UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Algorithmic Metatheorems 2.0

Till Tantau

Theorietag in Speyer, September 2015

IM FOCUS DAS LEBEN

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

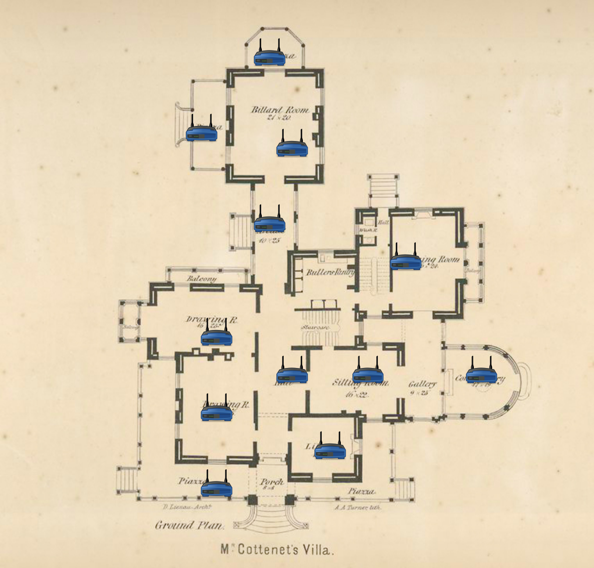
New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

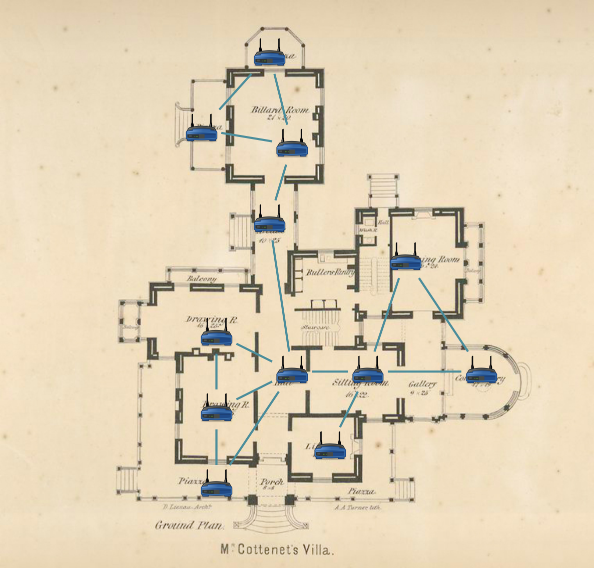
- New Theorems
- Applications: Visible Pushdown Languages

A Well-Known Problem



- We install WLAN routers in a home.

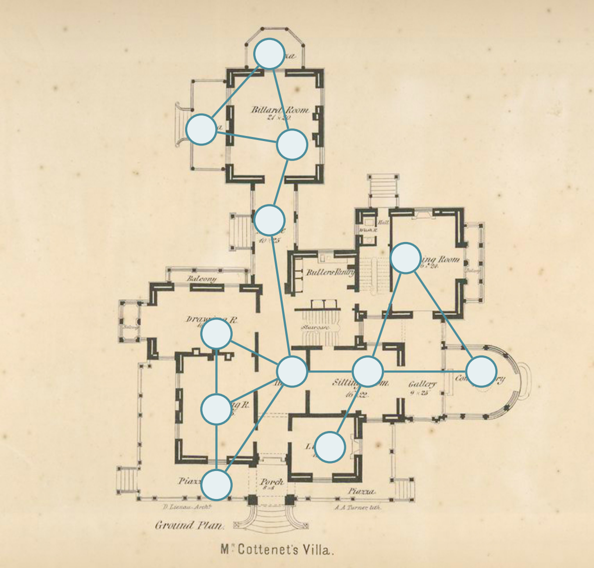
A Well-Known Problem



Public Domain

- We install WLAN routers in a home.
- Adjacent routers interfere.

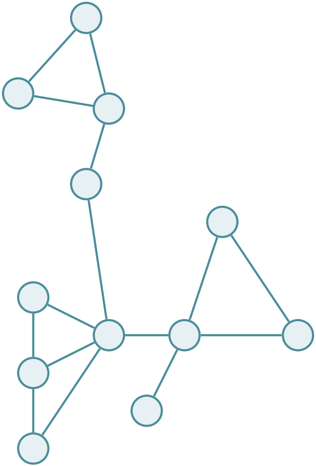
A Well-Known Problem



Public Domain

- We install WLAN routers in a home.
- Adjacent routers interfere.

A Well-Known Problem



- We install WLAN routers in a home.
- Adjacent routers interfere.
- The abstract problem is of course *3-colorability*.

3-Colorability is Hard.

- 3-Colorability is a classical NP-complete problem.
- Assuming P $\neq$ NP, it *cannot* be solved efficiently.

However

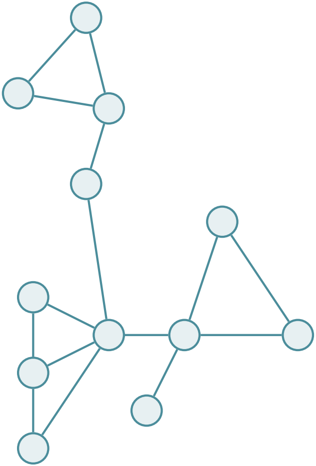
What happens, when the input has a *tree-like decomposition* and we can apply *divide and conquer*?

Divide and Conquer, First Attempt

Entzwei und gebiete!
Tüchtig Wort. –
Verein und leite!
Besserer Hort.

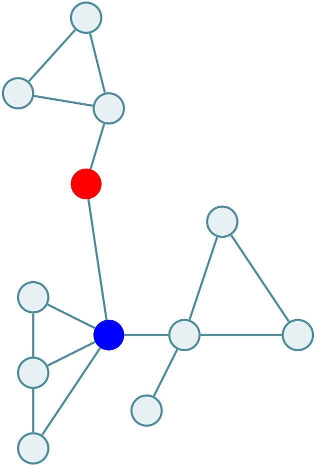
Johann Wolfgang Goethe

Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

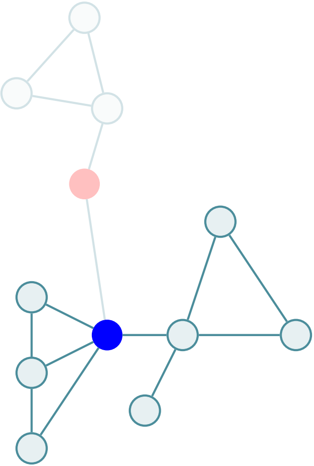
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.

