$ for PMC

Ingredients for given PMC instance (P, F)

- Equivalence Classes (Partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$
- Map set σ of rows to counter ipmc (intersection PMC)
- Maintain counter using principle of inclusion and exclusion

Ingredients for given PMC instance (P, F)

- Intersection PMC c for each set σ of rows:

  $\mathrm{PMC}(\sigma)_{\leq t} = |\bigcap_{u \in \sigma} \{J \cap P \mid I(u) \subseteq J \subseteq \mathbf{var}_{\leq t}, J \models F_{\leq t}\}|$
- Computing c

Dynamic Programming via Table algorithm $\mathcal{P}$ for PMC

Ingredients for given PMC instance (P, F)

- Equivalence Classes (Partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$
- Map set σ of rows to counter ipmc (intersection PMC)
- Maintain counter using principle of inclusion and exclusion

Ingredients for given PMC instance (P, F)

- Intersection PMC c for each set σ of rows:

  $\mathrm{PMC}(\sigma)_{\leq t} = |\bigcap_{u \in \sigma} \{J \cap P \mid I(u) \subseteq J \subseteq \mathbf{var}_{\leq t}, J \models F_{\leq t}\}|$
- Computing c

Dynamic Programming via Table algorithm $\mathcal{P}$ for PMC

Ingredients for given PMC instance (P, F)

- Equivalence Classes (Partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$
- Map set σ of rows to counter ipmc (intersection PMC)
- Maintain counter using principle of inclusion and exclusion

Ingredients for given PMC instance (P, F)

- Intersection PMC c for each set σ of rows:

  $\mathrm{PMC}(\sigma)_{\leq t} = |\bigcap_{u \in \sigma} \{J \cap P \mid I(u) \subseteq J \subseteq \mathbf{var}_{\leq t}, J \models F_{\leq t}\}|$
- Computing c

Dynamic Programming via Table algorithm $\mathcal{P}$ for PMC

Ingredients for given PMC instance (P, F)

- Equivalence Classes (Partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$
- Map set σ of rows to counter ipmc (intersection PMC)
- Maintain counter using principle of inclusion and exclusion

Ingredients for given PMC instance (P, F)

- Intersection PMC c for each set σ of rows:

  $\mathrm{PMC}(\sigma)_{\leq t} = |\bigcap_{u \in \sigma} \{J \cap P \mid I(u) \subseteq J \subseteq \mathbf{var}_{\leq t}, J \models F_{\leq t}\}|$
- Computing c

Towards projection via local algorithm $\mathcal{P}$

Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$

t :

...	...	x
...	...	0
...	...	0
...	...	1

...	...	...	x	counter		
...	...	...	0	1		
...	...	...	0	3	1	
...	...	...	0	2	2	1
...	...	...	1	3		

Towards projection via local algorithm $\mathcal{P}$

Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$

t :

...	...	x
...	...	0
...	...	0
...	...	1

...	...	...	x	counter
...	...	...	0	1
...	...	...	0	3
...	...	...	0	2
...	...	...	1	3

1 1 2

Towards projection via local algorithm $\mathcal{P}$

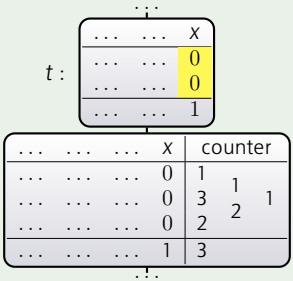
Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$



Towards projection via local algorithm $\mathcal{P}$

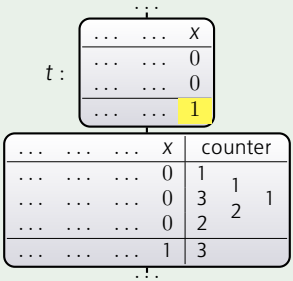
Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$



Towards projection via local algorithm $\mathcal{P}$

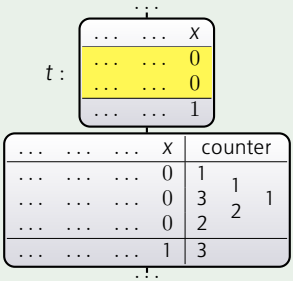
Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$



Towards projection via local algorithm $\mathcal{P}$

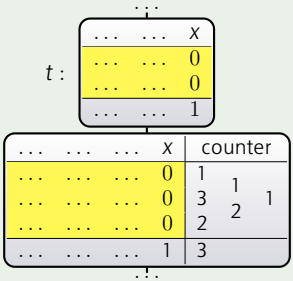
Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$



Towards projection via local algorithm $\mathcal{P}$

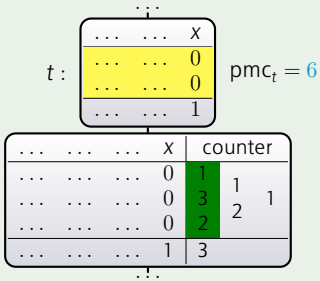
Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$



Towards projection via local algorithm $\mathcal{P}$

Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$

t :

...	...	x
...	...	0
...	...	0
...	...	1

$\text{pmc}_t = 6 - 4^*$

...	...	...	x	counter		
...	...	...	0	1	1	1
...	...	...	0	3	2	
...	...	...	0	2		
...	...	...	1	3		

*: Counter for 1st and 3rd row: 1

Towards projection via local algorithm $\mathcal{P}$

Ingredients for given PMC instance (F, P)

- Equivalence classes (partitioning) of table rows

  For rows u, v of a table: $u \equiv_P v \iff I(u) \cap P = I(v) \cap P$

- Map subsets of resulting "row classes" to a counter
- Maintain counters using the principle of inclusion and exclusion

Example: Computing pmc_t given $P = \{x, y, z\}$

t :

...	...	x
...	...	0
...	...	0
...	...	1

$\mathrm{pmc}_t = 6 - 4^* + 1 = 3$

...	...	...	x	counter		
...	...	...	0	1	1	1
...	...	...	0	3	2	
...	...	...	0	2		
...	...	...	1	3		

*: Counter for 1st and 3rd row: 1

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$
\begin{aligned}
|\bigcup_{X_i \in \mathcal{X}} X_i| &= |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i| \\
&= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \quad + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|
\end{aligned}
$$

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$
\begin{aligned}
|\bigcup_{X_i \in \mathcal{X}} X_i| &= |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i| \\
&= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \qquad + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|
\end{aligned}
$$

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$\underbrace{|\bigcup_{X_i \in \mathcal{X}} X_i|}_{\text{pmc}_t} = |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|$$

$$= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \quad + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|$$

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$\underbrace{|\bigcup_{X_i \in \mathcal{X}} X_i|}_{\text{pmc}_t} = |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|$$

$$= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \quad + (-1)^{|\mathcal{X}|-1} \cdot \underbrace{|\bigcap_{X_i \in \mathcal{X}} X_i|}_{\text{ipmc}_t}$$

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$\underbrace{|\bigcup_{X_i \in \mathcal{X}} X_i|}_{\text{pmc}_t} = |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|$$

$$= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \quad + (-1)^{|\mathcal{X}|-1} \cdot \underbrace{|\bigcap_{X_i \in \mathcal{X}} X_i|}_{\text{ipmc}_t}$$

Towards projection via local algorithm $\mathcal{P}$

Principle of Inclusion-Exclusion (PIE) for family $\mathcal{X}$ of finite sets X_i

$$\underbrace{|\bigcup_{X_i \in \mathcal{X}} X_i|}_{\text{pmc}_t} = |X_1| + |X_2| + \ldots - |X_1 \cap X_2| + \ldots + (-1)^{|\mathcal{X}|-1} \cdot |\bigcap_{X_i \in \mathcal{X}} X_i|$$

$$= \Sigma_{I \subsetneq \{1,\ldots,|\mathcal{X}|\}} (-1)^{|I|-1} \cdot |\bigcap_{i \in I} X_i| \quad + (-1)^{|\mathcal{X}|-1} \cdot \underbrace{|\bigcap_{X_i \in \mathcal{X}} X_i|}_{\text{ipmc}_t}$$

Local algorithm $\mathcal{P}$ in more details:

- Only store ("counter") ipmc_t-values
- pmc_t-values are used intermediately
- Generalization of PIE that – instead of cardinalities – relies on stored ipmc_t-values

Back to Our Example

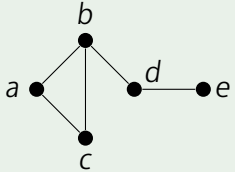
$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

$Mod(F) = \{\{a, b\}, \{a, b, c\}, \{a, b, y\}, \{a, b, c, x\}, \{b\}, \{b, c\}, \{b, y\}, \{b, c, x\}\}$
$\mathrm{PMod}_P(F) = \{\emptyset, \{x\}, \{y\}\}, \mathrm{PMC}_P(F) = |\mathrm{PMod}_P(F)| = 3$

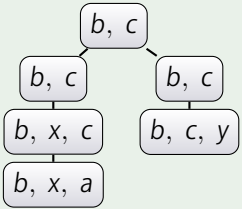
1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$



1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c
1	0
1	1

t_3

b	c
1	0
1	1

t_2

b	x	c
1	0	0
1	0	1
1	1	1

t_1

b	x	a
1	0	0
1	0	1
1	1	0
0	1	1
1	1	1

t_5

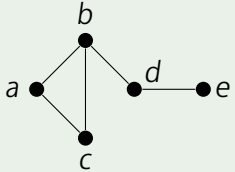
b	c
0	0
1	0
1	1

t_4

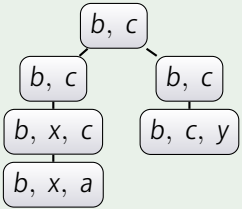
b	c	y
0	0	0
1	0	0
1	1	0
0	0	1
1	0	1

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$



1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc
1	0	2
1	1	2

1

t_3

b	c	ipmc
1	0	1
1	1	2

1

t_5

b	c	ipmc
1	0	2
1	1	1

1

t_2

b	x	c	ipmc
1	0	0	1
1	0	1	1
1	1	1	1

1

t_4

b	c	y	ipmc
1	0	0	1
1	1	0	1
1	0	1	1

1

t_1

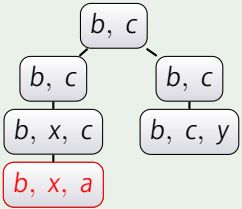
b	x	a	ipmc
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	1

1

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

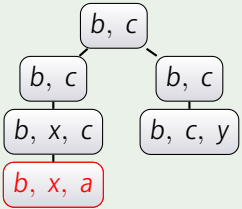
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

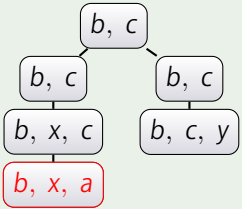
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

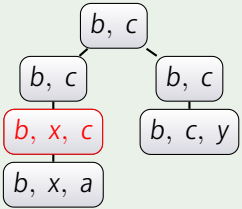
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