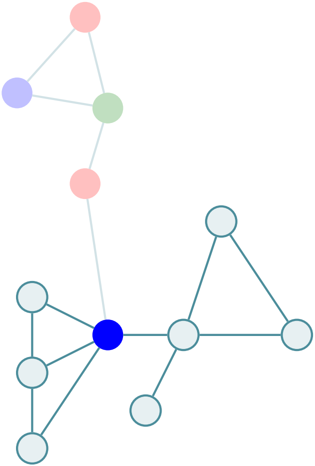
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph

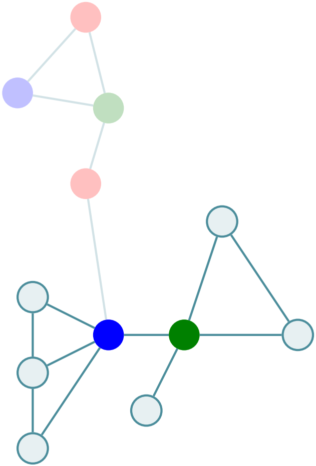
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph
- and we can solve the problem *recursively* on the *independent* parts.

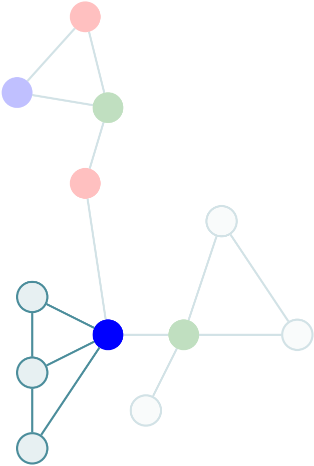
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph
- and we can solve the problem *recursively* on the *independent* parts.

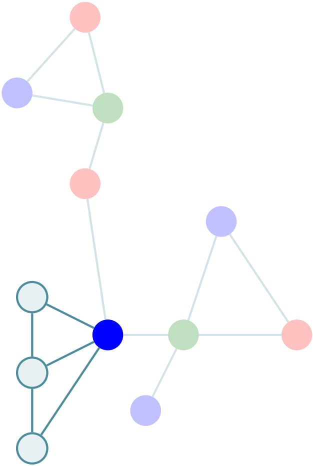
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph
- and we can solve the problem *recursively* on the *independent* parts.

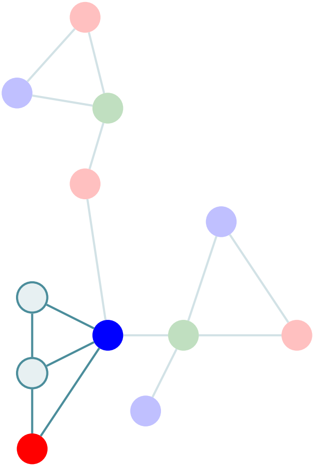
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph
- and we can solve the problem *recursively* on the *independent* parts.

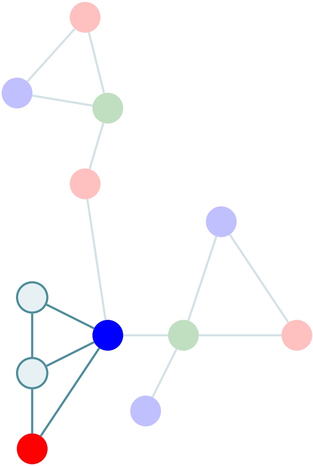
Divide and Conquer, First Attempt



Sometimes divide and conquer can be applied to 3-colorability.

- Pick *two appropriate* vertices and consider the 6 possible colorings.
- A vertex *screens* part of the graph
- and we can solve the problem *recursively* on the *independent* parts.
- It can happen that we have to *remember several vertices* during the recursion.

Divide and Conquer, First Attempt



A *game* describes how we can proceed.

Robbers and Coppers

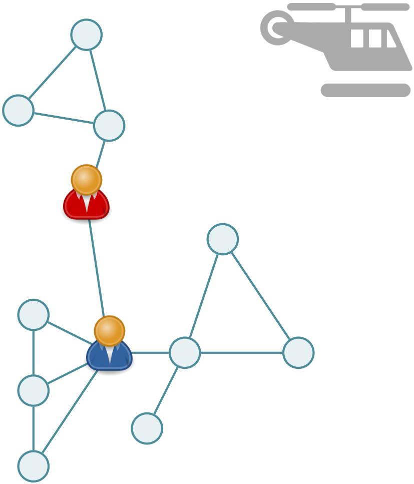
Game Objective

The *k cops* try to catch a *robber* by being on the *same vertex* as her.

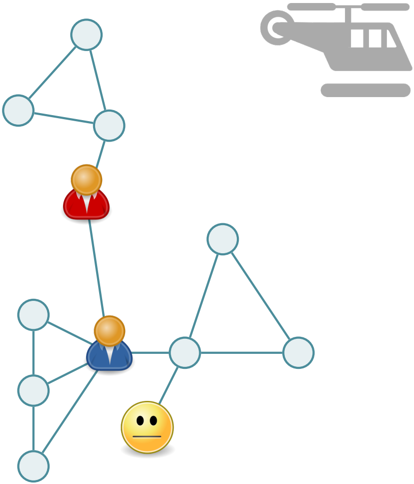
Game Rules

1. First the cops, then the robber pick a start vertex.
2. A cop gets on a *helicopter* and heads towards some vertex.
3. *Meanwhile* the robber moves along *unoccupied vertices*.