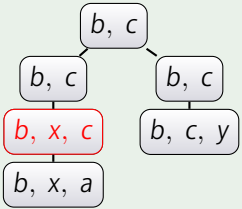
t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

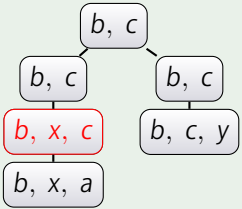
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

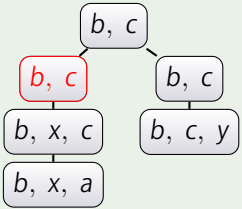
t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

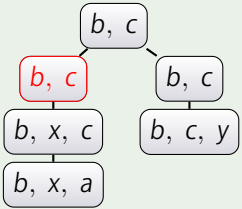
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

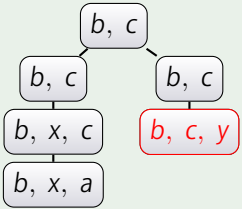
t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

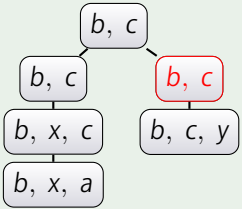
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

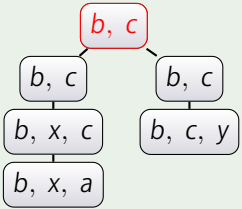
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

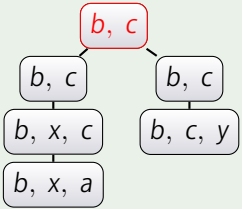
t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

$\rightsquigarrow$ **PMC**$_P(\varphi) = 2 + 2 - 1 = 3$

$\forall x, y.\ \exists a, b, c.\ \varphi$ invalid

t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

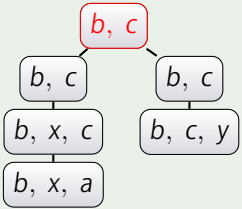
t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

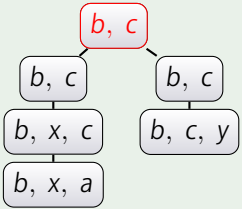
1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



Back to Our Example

$P = \{x, y\}$, $F = (\neg a \vee b \vee x) \wedge (a \vee b) \wedge (c \vee \neg x) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg c \vee \neg y)$

1. Local algorithm $\mathcal{S}$
2. Purge non-solutions
3. Solve PMC via $\mathcal{P}$



Runtime: $2^{2^{O(tw)}} \cdot \text{poly}(|\varphi|)$

t_6

b	c	ipmc	
1	0	2	1
1	1	2	

t_3

b	c	ipmc	
1	0	1	1
1	1	2	

t_5

b	c	ipmc	
1	0	2	1
1	1	1	

t_2

b	x	c	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	1	1	

t_4

b	c	y	ipmc	
1	0	0	1	1
1	1	0	1	
1	0	1	1	

t_1

b	x	a	ipmc	
1	0	0	1	1
1	0	1	1	
1	1	0	1	1
1	1	1	1	

Projected Model Counting

Summary

Conclusion & Future Work

Projected Model Counting (PMC)

- PMC not only believed harder than #SAT w.r.t. classical counting complexity

$\rightsquigarrow$ Under ETH, PMC requires at least double-exponential runtime (in the tw)

- PMC for QBF can also be solved via the presented algorithm $\mathcal{DP}_{\mathcal{P}}$ [FHecherMorakWoltran18]
- Established algorithm $\mathcal{DP}_{\mathcal{P}}$ can also count projected solutions for AI-relevant problems, as, e.g., Argumentation [FHecherMeier19], logic programming [FHecher18], etc.

Conclusion & Future Work

Projected Model Counting (PMC)

- PMC not only believed harder than #SAT w.r.t. classical counting complexity

$\rightsquigarrow$ Under ETH, PMC requires at least double-exponential runtime (in the tw)

- PMC for QBF can also be solved via the presented algorithm $\mathcal{DP}_{\mathcal{P}}$ [FHecherMorakWoltran18]
- Established algorithm $\mathcal{DP}_{\mathcal{P}}$ can also count projected solutions for AI-relevant problems, as, e.g., Argumentation [FHecherMeier19], logic programming [FHecher18], etc.

Answer-Set Programming

Wake up again!

Even if you are not entirely excited by Algorithms and Complexity Theory, the following might be very interesting for you in industrial practice later.

Purpose of this Chapter

1. Get familiar with Answer Set Programming (ASP)
2. Consider Parameterizations of ASP
3. Do some hardness reductions

Based on join research [FSzeider'15a+b], [FHecherMorakWoltran'17].

Motivation

Answer Set Programming (ASP)

- Declarative programming (modelling) framework
- Widely applied in artificial intelligence and reasoning
- Most reasoning problems of propositional disjunctive ASP located on the second level of the Polynomial Hierarchy

Answer-Set Programming

Nutshell

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SAT)
- ASP embraces many emerging application areas
- ASP is not Prolog! (ASP is model based; more like SAT)

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SAT)
- ASP embraces many emerging application areas
- ASP is not Prolog! (ASP is model based; more like SAT)

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SAT)
- ASP embraces many emerging application areas
- ASP is not Prolog! (ASP is model based; more like SAT)

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SAT)
- ASP embraces many emerging application areas
- ASP is not Prolog! (ASP is model based; more like SAT)

Answer Set Programming

in a Nutshell

- ASP is an approach to declarative problem solving, combining
  - a rich yet simple modeling language
  - with high-performance solving capacities
- ASP has its roots in
  - (deductive) databases
  - logic programming (with negation)
  - (logic-based) knowledge representation and (nonmonotonic) reasoning
  - constraint solving (in particular, SAT)
- ASP embraces many emerging application areas
- ASP is not Prolog! (ASP is model based; more like SAT)

Answer-Set Programming

Using ASP

What does ASP offer?

- Integration of DB, KR, SAT, PB, CSP, SMT, ILP
- Succinct problem representations

$\Rightarrow$ Rapid application development tool

- Easy handling of dynamic, knowledge intensive applications, including: data, rules, exceptions
- Highly efficient solvers (important to success story: CDCL-solving)

Two Sides of a Coin

- ASP as High-level Language
  - Express problem instance(s) as sets of facts
  - Encode problem (class) as a set of rules
  - Read off solutions from stable models of facts and rules
- ASP as Low-level Language
  - Compile a problem into a logic program
  - Solve the original problem by solving its compilation

What is ASP good for?

- Combinatorial search problems in realm of P, NP, and $\mathrm{NP}^{\mathrm{NP}}$ (Σ_2^p) (some with substantial amount of data), like
  - Automated Planning
  - Code Optimization
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - Systems Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

What is ASP good for?

- Combinatorial search problems in realm of P, NP, and $\mathrm{NP}^{\mathrm{NP}}$ (Σ_2^p) (some with substantial amount of data), like
  - Automated Planning
  - Code Optimization
  - Composition of Renaissance Music
  - Database Integration
  - Decision Support for NASA shuttle controllers
  - Model Checking
  - Product Configuration
  - Robotics
  - Systems Biology
  - System Synthesis
  - (industrial) Team-building
  - and many many more

Language Constructs

- Variables (over the Herbrand Universe)

  `p(X) :- q(X)` over constants $\{a, b, c\}$ short for
  `p(a) :- q(a), p(b) :- q(b), p(c) :- q(c)`
- Conditional Literals

  `p :- q(X) : r(X)` given `r(a), r(b), r(c)` short
  for `p :- q(a), q(b), q(c)`
- Disjunction

  `p(X) | q(X) :- r(X)`
- Integrity Constraints

  `:- q(X), p(X)`
- Choice

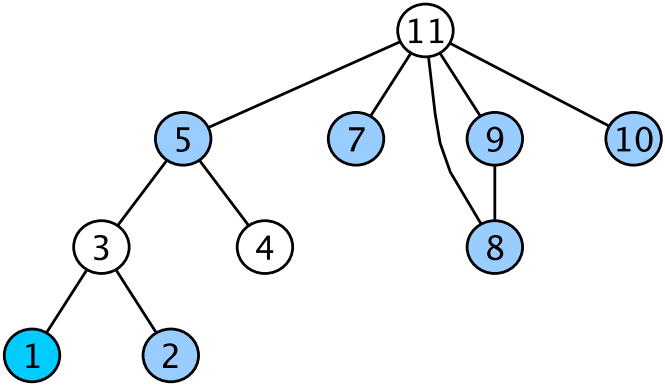
  `2 { p(X,Y) : q(X) } 7 :- r(Y)`
- Aggregates

  `s(Y) :- r(Y), 2 #count { p(X,Y) : q(X) } 7`
  also: `#sum`, `#avg`, `#min`, `#max`, `#even`, `#odd`

An easy example... k-vertex Cover

Definition (Recall)

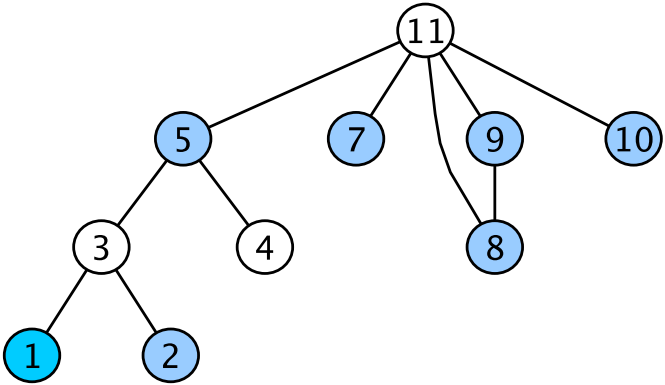
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



An easy example... k-Vertex Cover

Definition (Recall)

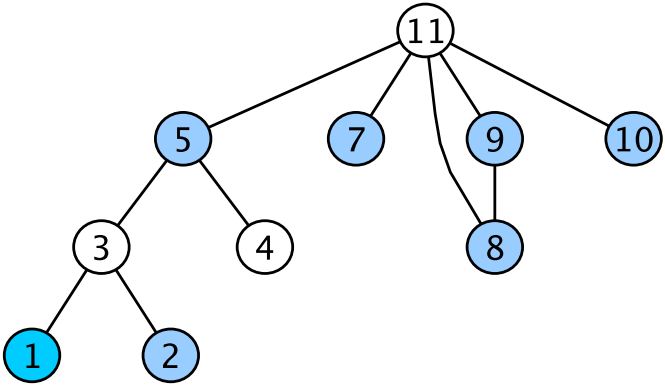
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



An easy example... k-Vertex Cover

Definition (Recall)

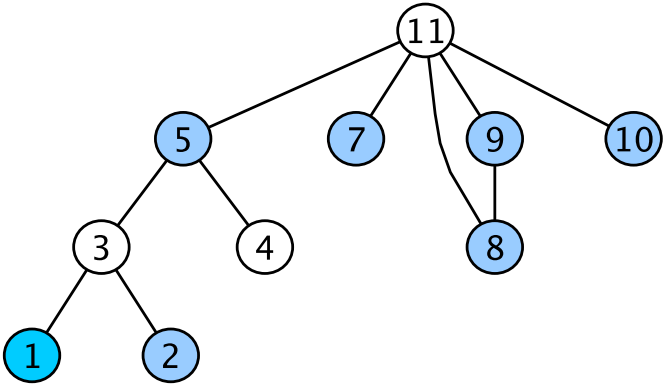
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



An easy example... k-Vertex Cover

Definition (Recall)

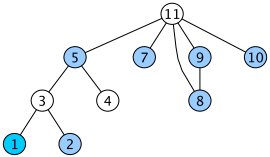
$S \subseteq V$ is a vertex cover of G if for every $uv \in E$ we have $u \in S$ or $v \in S$
In other words, at least one endpoint of an edge belongs to S.



Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to S.

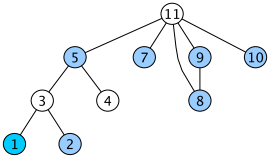


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to *S*.

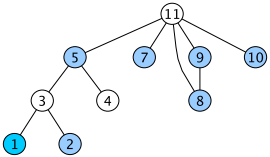


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to S.

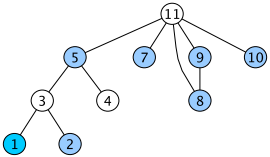


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to S.

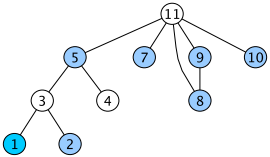


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to S.

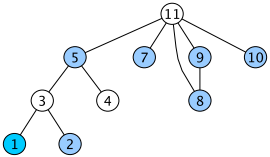


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to *S*.

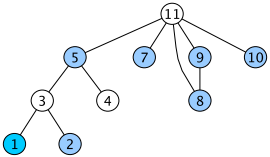


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to *S*.

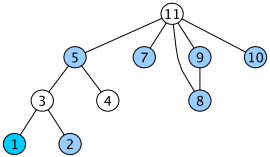


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to *S*.

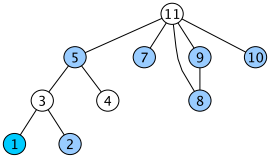


1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

Encode *k*-Vertex Cover into ASP

Definition (Recall)

At least one endpoint of an edge belongs to S.



1. Edges in the following form: `e(x,y).`
   `e(1,3). e(2,3). e(4,5). e(5,11). e(7,11).`
   `e(9,11). e(8,11). e(10,11).`
2. We need both directions: `e(Y,X) :- e(X,Y).`
3. Need vertices later: `v(X) :- e(X,_).`
4. Choose an endpoint: `{vc(X); vc(Y)} :- e(X,Y).`
5. Make sure that one endpoint is selected:
   `:- e(X,Y), not vc(X), not vc(Y).`
6. Alternative (replace 3+4) by:
   `1{vc(X); vc(Y)} :- e(X,Y).`
7. How about the size? `:- k+1 #count {v(X) : vc(X)}.`
8. Define size `# const k=4`
9. Goody for solving (show only vc): `#show vc/1.`

k-Vertex Cover Encoding

```
%Item (1)
e(1,3). e(2,3). e(4,5). e(5,11).  e(7,11). e(9,11). e(8,11). e(10,11).

%Item (2) + (3)
e(Y,X) :- e(X,Y).        v(X) :- e(X,Y).

% Item (4)
{vc(X); vc(Y)} :- e(X,Y).

%Item (5)
:- e(X,Y), not vc(X), not vc(Y).

%Item (7)
c(C) :- #count {v(X) : vc(X)} = C.
:- k+1 #count {v(X) : vc(X)}.

%Item (8) + (9)
#const k=2. #show vc/1. #show c/1.
```

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. [illegible]
      [illegible]

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
      Why?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
       How?
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
   1. That we talk about a reduction here, right?
   2. Reduce from VC to ASP
   3. Provide an encoding. Encodings are reductions
   4. What do we show with this direction (VC $\Rightarrow$ ASP)?
      VC is "less hard" than ASP (or ASP is at least as hard as VC)
      $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
   5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
      Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
    $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
       $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
    $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

What does this has to do with complexity?

1. Well I like those things.
2. But indeed we can pretend:
    1. That we talk about a reduction here, right?
    2. Reduce from VC to ASP
    3. Provide an encoding. Encodings are reductions
    4. What do we show with this direction (VC $\Rightarrow$ ASP)?
       VC is "less hard" than ASP (or ASP is at least as hard as VC)
    $\Rightarrow$ ASP is NP-hard (since VC is NP-hard)
    5. ASP $\in$ NP (membership or ASP $\Rightarrow$ VC)?
       Not possible. Why? ASP is much more powerful. Even if we talk about propositional ASP (no variables over constants).

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another

   k-VC $\Rightarrow$ ASP

   (assume ASP outputs only yes/no; not also an answer set)

2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:

   If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

   (True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:

   If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

   (True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another
   k-VC $\Rightarrow$ ASP
   (assume ASP outputs only yes/no; not also an answer set)
2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another
   k-VC $\Rightarrow$ ASP
   (assume ASP outputs only yes/no; not also an answer set)
2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another

✓ k-VC $\Rightarrow$ ASP
(assume ASP outputs only yes/no; not also an answer set)

2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another
   k-VC $\Rightarrow$ ASP
   (assume ASP outputs only yes/no; not also an answer set)
2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another
   k-VC $\Rightarrow$ ASP
   (assume ASP outputs only yes/no; not also an answer set)
2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

1. Reduce from one decision problem to another
   k-VC $\Rightarrow$ ASP
   (assume ASP outputs only yes/no; not also an answer set)
2. For every yes instance of VC, we get yes from our ASP program and vice versa

2a) VC $\Rightarrow$ ASP:
If G has a vertex cover of size k, then solver on our encoding says yes. In other words, we do not forget vertex covers.

(True, because that's exactly how we designed our encoding.)

2b) ASP $\Rightarrow$ VC:
If the solver returns yes on the encoding for graph G (so there is an answer set), then there is a vertex cover of size k. In other words, we do not invent vertex covers.

(True, because that's exactly how we designed our encoding.)

Remark: Formally, just saying that that we designed the encoding exactly how it is supposed to be is really not enough. But, since I did not formally introduce the semantics of ASP, we cannot give a formal proof here.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

3. Can we encode in polynomial time?

Let's do a runtime analysis.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

3. Can we encode in polynomial time?

Let's do a runtime analysis.

Reduction: VC $\Rightarrow$ ASP

Is our reduction (encoding above) a many-to-one reduction?

✓3. Can we encode in polynomial time?

Let's do a runtime analysis.

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

Reductions cont.

Say we parameterize in the solution size (size of the vertex cover).

Parameterized Reduction

We already checked the many-one-reduction. Remains:

+ Parameter is preserved

Solution size: k

What do you think?

1. How about atoms that are set to true in a solution.
2. Problem: atoms of the form $v(1), v(2), \ldots$
3. We took care of it by projection. Right?...
4. However, we can also modify the encoding so that it also preserves the parameter (see whiteboard).

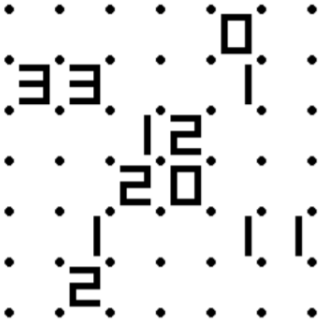
Hands On

Pick up computers and try out ASP solvers
See what happens with the vertex cover encoding

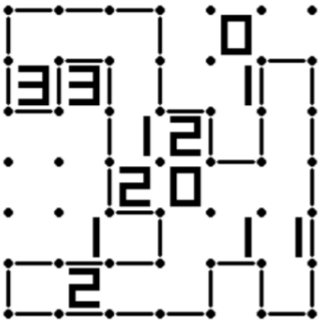
Jacob's [KTH] Lecture Puzzle: Slitherlink

Problem Description

- Given: planar graph and clues
- Task: draw a cycle in graph, such that
  - # edges incident to a clue equals the value of the clue
  - Simplification: grid graph



(a) Slitherlink problem instance



How to solve it with ASP?

1. Encoder (Instance):
   Transforms grid graph, clues into (atomic) predicates
2. Encoding (Problem):
   Describes a solution of the problem slitherlink
3. Decoder:
   Maps it back into a visual representation

How to solve it with ASP?

1. Encoder (Instance):
   Transforms grid graph, clues into (atomic) predicates
2. Encoding (Problem):
   Describes a solution of the problem slitherlink
3. Decoder:
   Maps it back into a visual representation

No magic.
We just need some representation.

Idea:

- Generate grid (consecutive ids) for graph and clues
- edge(x,y) ... there is an edge from x to y
- clue(c,v) ... cell c has value v
- cell_contains(c,x,y) ... edge(x,y) is a neighbor of cell c

How to solve it with ASP?

1. Encoder (Instance):
   Transforms grid graph, clues into (atomic) predicates
2. Encoding (Problem):
   Describes a solution of the problem slitherlink
3. Decoder:
   Maps it back into a visual representation

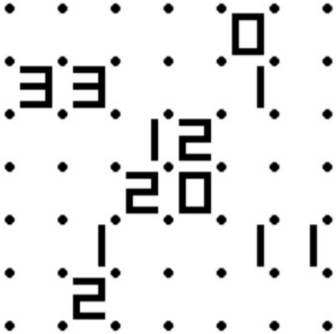
Not implemented (technical exercise).

How to solve it with ASP?

1. Encoder (Instance):
   Transforms grid graph, clues into (atomic) predicates
2. Encoding (Problem):
   Describes a solution of the problem slitherlink
3. Decoder:
   Maps it back into a visual representation

We write an ASP encoding with the following ideas...

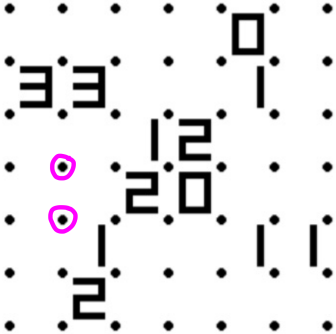
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

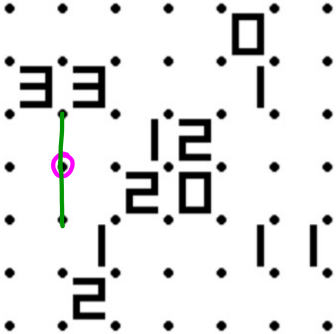
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (and vice versa)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

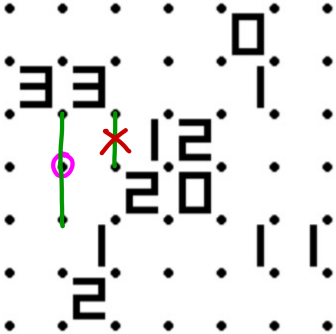
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

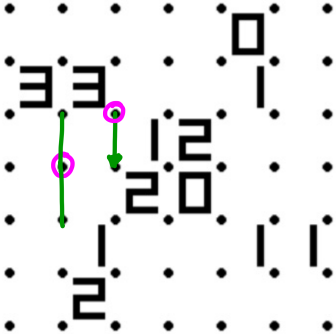
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

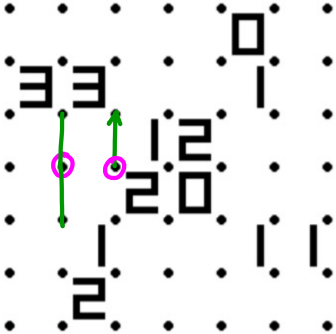
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

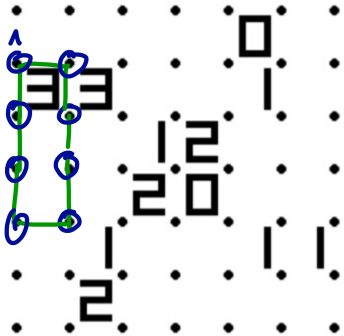
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

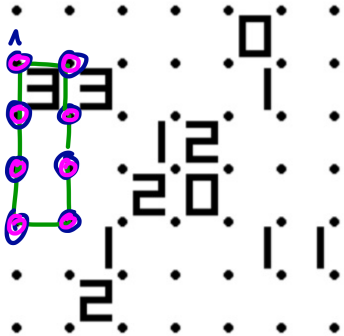
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

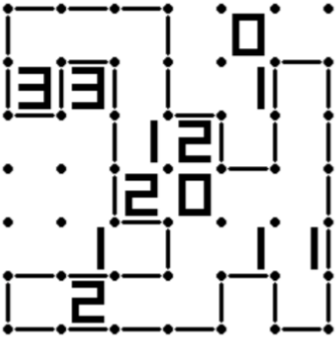
Idea: ASP encoding slitherlink



(a) Slitherlink problem instance.

1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

Idea: ASP encoding slitherlink



(b) Solution.

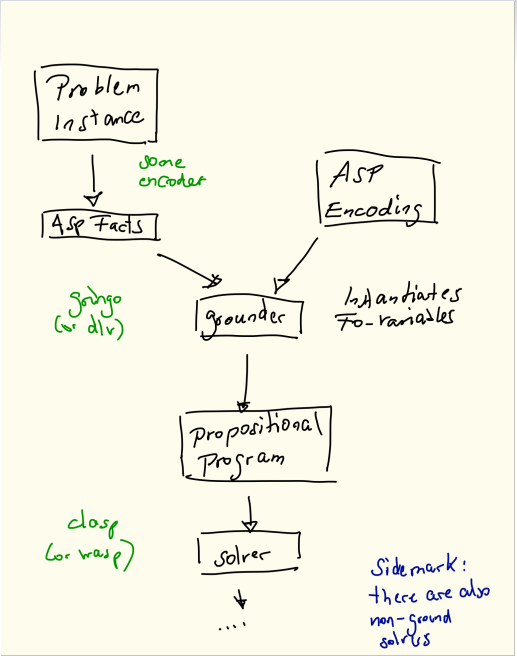
1. Pick terminals
   (from endpoints of an edge)
2. Start a link into both directions from a terminal
3. Make sure there is no link unless they are terminals (both dir.)
4. Define reachable over links
5. Make sure every terminal is reachable from itself
6. Make sure neighbor counts for clues are fulfilled

Listing

```
%(1) Pick (neighboring) terminals
{ terminal(U;V) : edge(U,V) }.

%(2) Pick two links starting from the terminals
2 { link(U,V) : edge(U,V); link(W,U) : edge(W,U) } 2 ← terminal(U).

%(3) No link unless terminals
← link(U,V), not terminal(U).
← link(U,V), not terminal(V).

%(4) Define reachable over links
%Start with fixed min order (for symmetry reasons)
reached(N) ← N = #min{V : terminal(V)}.

%(5) One terminal is reachable from itself
← terminal(U), not reached(U).
%direction does not matter
reached(V) ← reached(U), link(U,V).
reached(V) ← reached(U), link(V,U).

%(5) Make sure neighbor counts for clues are fulfilled
← clue(C,N), not N { link(U,V) : cell_contains(C,U,V) } N.
#show link/2.
```

Demo?

▸ Open Demo Shell


Problem instance
done encoder
ASP Facts
ASP Encoding
Gringo (or dlv)
grounder
Instantiates FO-variables
Propositional Program
clasp (or wasp)
solver
....
Sidemark: there are also non-ground solvers

Answer-Set Programming

Notations / Semantics

Basic Notations

Answer Set Programs

- (Disjunctive) program P set of rules of the form

$$r : \underbrace{x_1 \vee \cdots \vee x_l}_{H(r)} \quad \leftarrow \quad \underbrace{y_1, \ldots, y_n}_{B^+(r)}, \underbrace{\neg z_1, \ldots, \neg z_m}_{B^-(r)}$$

Solutions of a Program (Answer Sets) AS(P)

- Is $M \subseteq \mathrm{at}(P)$ an answer set (stable model) of P?
  - (i) M is a model of P
  - (ii) M is a minimal model wrt P^M

Basic Notations

Answer Set Programs

- (Disjunctive) program P set of rules of the form

$$r : \underbrace{x_1 \vee \cdots \vee x_l}_{H(r)} \quad \leftarrow \quad \underbrace{y_1, \ldots, y_n}_{B^+(r)}, \underbrace{\neg z_1, \ldots, \neg z_m}_{B^-(r)}$$

Solutions of a Program (Answer Sets) AS(P)

- Is $M \subseteq \mathrm{at}(P)$ an answer set (stable model) of P?
  - (i) M is a model of P
  - (ii) M is a minimal model wrt P^M

Answer Sets (basic idea)

GL reduct

Let P be a program and $M \subseteq \mathrm{at}(P)$.

$$P_M = \{\, H(r) \leftarrow B^+(r) \mid r \in P \text{ and } B^-(r) \cap M = \emptyset \,\}$$

Note P^M only considers negative body literals

Answer Sets (basic idea)

GL reduct

Let P be a program and $M \subseteq \mathrm{at}(P)$.

$$P_M = \{\, H(r) \leftarrow B^+(r) \mid r \in P \text{ and } B^-(r) \cap M = \emptyset \,\}$$

Note P^M only considers negative body literals

Problem of Interest

1. Consistency: Does P have an answer set?
2. Brave (Cautious) Reasoning : Given P and an atom a of P. Does a belong to some (all) answer set of P?
3. Compute: Output an answer set of P.
4. Count: Output the number of answer sets of P.

Answer Sets (basic example)

Consider the program P and its three models:

$$F_P \quad q \wedge ((q \wedge \neg r) \rightarrow p)$$

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Program P has only one answer set:

$$\{p, q\}$$

$$P \quad \begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \neg r \end{array}$$

Skip Examples

Answer Sets (basic example)

Consider the program P and its three models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

Program P has only one answer set:

$$\{p, q\}$$

F_P $\boxed{q \ \wedge \ ((q \wedge \neg r) \rightarrow p)}$

$$P \quad \boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \neg r \end{array}}$$

Skip Examples

Answer Sets (basic example)

Consider the program P and its three models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

F_P $\boxed{q \ \wedge \ ((q \wedge \neg r) \rightarrow p)}$

Program P has only one answer set:

$$\{p, q\}$$

P

$$\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \neg r \end{array}}$$

Skip Examples

Answer Sets (basic example)

Consider the program P and its three models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

F_P $\boxed{q \ \wedge \ ((q \wedge \neg r) \rightarrow p)}$

Program P has only one answer set:

$$\{p, q\}$$

$$\begin{array}{lll} p & \mapsto & q \ 1 \leftarrow \\ q & \mapsto & p \ 1 \leftarrow q, \neg r \\ r & \mapsto & 0 \end{array}$$

Skip Examples

Answer Sets (basic example)

Consider the program P and its three models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

F_P $\boxed{q \ \wedge \ ((q \wedge \neg r) \rightarrow p)}$

Program P has only one answer set:

$$\{p, q\}$$

P

$$\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \neg r \end{array}}$$

Skip Examples

Answer Sets (basic example)

Consider the program P and its three models:

$$\{p, q\}, \{q, r\}, \text{ and } \{p, q, r\}$$

F_P $\boxed{q \ \wedge \ ((q \wedge \neg r) \rightarrow p)}$

Program P has only one answer set:

$$\{p, q\}$$

P

$$\boxed{\begin{array}{lll} q & \leftarrow & \\ p & \leftarrow & q, \neg r \end{array}}$$

Skip Examples

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$ ✗ ✗
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$ ✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	$\times$
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	✓
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	$\times$
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	

Skip Examples

[cf. lecture slide by Schaub]

An Example

$$P = \{p \leftarrow p,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\times\ \times$
$\{p\quad\}$	$p \leftarrow p$	$\emptyset$	$\times$
$\{\quad q\}$	$p \leftarrow p$ $q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$	$p \leftarrow p$	$\emptyset$	$\times$

Skip Examples

[cf. lecture slide by Schaub]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	$\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$	$\checkmark$
$\{\quad q\}$	$q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$		$\emptyset$	

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$ $\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$ $\checkmark$
$\{\quad q\}$	$q \leftarrow$	$\{q\}$ $\checkmark$
$\{p, q\}$		$\emptyset$

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$ ✗
$\{p\quad\}$	$p \leftarrow$	$\{p\}$ ✓
$\{\quad q\}$	$q \leftarrow$	$\{q\}$ ✓
$\{p, q\}$		$\emptyset$

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	$\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$	✓
$\{\quad q\}$	$q \leftarrow$	$\{q\}$	✓
$\{p, q\}$		$\emptyset$	

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	$\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$	✓
$\{\quad q\}$	$q \leftarrow$	$\{q\}$	✓
$\{p, q\}$		$\emptyset$	

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	$\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$	$\checkmark$
$\{\quad q\}$	$q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$		$\emptyset$	

[cf. lecture slide by Schaub et al.]

A 2nd Example

$$P = \{p \leftarrow \neg q,\ q \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\quad\}$	$p \leftarrow$ $q \leftarrow$	$\{p, q\}$	$\times$
$\{p\quad\}$	$p \leftarrow$	$\{p\}$	$\checkmark$
$\{\quad q\}$	$q \leftarrow$	$\{q\}$	$\checkmark$
$\{p, q\}$		$\emptyset$	$\times$

[cf. lecture slide by Schaub et al.]

A 3rd Example

$$P = \{p \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	$\times$
$\{p\}$		$\emptyset$	

[cf. lecture slide by T. Schaub et al.]

A 3rd Example

$$P = \{p \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$
$\{\ \}$	$p \leftarrow$	$\{p\}$ $\times$
$\{p\}$		$\emptyset$

[cf. lecture slide by T. Schaub et al.]