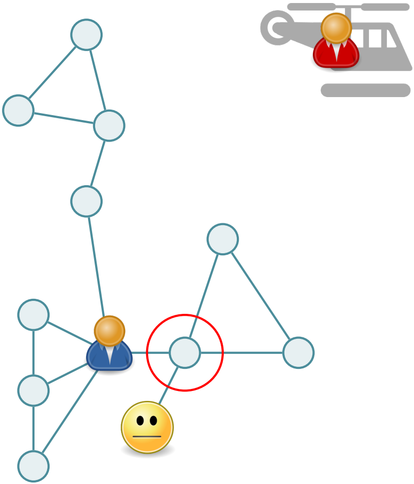
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



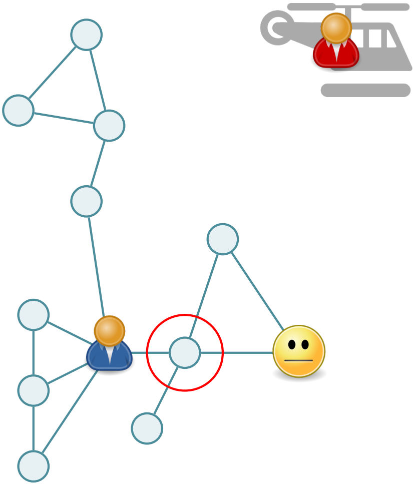
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



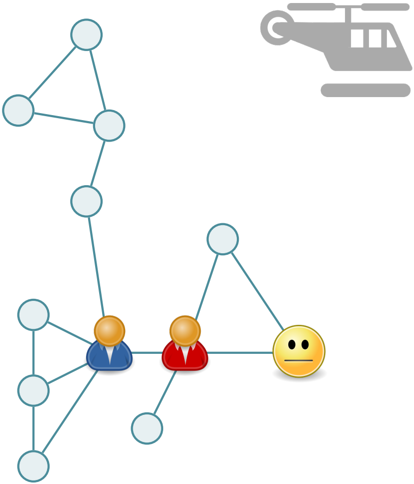
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



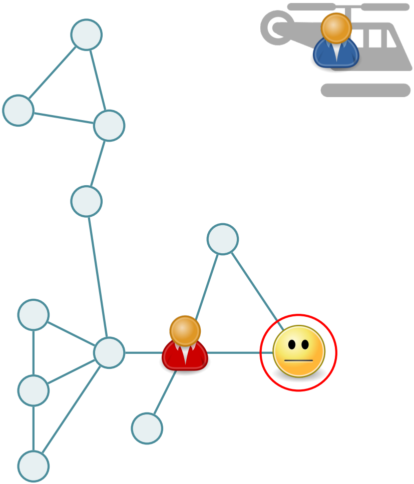
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



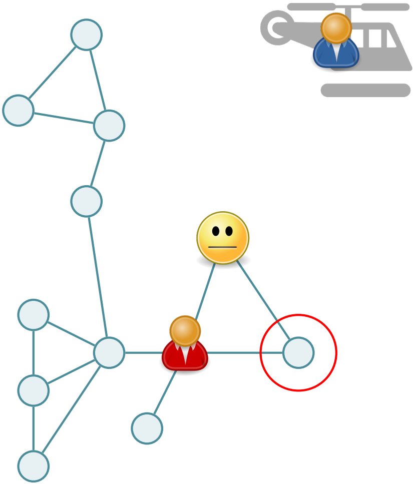
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



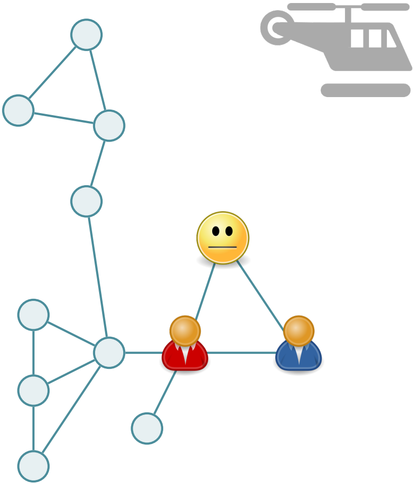
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



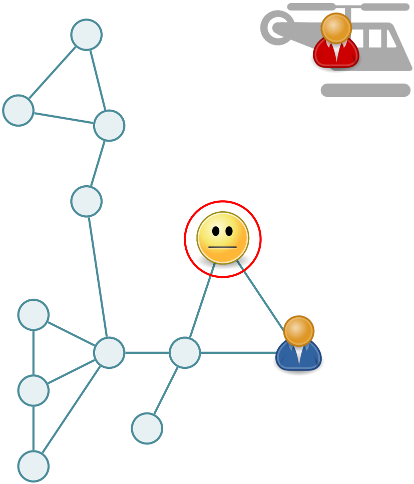
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



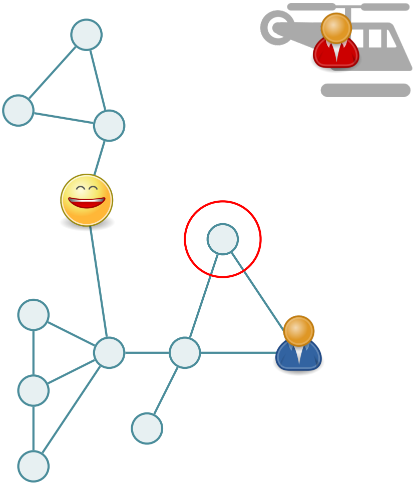
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



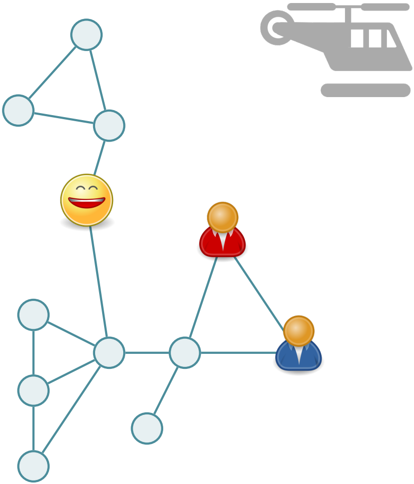
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



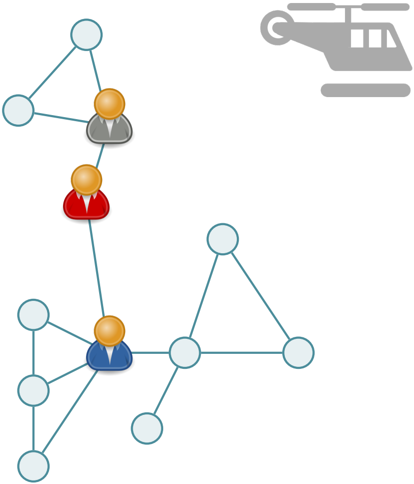
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



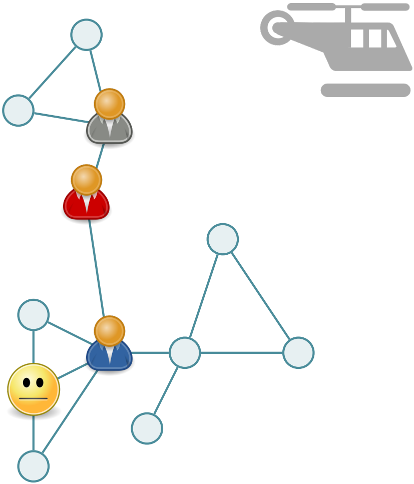
Robbers and Coppers:
Two Cops Do Not Catch the Robber.