A 3rd Example

$$P = \{p \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$
$\{\ \}$	$p \leftarrow$	$\{p\}$ ×
$\{p\}$		$\emptyset$

[cf. lecture slide by T. Schaub et al.]

A 3rd Example

$$P = \{p \leftarrow \neg p\}$$

M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	$\times$
$\{p\}$		$\emptyset$	

[cf. lecture slide by T. Schaub et al.]

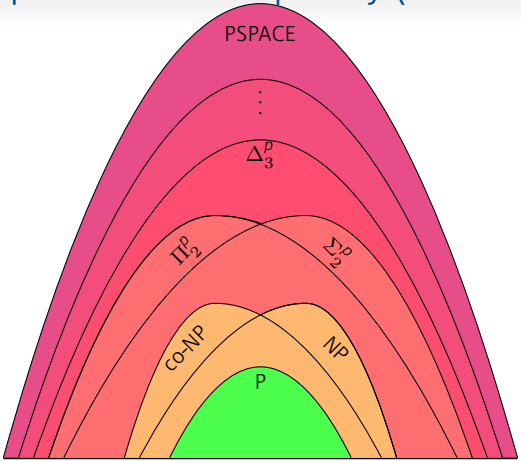
A 3rd Example

$$P = \{p \leftarrow \neg p\}$$

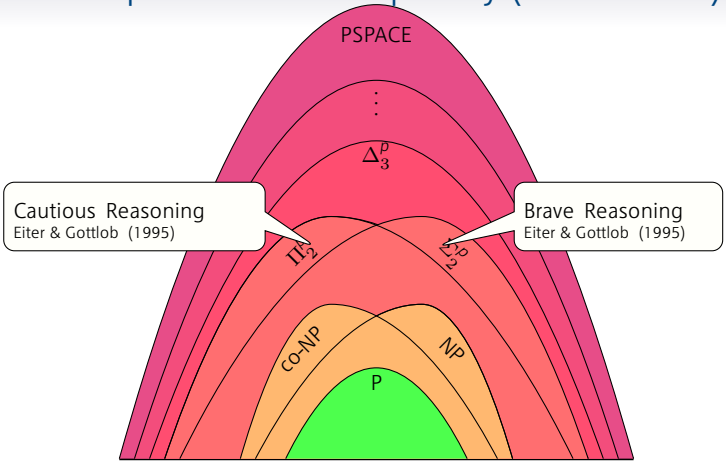
M	P^M	$\mathrm{MinMod}(P^M)$	
$\{\ \}$	$p \leftarrow$	$\{p\}$	$\times$
$\{p\}$		$\emptyset$	$\times$

[cf. lecture slide by T. Schaub et al.]

Computational Complexity (Worst Case)

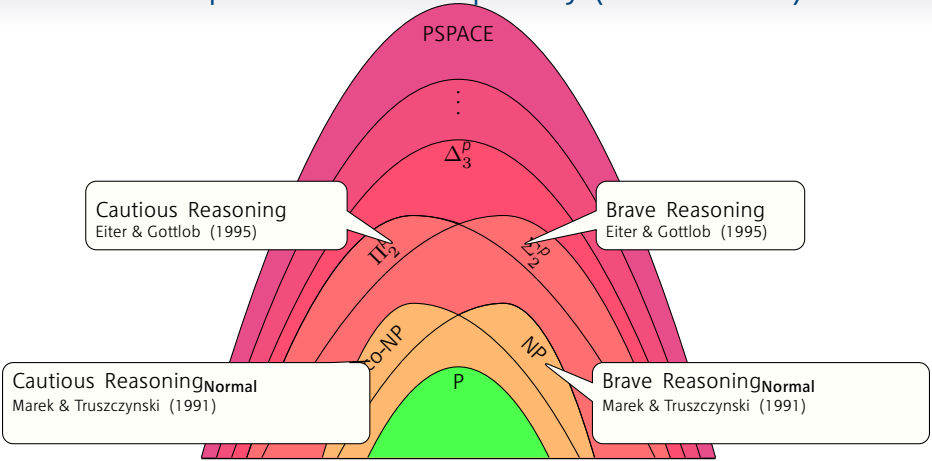


Computational Complexity (Worst Case)



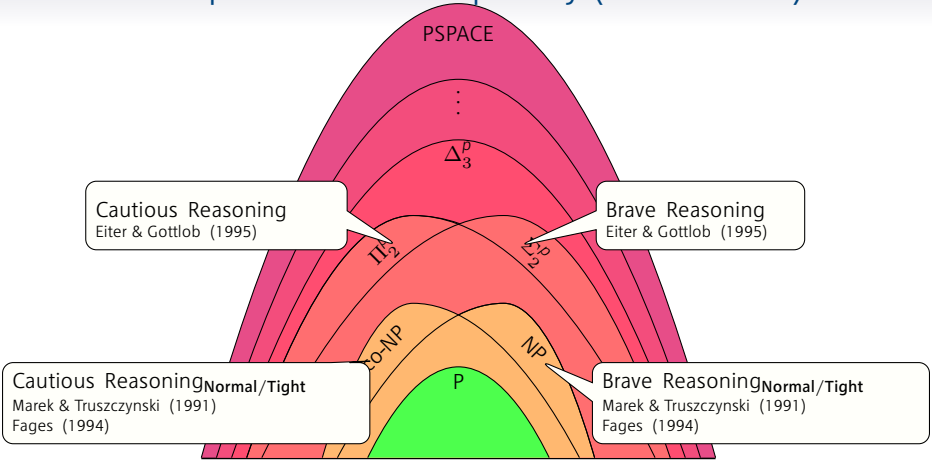
$$x_1 \vee \ldots \vee x_l \leftarrow y_1, \ldots, y_m, \neg z_1, \ldots, \neg z_n$$

Computational Complexity (Worst Case)



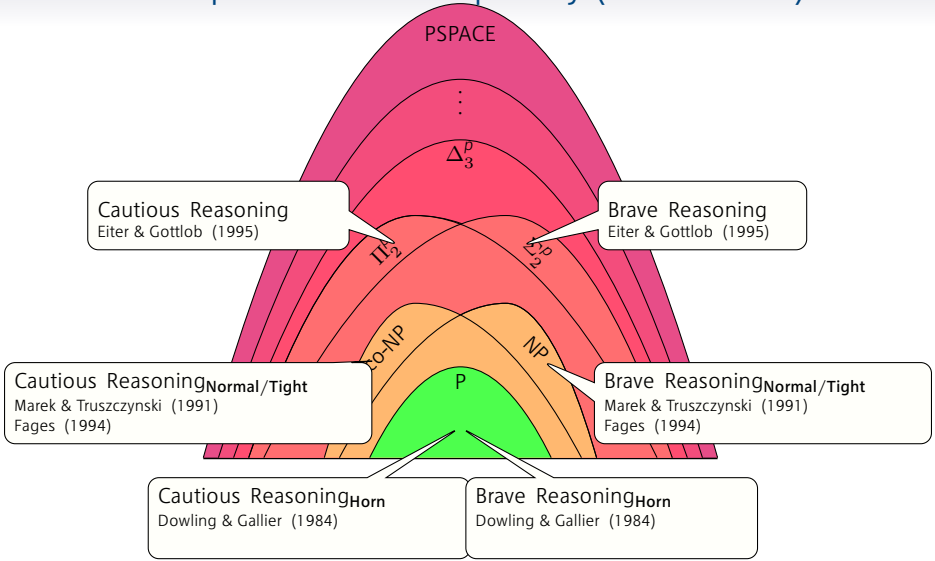
Normal: $\cancel{x_1 \vee \ldots \vee x_l} \leftarrow y_1, \ldots, y_m, \neg z_1, \ldots, \neg z_n$

Computational Complexity (Worst Case)



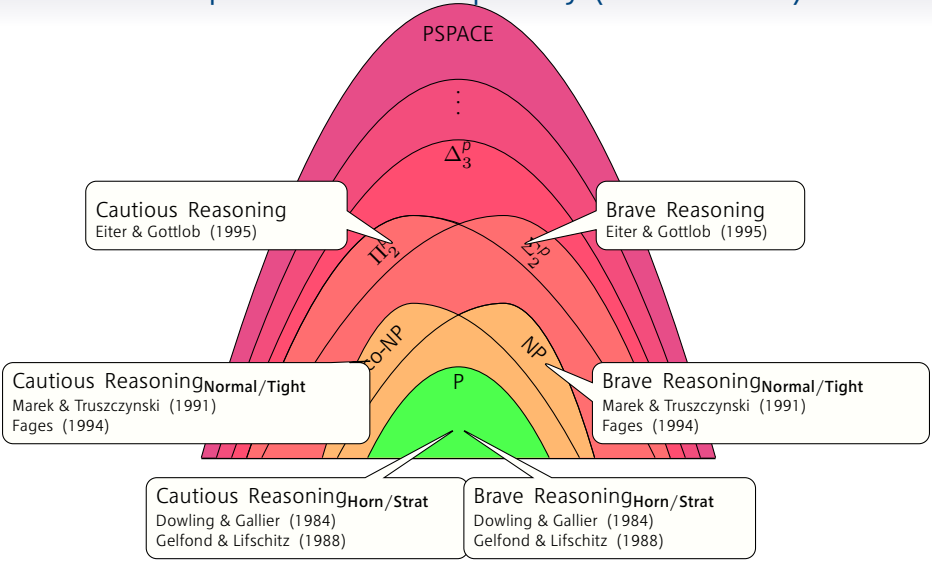
Tight: acyclic positive dependency digraph

Computational Complexity (Worst Case)



Horn: $x_1 \vee \cancel{\ldots \vee x_l} \leftarrow y_1, \ldots, y_m, \cancel{\neg z_1, \ldots, \neg z_n}$

Computational Complexity (Worst Case)



Strat: acyclic dependency digraph

Computational Complexity

There is an analog to Schaefer's classes
Trichotomy for Complexity of ASP
by Mirek Truszczyński

Answer-Set Programming

Backdoors

Backdoors (Informal)

Ideas

- Restrict the input to a certain subclass
  $\Rightarrow$ solve problem efficiently
- Identify backdoor into such a subclass (target class)
- Exploit the backdoor
  $\Rightarrow$ solve the problem faster

Answer-Set Programming

Strong Backdoors

ASP-backdoors

Idea

- Identifying sets of atoms that are important for combinatorial search

Strong ASP-backdoors into $\mathcal{C}$ [FS11,FS15]

- Assigning truth values 0 or 1 to those atoms yields a simplified program P_τ

- P_τ belongs to a target class $\mathcal{C}$ that is "beneficial" for us
- Set X of atoms is a strong backdoor into $\mathcal{C}$ if for all truth assignments $\tau : X \to \{0, 1\}$ simplified program $P_\tau \in \mathcal{C}$

$\Rightarrow$ Parameter: size k of the backdoor

$\Rightarrow$ Measures distance to tractability

Skip...

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignments

$c \mapsto 0$ or $c \mapsto 1$, $g \mapsto 0$ or $g \mapsto 1$

Program under Truth Assignment

Check whether

$$P_{c\mapsto 0, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 0, g\mapsto 1} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 1} \in \textbf{Horn}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignments

$c \mapsto 0$ or $c \mapsto 1$, $g \mapsto 0$ or $g \mapsto 1$

Program under Truth Assignment

Check whether

$$P_{c\mapsto 0, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 0, g\mapsto 1} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 1} \in \textbf{Horn}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignments

$c \mapsto 0$ or $c \mapsto 1$, $g \mapsto 0$ or $g \mapsto 1$

Program under Truth Assignment

Check whether

$$P_{c\mapsto 0, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 0, g\mapsto 1} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 1} \in \textbf{Horn}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignments

$c \mapsto 0$ or $c \mapsto 1$, $g \mapsto 0$ or $g \mapsto 1$

Program under Truth Assignment

Check whether

$$P_{c\mapsto 0, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 0, g\mapsto 1} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 0} \in \textbf{Horn},$$
$$P_{c\mapsto 1, g\mapsto 1} \in \textbf{Horn}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{a \quad \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{a \quad \leftarrow b;\ b \quad \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{a \quad \leftarrow b;\ b \quad \leftarrow e; \qquad\qquad e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 0$

Program under Truth Assignment

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn**

$P_{c,g} \in$ **Horn**

$P_{c,g} \in$ **Horn**

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 1$

Program under Truth Assignment

$$P_{c,g} = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 1$

Program under Truth Assignment

$$P_{c,g} = \{a \quad \leftarrow b;\ b \quad \leftarrow e \qquad\qquad e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 0$ and $g \mapsto 1$

Program under Truth Assignment

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn**

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn**

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 0$

Program under Truth Assignment

$$P_{c,g} = \{ \qquad\qquad b \leftarrow \qquad ;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 0$

Program under Truth Assignment

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn**

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 1$

Program under Truth Assignment

$$P_{c,g} = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 1$

Program under Truth Assignment

$$P_{c,g} = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

$c \mapsto 1$ and $g \mapsto 1$

Program under Truth Assignment

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓

Strong Backdoor into **Horn** (Example)

Program

$$P = \{a \vee c \leftarrow b;\ b \vee c \leftarrow e;\ b \leftarrow c, \neg g;\ e \leftarrow\}$$

Backdoor

$X = \{c, g\}$

Truth Assignment

Program under Truth Assignment

$P_{c,g} \in$ **Horn** ✓ $\quad$ $P_{c,g} \in$ **Horn** ✓

$P_{c,g} \in$ **Horn** ✓ $\quad$ $P_{c,g} \in$ **Horn** ✓

$\Rightarrow$ X is a strong backdoor into **Horn** of P

Answer-Set Programming

Deletion Backdoors

Deletion Backdoors?

Idea

- Again set $X \subseteq \mathrm{at}(P)$ of atoms
- Consider $P - X$
  removing atoms in X from a program P
  (instead of truth assignments)
- X deletion backdoor into $\mathcal{C}$ if simplified program $P - X \in \mathcal{C}$

Deletion Backdoors?

Idea

- Again set $X \subseteq \mathrm{at}(P)$ of atoms
- Consider $P - X$
  removing atoms in X from a program P
  (instead of truth assignments)
- X deletion backdoor into $\mathcal{C}$ if simplified program $P - X \in \mathcal{C}$

Why?

- Hard to find strong backdoors in many cases
- Simplify notation
- Still good enough, but might be larger

▸ Deletion vs. Strong Backdoors

How can we “use” backdoors?

Answer-Set Programming

Backdoor Approach

Backdoor Approach

Phases

0. Fix a target class $\mathcal{C}$
1. Find a backdoor into $\mathcal{C}$
   (backdoor detection)
2. Use backdoor to solve the reasoning problem
   (backdoor evaluation)

Questions

(i) When is backdoor detection fixed-parameter tractable (solvable in time $f(k)n^{\mathcal{O}(1)}$)?

(ii) Is backdoor evaluation fixed-parameter tractable?

Questions

(i) When is backdoor detection fixed-parameter tractable (solvable in time $f(k)n^{\mathcal{O}(1)}$)?

(ii) Is backdoor evaluation fixed-parameter tractable?

Questions

(i) When is backdoor detection fixed-parameter tractable (solvable in time $f(k)n^{\mathcal{O}(1)}$)?

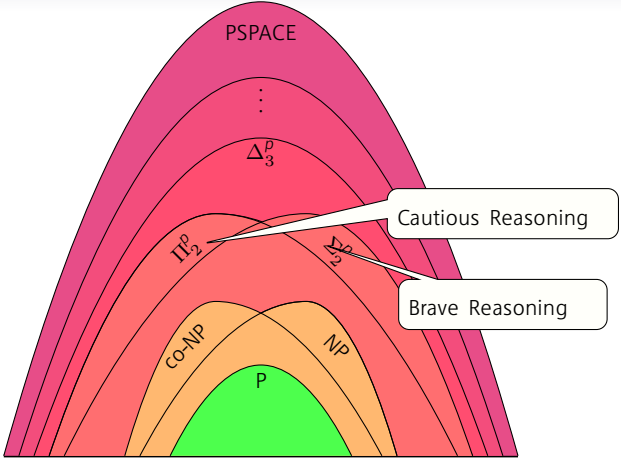
(ii) Is backdoor evaluation fixed-parameter tractable?

Good news: Yes

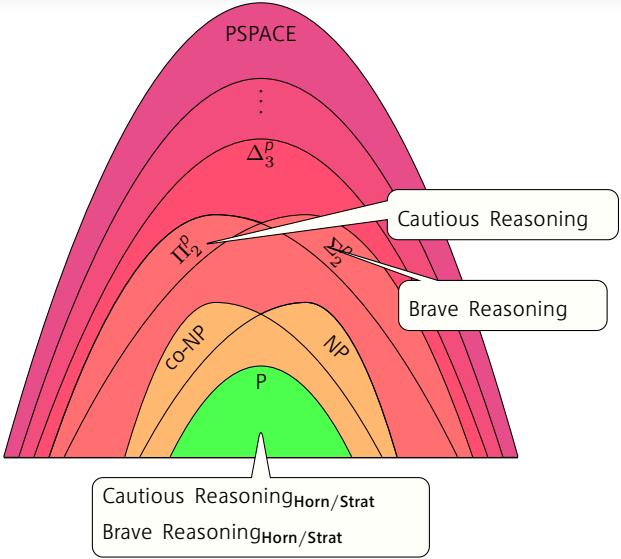
Answer-Set Programming

Backdoor Evaluation

Backdoor Evaluation (Enum. Classes, FS11, FS15)



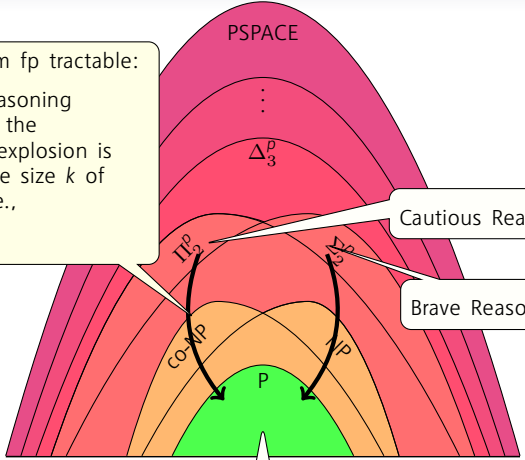
Backdoor Evaluation (Enum. Classes, FS11, FS15)



Backdoor Evaluation (Enum. Classes, FS11, FS15)

Make the problem fp tractable:

Reduce the reasoning problem while the combinatorial explosion is confined to the size k of a backdoor, i.e., $f(k) \cdot n^{\mathcal{O}(1)}$



Cautious Reasoning

Brave Reasoning

Cautious Reasoning$_{\text{Horn/Strat}}$

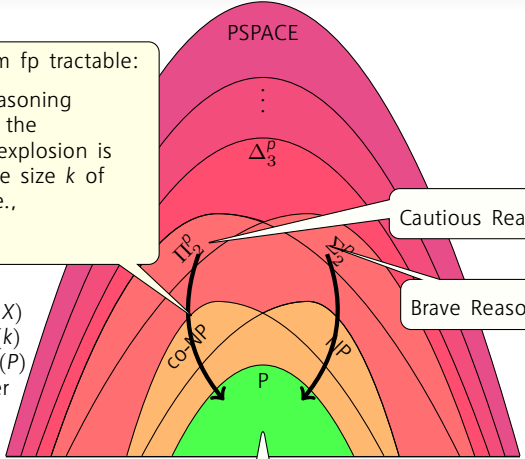
Brave Reasoning$_{\text{Horn/Strat}}$

Backdoor Evaluation (Enum. Classes, FS11, FS15)

Make the problem fp tractable:

Reduce the reasoning problem while the combinatorial explosion is confined to the size k of a backdoor, i.e., $f(k) \cdot n^{\mathcal{O}(1)}$

Main Ingredients:
1. $\mathrm{AS}(P) \subseteq \mathrm{AS}(P, X)$
2. $|\mathrm{AS}(P, X)| \leq g(k)$
3. Check $M \in \mathrm{AS}(P)$ fixed-parameter tractable



Cautious Reasoning

Brave Reasoning

Cautious Reasoning$_{\text{Horn/Strat}}$

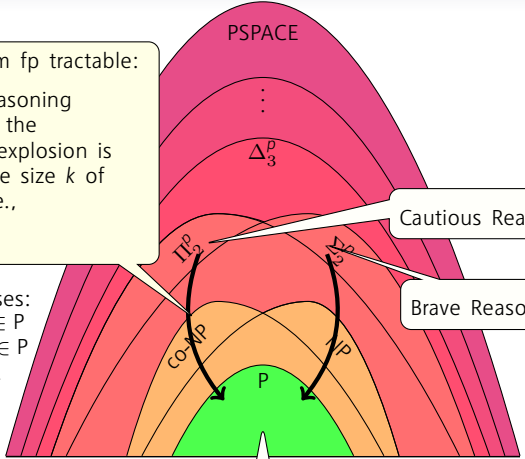
Brave Reasoning$_{\text{Horn/Strat}}$

Backdoor Evaluation (Enum. Classes, FS11, FS15)

Make the problem fp tractable:

Reduce the reasoning problem while the combinatorial explosion is confined to the size k of a backdoor, i.e., $f(k) \cdot n^{\mathcal{O}(1)}$

Good target classes:
1. Membership $\in$ P
2. Enumeration $\in$ P
3. Backdoor Det. $f(k) \cdot n^{\mathcal{O}(1)}$



Cautious Reasoning

Brave Reasoning

Cautious Reasoning$_{\text{Horn/Strat}}$

Brave Reasoning$_{\text{Horn/Strat}}$

Questions

(i) When is backdoor detection fixed-parameter tractable (solvable in time $f(k)n^{\mathcal{O}(1)}$)?