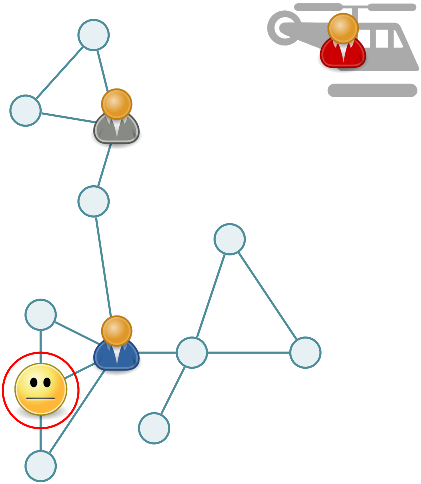
Robbers and Coppers:
Three Cops Always Catch the Robber.



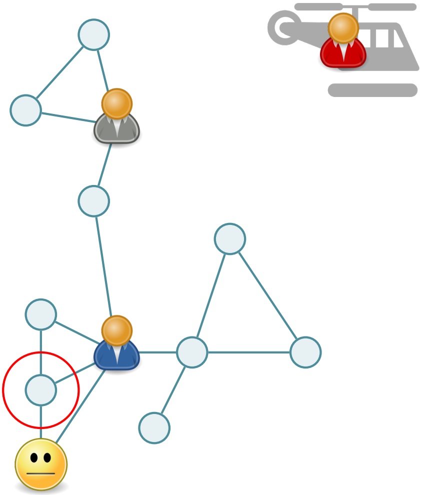
Robbers and Coppers:
Three Cops Always Catch the Robber.



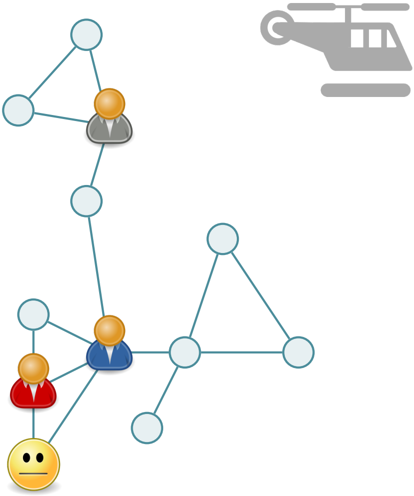
Robbers and Coppers:
Three Cops Always Catch the Robber.



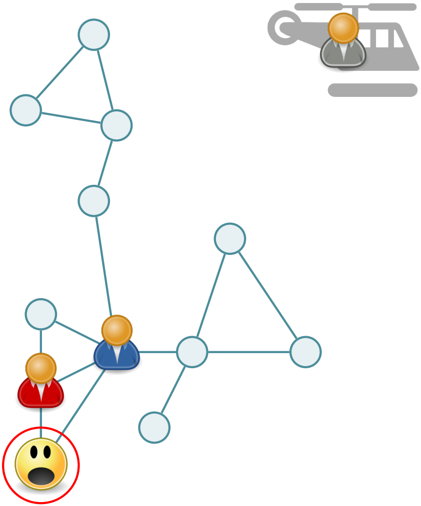
Robbers and Coppers:
Three Cops Always Catch the Robber.



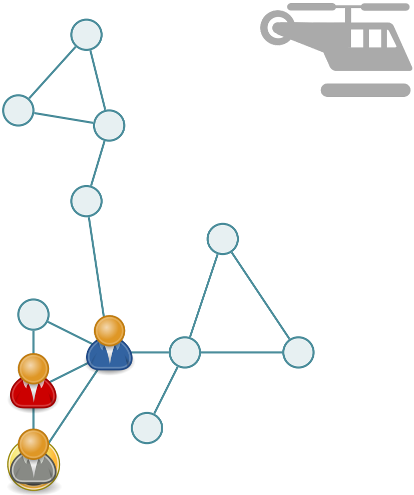
Robbers and Coppers:
Three Cops Always Catch the Robber.



Robbers and Coppers:
Three Cops Always Catch the Robber.



Robbers and Coppers:
Three Cops Always Catch the Robber.

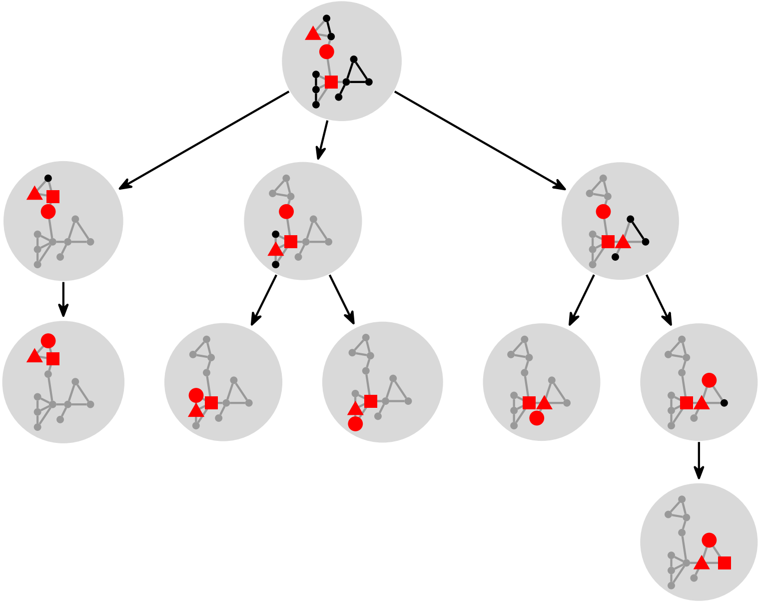


The Cops's Strategy = Tree Decomposition

The *winning strategy of the cops* forms a *tree*:

- The *nodes* are positions of the cops.
- The *root* is their initial position.
- The *children* of a tree node are the *graph components that could contain the robber*.