(ii) Is backdoor evaluation fixed-parameter tractable? ✓

Questions

(i) When is backdoor detection fixed-parameter tractable (solvable in time $f(k)n^{\mathcal{O}(1)}$)?

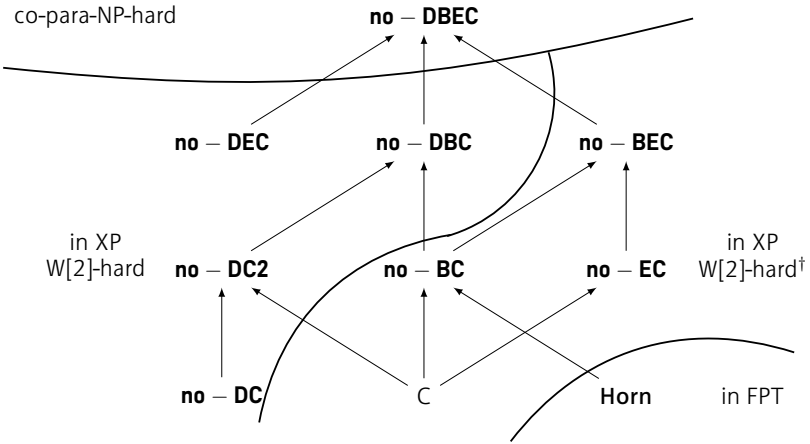
(ii) Is backdoor evaluation fixed-parameter tractable? ✓

News: Well it depends

Answer-Set Programming

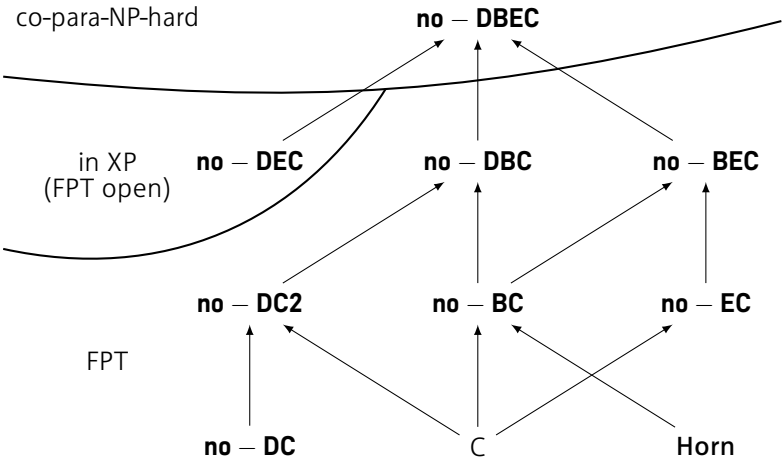
Backdoor Detection

(ii) Strong Backdoor Detection (cont.)



▸ Dependency Digraph ▸ Undirected Dependency Graph ▸ Proof Ideas

(ii) Deletion Backdoor Detection (cont.)



Show Table

Answer-Set Programming

Experiments

Some Experiments (**Horn**, **no-DC**)

domain	instance set (#instances)	disj.	#atoms	horn bd(%)	no-dc bd(%)
AI	`HanoiTower (60)`	–	32956.7	4.28	4.28
	`StrategicCompanies (15)`	+	2002.0	6.03	**
	`MinimalDiagnosis (551)`	+	111856.5	18.34	10.74
Graph	`GraphColoring (60)`	–	3544.4	19.47	19.47
Planning	`MSS/MUS (38)`	+	49402.3	3.80	2.20
	`ConformantPlanning (24)`	+	1378.2	8.43	6.95
Cryptography	`Factoring (10)`	–	3336.8	25.76	1.65
Puzzle	`Labyrinth (261)`	–	55604.9	3.42	1.15
	`KnightTour (10)`	–	23156.9	33.08	25.36
	`Solitaire (25)`	–	11486.8	38.88	20.47
Random	`RandomQBF (15)`	+	160.1	49.69	50.31
	`RLP (572)`	–	174.7	68.89	**
	`RandomNonTight (220)`	–	200.0	89.85	**

Some Experiments (**Horn**, **no-DC**)

domain	instance set (#instances)	disj.	#atoms	horn bd(%)	no-dc bd(%)
AI	`HanoiTower (60)`	–	32956.7	4.28	4.28
	`StrategicCompanies (15)`	+	2002.0	6.03	**
	`MinimalDiagnosis (551)`	+	111856.5	18.34	10.74
Graph	`GraphColoring (60)`	–	3544.4	19.47	19.47
Planning	`MSS/MUS (38)`	+	49402.3	3.80	2.20
	`ConformantPlanning (24)`	+	1378.2	8.43	6.95
Cryptography	`Factoring (10)`	–	3336.8	25.76	1.65
Puzzle	`Labyrinth (261)`	–	55604.9	3.42	1.15
	`KnightTour (10)`	–	23156.9	33.08	25.36
	`Solitaire (25)`	–	11486.8	38.88	20.47
Random	`RandomQBF (15)`	+	160.1	49.69	50.31
	`RLP (572)`	–	174.7	68.89	**
	`RandomNonTight (220)`	–	200.0	89.85	**

Backdoors into tractable classes too large

Parameter Map

But backdoors generalize various previous results

Skip...

Answer-Set Programming

Comparison

Parameters in Literature

Deletion Based

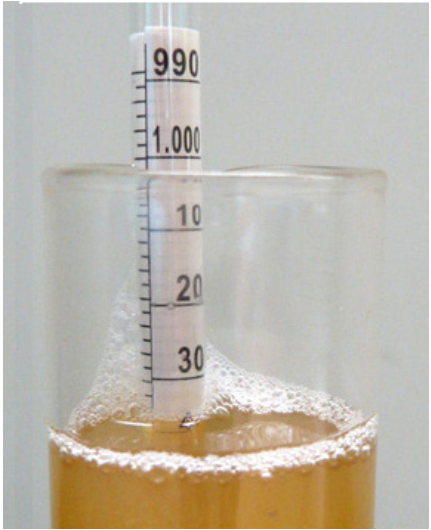
1. Distance from **Horn**
2. Distance from **Strat**

Treewidth Based

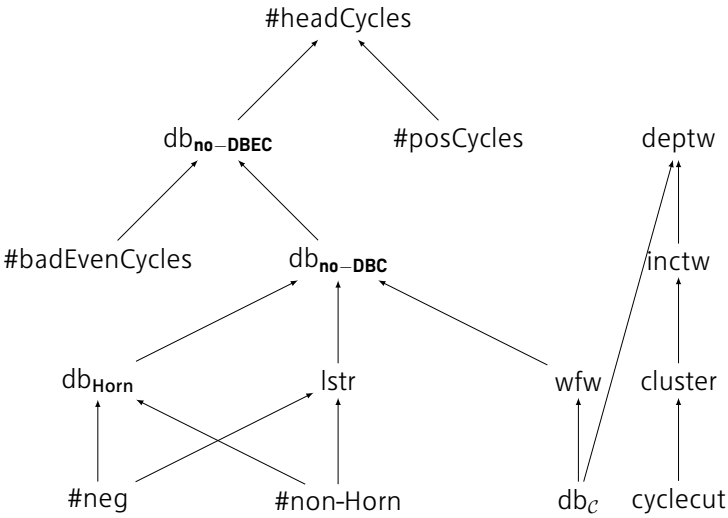
1. Incidence Treewidth
2. Dependency Treewidth
3. Interaction Treewidth

Cycles

1. Number of bad even cycles
2. Number of positive cycles
3. Head-cycles



Parameter Map



Answer-Set Programming

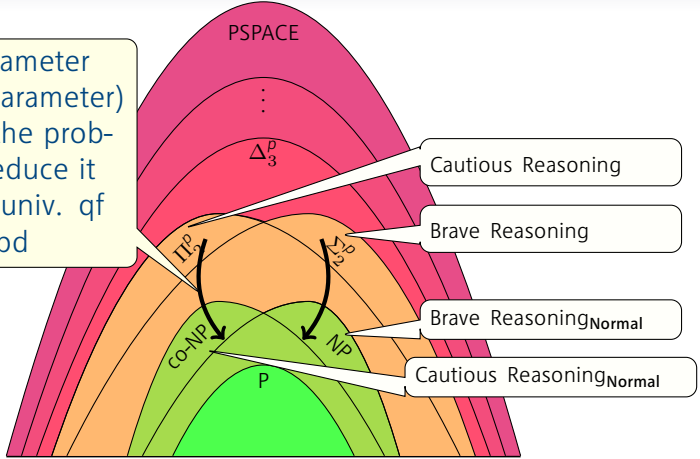
Backdoors Beyond NP 2 NP (or the case when paraNP-c is actually good)

Answer-Set Programming

Backdoor Evaluation

Backdoor Evaluation (NP Classes FS12,13,15a)

Use Fixed-Parameter Tractability (parameter) not to solve the problem, but to reduce it to QBFSAT! #univ. qf bounded by bd

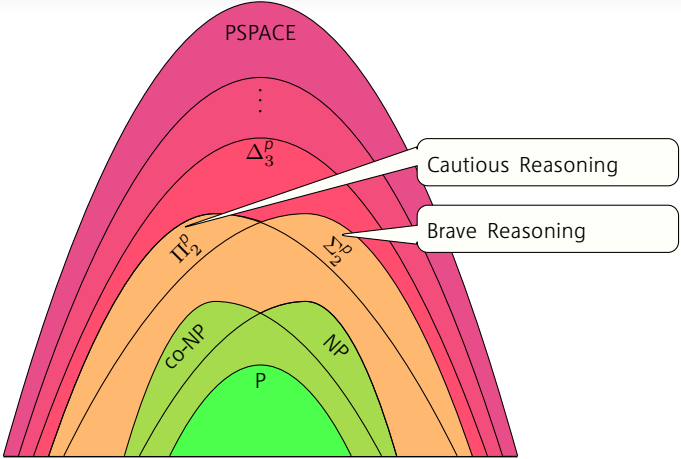


Skip...

Answer-Set Programming

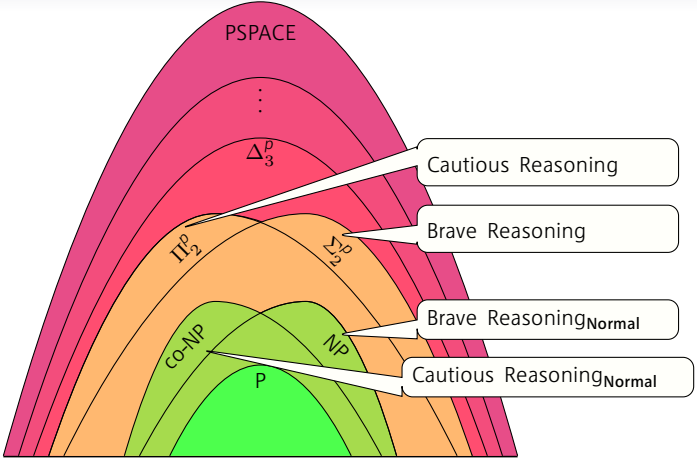
Backdoor Evaluation

Backdoor Evaluation (NP Classes FS12,13,15a)



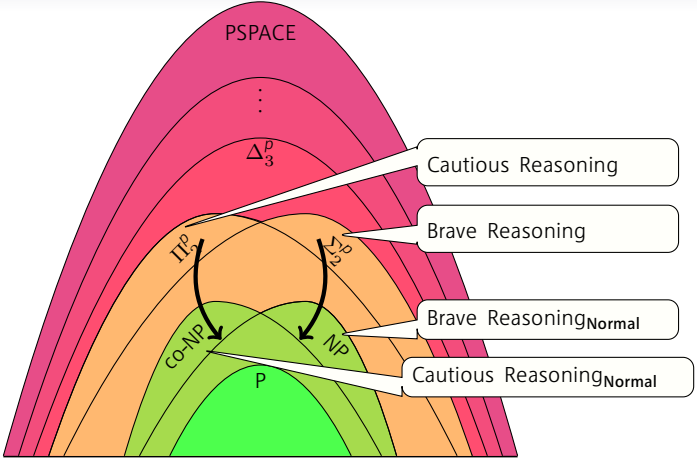
▸ Construction / Proof Idea
▸ Main Theorem

Backdoor Evaluation (NP Classes FS12,13,15a)



▸ Construction / Proof Idea
▸ Main Theorem

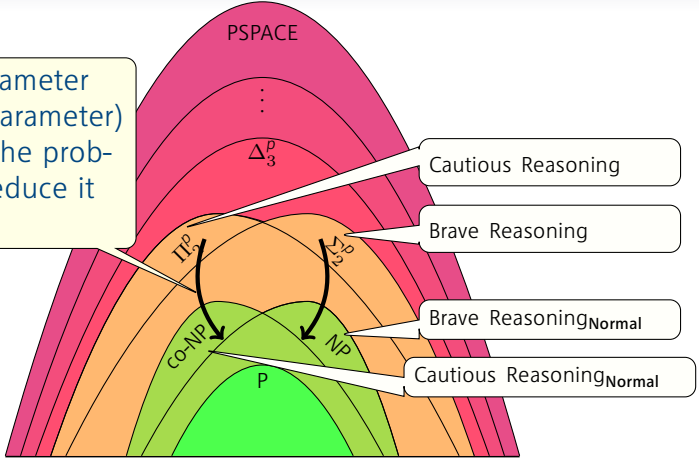
Backdoor Evaluation (NP Classes FS12,13,15a)



▸ Construction / Proof Idea
▸ Main Theorem

Backdoor Evaluation (NP Classes FS12,13,15a)

Use Fixed-Parameter Tractability (parameter) not to solve the problem, but to reduce it to QBFSAT!



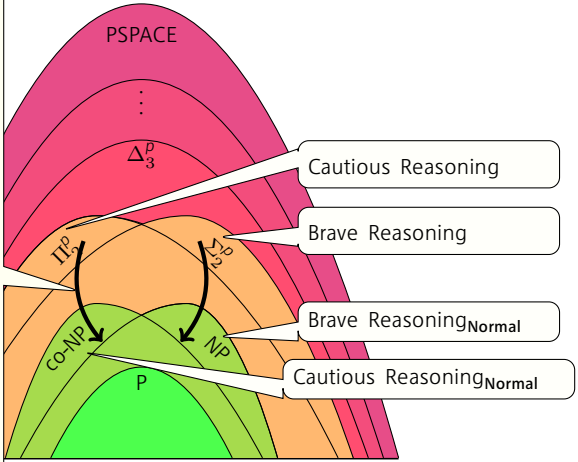
▸ Construction / Proof Idea
▸ Main Theorem

Backdoor Evaluation (NP Classes FS12,13,15a)

Break problem reasonably down:

- Take advantage of QBF and use QBF solver as a black box
- Ptime reduction to SAT wont work unless PH collapses; Explosion confined to the size k of a backdoor,
- Ltime-reduction from ASP-reasoning to QBF; #univ qf bounded by size of the bd

⇒ Compact encoding; Reduction to NP-c problem; guarantee that blowup confined in bd;

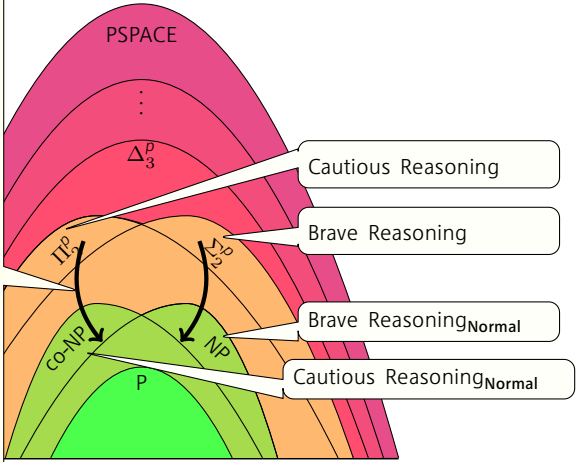


Backdoor Evaluation (NP Classes FS12,13,15a)

Break problem reasonably down:

- Take advantage of QBF and use QBF solver as a black box
- Ptime reduction to SAT wont work unless PH collapses; Explosion confined to the size k of a backdoor,
- Ltime-reduction from ASP-reasoning to QBF; #univ qf bounded by size of the bd

⇒ Compact encoding; Reduction to NP-c problem; guarantee that blowup confined in bd;

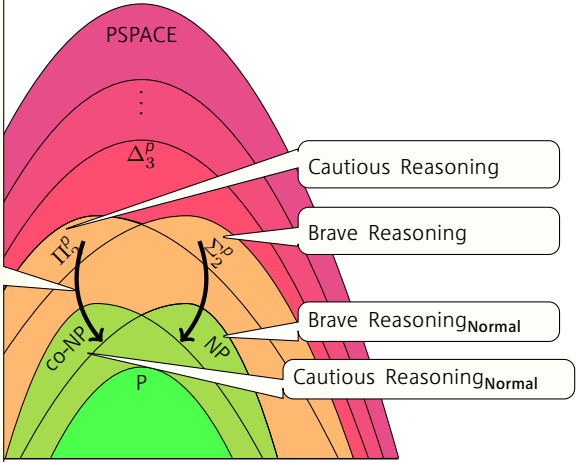


Backdoor Evaluation (NP Classes FS12,13,15a)

Break problem reasonably down:

- Take advantage of QBF and use QBF solver as a black box
- Ptime reduction to SAT wont work unless PH collapses; Explosion confined to the size k of a backdoor,
- Ltime-reduction from ASP-reasoning to QBF; #univ qf bounded by size of the bd

⇒ Compact encoding; Reduction to NP-c problem; guarantee that blowup confined in bd;

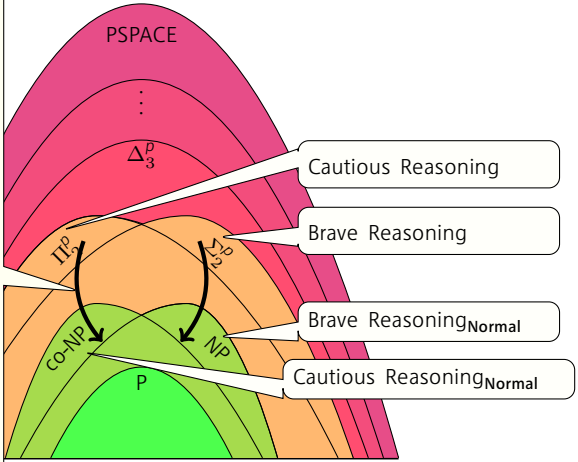


Backdoor Evaluation (NP Classes FS12,13,15a)

Break problem reasonably down:

- Take advantage of QBF and use QBF solver as a black box
- Ptime reduction to SAT wont work unless PH collapses; Explosion confined to the size k of a backdoor,
- Ltime-reduction from ASP-reasoning to QBF; #univ qf bounded by size of the bd

⇒ Compact encoding; Reduction to NP-c problem; guarantee that blowup confined in bd;

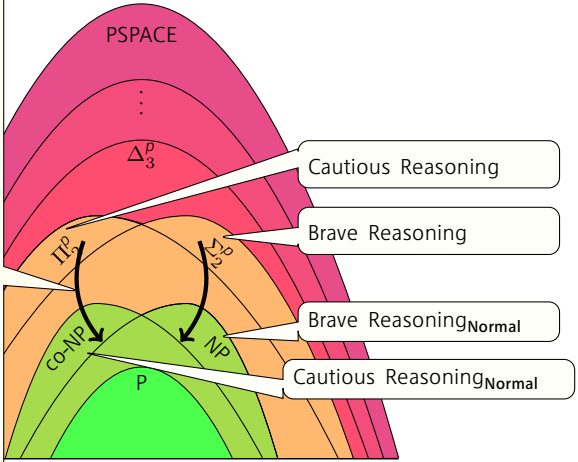


Backdoor Evaluation (NP Classes FS12,13,15a)

Break problem reasonably down:

- Take advantage of QBF and use QBF solver as a black box
- Ptime reduction to SAT wont work unless PH collapses; Explosion confined to the size k of a backdoor,
- Ltime-reduction from ASP-reasoning to QBF; #univ qf bounded by size of the bd

$\Rightarrow$ Compact encoding; Reduction to NP-c problem; guarantee that blowup confined in bd;



Answer-Set Programming

Implications

Implications

Improved approach to use QBF to solve ASP

Problem in Practice

- Our construction produces many aux. variables
- Solvers cannot deal with it very well
- Ongoing: replace parts by propagators

Implications

Improved approach to use QBF to solve ASP

Problem in Practice

- Our construction produces many aux. variables
- Solvers cannot deal with it very well
- Ongoing: replace parts by propagators

Answer-Set Programming

Experiments

Experiments: Size of Backdoors into **Normal**

domain	instance set (#instances)	disj.	#atoms	horn bd(%)	normal bd (%)
AI	`StrategicCompanies (15)`	+	2002.0	6.03	6.03
	`MinimalDiagnosis (551)`	+	111856.5	10.74	10.74
Planning	`MSS/MUS (38)`	+	49402.3	3.80	1.90
	`ConformantPlanning (24)`	+	1378.2	8.43	0.69
Random	`RandomQBF (15)`	+	160.1	49.69	49.69

Answer-Set Programming

Implications

Implications

Improved approach to use QBF to solve ASP
Smaller Backdoors (can verify this experimentally)

Problem in Practice

- Our construction produces many aux. variables
- Solvers cannot deal with it very well
- Backlog: replace parts by propagators

Implications

Improved approach to use QBF to solve ASP
Smaller Backdoors (can verify this experimentally)

Problem in Practice

- Our construction produces many aux. variables
- Solvers cannot deal with it very well
- Backlog: replace parts by propagators