Example of a Tree Decomposition of Width 2.



Divide and Conquer, Second Attempt

Theorem

For a graph G let a tree decomposition of width k with n nodes be given. Then we can decide in time

$$O(3^{k+1}n)$$

whether G is 3-colorable.

Divide and Conquer, Second Attempt

Theorem

For a graph G let a tree decomposition of width k with n nodes be given. Then we can decide in time

$$O(3^{k+1}n)$$

whether G is 3-colorable.

Theorem

For a graph G let a tree decomposition of width k with n nodes be given. Then we can compute in time

$$O(2^{k+1}n)$$

the largest independent set of G.

Why We Need *Meta* theorems.

Theorem 4.4

Each of the following problems is in NC, when restricted to graphs with treewidth $\leq K$, for constant K: vertex cover [GT1], dominating set [GT2], domatic number [GT3], chromatic number [GT4], monochromatic triangle [GT5], feedback vertex set [GT7], feedback arc set [GT8], partial feedback edge set [GT9], minimum maximal matching [GT10], partition into triangles [GT11], partition into isomorphic subgraphs for fixed H [GT12], partition into Hamiltonian subgraphs [GT13], partition into forests [GT14], partition into cliques [GT15], partition into perfect matchings [GT16], clique [GT19], independent set [GT20], induced subgraph with property P (for monadic second order properties P) [GT21], induced connected subgraph with property P (for monadic second order properties P) [GT22], induced path [GT23], balanced complete bipartite subgraph [GT24], bipartite subgraph [GT25], degree bounded connected subgraph for fixed d [GT26], planar subgraph [GT27], transitive subgraph [GT29], uniconnected subgraph [GT30], minimum k-connected subgraph for fixed k [GT31], cubic subgraph [GT32], minimum equivalent digraph [GT33], Hamiltonian completion [GT34], Hamiltonian circuit [GT37], directed Hamiltonian circuit [GT38], Hamiltonian path (and directed Hamiltonian path) [GT39], subgraph isomorphism for fixed H, subgraph isomorphism for connected H with bounded valence [GT 48], graph contractability for fixed H [GT51], graph homomorphism for fixed H [GT52], path with forbidden pairs for fixed n [GT54], multiple choice matching for fixed J [GT55], graph grundy numbering for graphs with bounded valence [GT56], kernel [GT57], k-closure [GT58], path distinguishers [GT60], degree constrained spanning tree [ND1], maximum leaf spanning tree [ND2], bounded diameter spanning tree [ND3], k'th best spanning tree for fixed k [ND9], bounded component spanning forest for fixed k [ND10], multiple choice branching for fixed m [ND11], Steiner tree in graphs [ND12], max cut [ND16], minimum cut into bounded sets [ND17], rural postman [ND27], longest circuit [ND28], longest path [ND29], shortest weight-constrained path [ND30], k'th shortest path for fixed k [ND31], disjoint connecting paths for fixed k [ND40], maximum length-bounded disjoint paths for fixed J [ND41], maximum fixed-length disjoint paths for fixed J [ND42], chordal graph completion for fixed k, chromatix index, spanning tree parity problem, distance d chromatic number for fixed d and k, thickness $\leq k$ for fixed k, membership for each class C of graphs, which is closed unded minor taking.

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

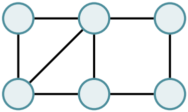
New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

All Problems Share one Property.

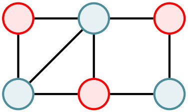
In 1990, Courcelle noted that all of the problems can be described in *monadic second order logic* (MSO logic).

A Logical Formula Specifying 3-Colorability.



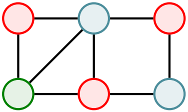
$$\models^? \exists R\, \exists G\, \exists B\, \forall x \forall y (E(x,y) \to$$
$$((R(x) \land \neg G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land \neg G(x) \land B(x))) \land$$
$$\neg(R(x) \land R(y)) \land$$
$$\neg(G(x) \land G(y)) \land$$
$$\neg(B(x) \land B(y)))$$

A Logical Formula Specifying 3-Colorability.



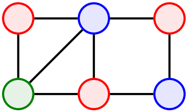
$$\models^? \exists R\, \exists G\, \exists B\, \forall x \forall y (E(x,y) \to$$
$$((R(x) \land \neg G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land \neg G(x) \land B(x))) \land$$
$$\neg(R(x) \land R(y)) \land$$
$$\neg(G(x) \land G(y)) \land$$
$$\neg(B(x) \land B(y)))$$

A Logical Formula Specifying 3-Colorability.



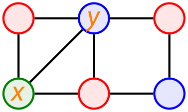
$$\models^? \exists R\, \exists G\, \exists B\, \forall x \forall y (E(x,y) \to$$
$$((R(x) \land \neg G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land \neg G(x) \land B(x))) \land$$
$$\neg(R(x) \land R(y)) \land$$
$$\neg(G(x) \land G(y)) \land$$
$$\neg(B(x) \land B(y)))$$

A Logical Formula Specifying 3-Colorability.



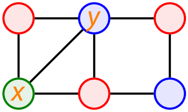
$$\models^{?} \exists R\, \exists G\, \exists B\, \forall x \forall y (E(x,y) \to$$
$$((R(x) \land \neg G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land G(x) \land \neg B(x)) \lor$$
$$(\neg R(x) \land \neg G(x) \land B(x))) \land$$
$$\neg(R(x) \land R(y)) \land$$
$$\neg(G(x) \land G(y)) \land$$
$$\neg(B(x) \land B(y)))$$

A Logical Formula Specifying 3-Colorability.



$$\models \quad \exists R\, \exists G\, \exists B\, \forall x \forall y (E(x,y) \to \\ ((R(x) \land \neg G(x) \land \neg B(x)) \lor \\ (\neg R(x) \land G(x) \land \neg B(x)) \lor \\ (\neg R(x) \land \neg G(x) \land B(x))) \land \\ \neg(R(x) \land R(y)) \land \\ \neg(G(x) \land G(y)) \land \\ \neg(B(x) \land B(y)))$$

A Logical Formula Specifying 3-Colorability.



$$\models \quad \underbrace{\exists R\, \exists G\, \exists B\, \forall x \forall y}_{\text{Pattern } E_1 E_1 E_1 aa}(E(x,y) \to$$
$$((R(x) \wedge \neg G(x) \wedge \neg B(x)) \vee$$
$$(\neg R(x) \wedge G(x) \wedge \neg B(x)) \vee$$
$$(\neg R(x) \wedge \neg G(x) \wedge B(x))) \wedge$$
$$\neg(R(x) \wedge R(y)) \wedge$$
$$\neg(G(x) \wedge G(y)) \wedge$$
$$\neg(B(x) \wedge B(y)))$$

Classical Algorithmic Metatheorems.

Theorem (Courcelle)

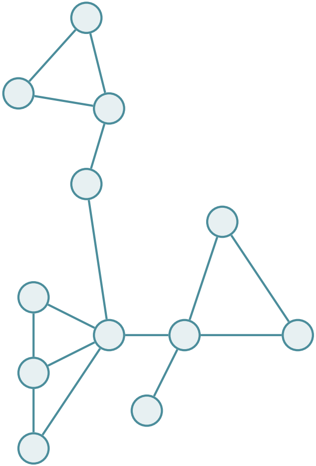
Let φ be an MSO formula and k a number. Then

$$\{G \mid G \models \varphi \text{ and } G \text{ has tree width at most } k\}$$

can be decided in linear time.

Proof Sketch

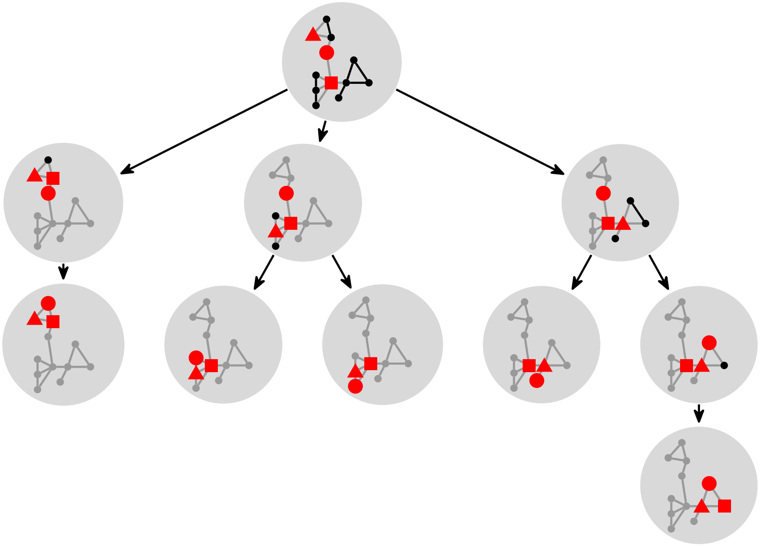
Step 1: The original question.



$\models \varphi_{\text{3-colorable}}$?

Proof Sketch

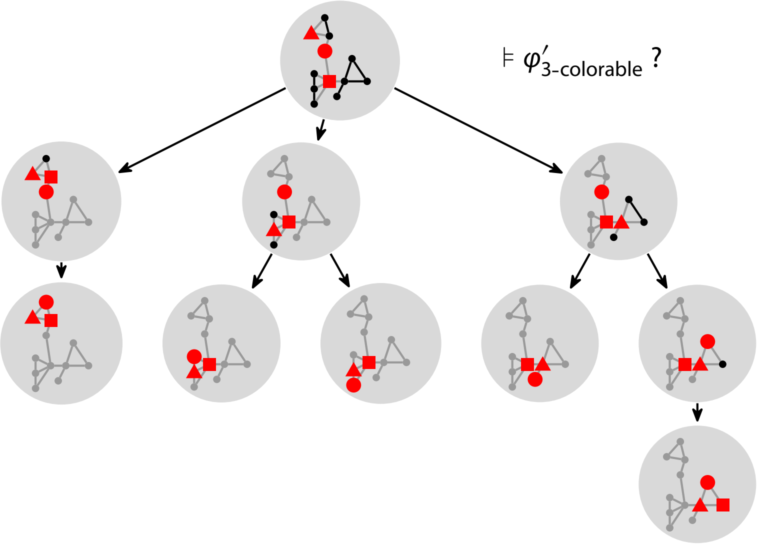
Step 2: Compute a tree decomposition in linear time.



Proof Sketch

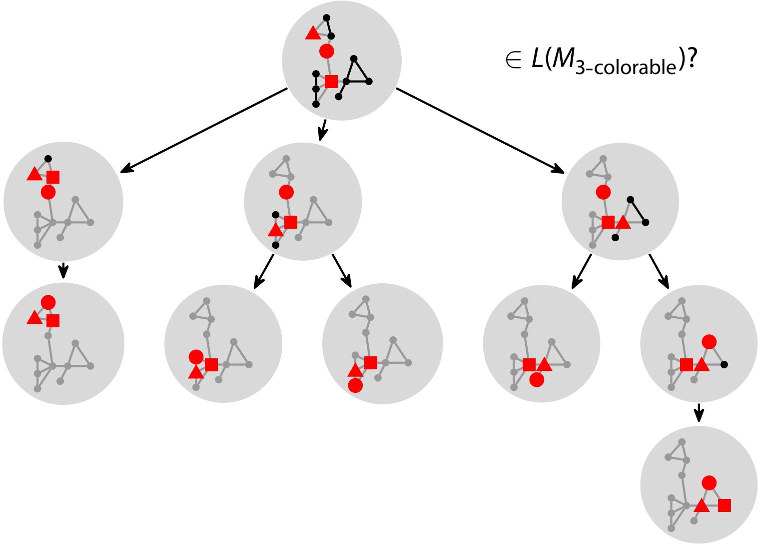
Step 3: Adjust the formula so that it applies to the tree.

$\models \varphi'_{\text{3-colorable}}$?



Proof Sketch

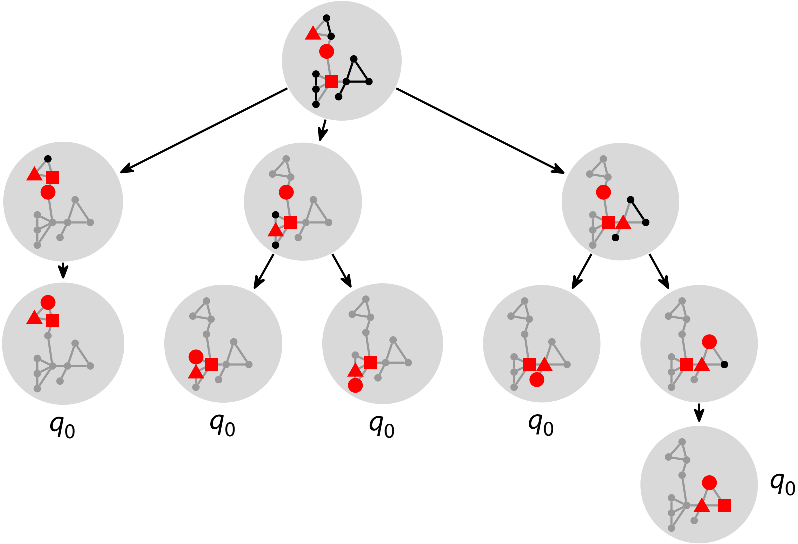
Step 4: Transform the formula into a tree automaton.



$\in L(M_{\text{3-colorable}})$?

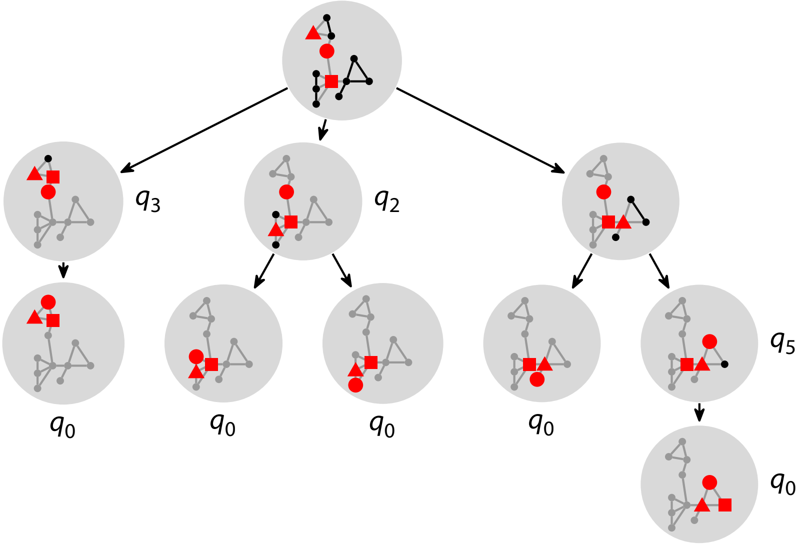
Proof Sketch

Step 5: Evaluate that automaton in linear time.



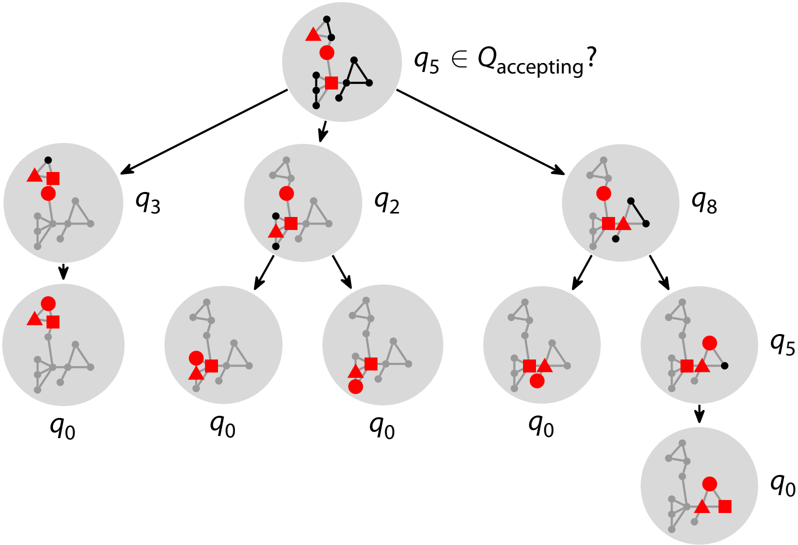
Proof Sketch

Step 5: Evaluate that automaton in linear time.



Proof Sketch

Step 5: Evaluate that automaton in linear time.



Classical Algorithmic Metatheorems.

Theorem (The General Pattern)

If

- *the problem can be described in a certain logic*
- *and the input can be decomposed in a certain way,*

then

- *there is a certain kind of algorithm for it.*

Classical Algorithmic Metatheorems.

Theorem (Courcelle)

If

- *the problem can be described in MSO logic*
- *and the input has tree width at most k,*

then

- *there is a linear time of algorithm for it.*

Classical Algorithmic Metatheorems.

Theorem (Bodlaender, Courcelle)

If

- *the problem can be described in MSO logic*
- *and the input has tree width at most k,*

then

- *there is a parallel algorithm running in time $O(\log n)$.*

Classical Algorithmic Metatheorems.

Theorem (Courcelle, Makowsky, Rotics)

If

- *the problem can be described in MSO logic*
- *and the input has clique width at most k,*

then

- *there is a polynomial time algorithm for it.*

Classical Algorithmic Metatheorems.

Theorem (Frick, Grohe)

If

- *the problem can be described in FO logic*
- *and the input is planar,*

then

- *there is a linear time algorithm for it.*

Classical Algorithmic Metatheorems.

Theorem (Flum, Grohe)

If

- *the problem can be described in FO logic*
- *and the input has a forbidden minor,*

then

- *there is a polynomial time algorithm for it.*

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

Beyond Time

- While the classical theorems yield *tight upper time* bounds…
- …they yield *no* completeness results.
- The new metatheorems concern
  - *space complexity* and
  - *circuit complexity*
- and yield *completeness results*.

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

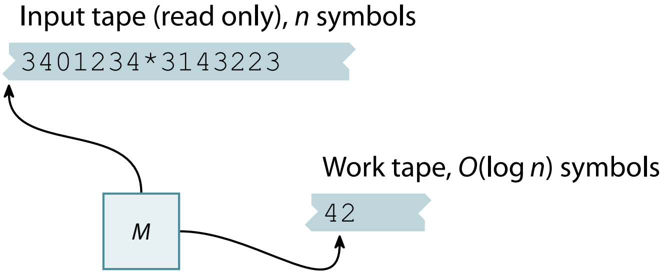
New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

A Logspace Turing Machine



The Logspace Version of Courcelle's Theorem.

Theorem (Elberfeld, T, Jakoby, 2010)

Let φ be an MSO formula and k a number. Then

$$\{G \mid G \models \varphi \text{ and } G \text{ has tree width at most } k\}$$

can be decided in logspace.

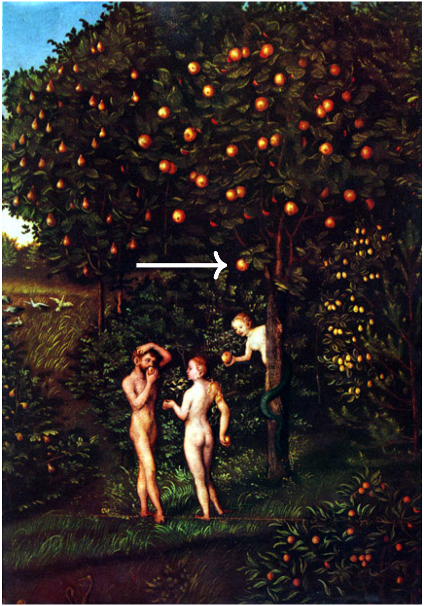
The Low-Hanging Fruits.

Corollary

For all k, we can solve in logspace:

- $\{G \mid G$ *is 3-colorable and G has tree width at most k*$\}$.
- $\{G \mid G$ *has a perfect matching and G has tree width at most k*$\}$.
- $\{(G, s, t) \mid t$ *is reachable from s and G has tree width at most k*$\}$.
- …

(Each result used to be a paper.)



Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

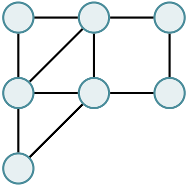
The High-Hanging Fruits.

Metatheorems make statements about *decomposable* graphs.

For *arbitrary* graph, this may work:

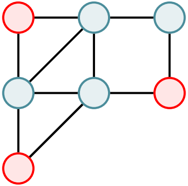
- If the graph *is* decomposable, apply the metatheorem.
- If the graph *is not* decomposable, it must have *many edges*, which we may *use algorithmically*.

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



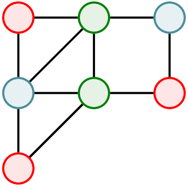
$$\models^? \exists R \exists G \exists B\, \forall x \exists y (E(x,y) \land \\
((R(x) \land \neg G(x) \land \neg B(x)) \lor \\
(\neg R(x) \land G(x) \land \neg B(x)) \lor \\
(\neg R(x) \land \neg G(x) \land B(x))) \land \\
(R(x) \to G(y)) \land \\
(G(x) \to B(y)) \land \\
(B(x) \to R(y)))$$

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



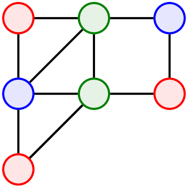
$$\models^? \exists R\, \exists G\, \exists B\, \forall x \exists y (E(x,y) \land \\
((R(x) \land \neg G(x) \land \neg B(x)) \lor \\
(\neg R(x) \land G(x) \land \neg B(x)) \lor \\
(\neg R(x) \land \neg G(x) \land B(x))) \land \\
(R(x) \to G(y)) \land \\
(G(x) \to B(y)) \land \\
(B(x) \to R(y)))$$

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



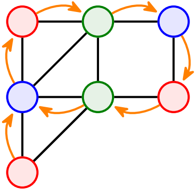
$$
\begin{aligned}
\models^{?}\ \exists R\, \exists G\, \exists B\, \forall x \exists y (E(x,y) \land{} \\
&((R(x) \land \neg G(x) \land \neg B(x)) \lor \\
&\ (\neg R(x) \land G(x) \land \neg B(x)) \lor \\
&\ (\neg R(x) \land \neg G(x) \land B(x))) \land \\
&(R(x) \to G(y)) \land \\
&(G(x) \to B(y)) \land \\
&(B(x) \to R(y)))
\end{aligned}
$$

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



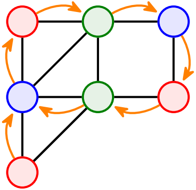
$$\models^? \exists R \exists G \exists B\, \forall x \exists y (E(x,y) \land ((R(x) \land \neg G(x) \land \neg B(x)) \lor (\neg R(x) \land G(x) \land \neg B(x)) \lor (\neg R(x) \land \neg G(x) \land B(x))) \land (R(x) \to G(y)) \land (G(x) \to B(y)) \land (B(x) \to R(y)))$$

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



$$\models \quad \exists R\, \exists G\, \exists B\, \forall x \exists y (E(x,y) \wedge \\ ((R(x) \wedge \neg G(x) \wedge \neg B(x)) \vee \\ (\neg R(x) \wedge G(x) \wedge \neg B(x)) \vee \\ (\neg R(x) \wedge \neg G(x) \wedge B(x))) \wedge \\ (R(x) \to G(y)) \wedge \\ (G(x) \to B(y)) \wedge \\ (B(x) \to R(y)))$$

The Formula φ_3 Describes the Existence of Cycles Whose Length is a Multiple of 3.



$$\models \quad \underbrace{\exists R\, \exists G\, \exists B\, \forall x \exists y}_{\text{Pattern } E_1E_1E_1ae}(E(x,y) \wedge$$

$$((R(x) \wedge \neg G(x) \wedge \neg B(x)) \vee$$

$$(\neg R(x) \wedge G(x) \wedge \neg B(x)) \vee$$

$$(\neg R(x) \wedge \neg G(x) \wedge B(x))) \wedge$$

$$(R(x) \to G(y)) \wedge$$

$$(G(x) \to B(y)) \wedge$$

$$(B(x) \to R(y)))$$

Detecting Cycles Whose Length is a Multiple of 3.

Theorem

There is a logspace algorithm that on input of an undirected graph G decides whether G has a cycle whose length is a multiple of 3.

Proof.

1. Check, whether G has small tree width.
2. If yes, subdivide all edges and apply the metatheorem to φ_4.
3. *If no, output "yes".*

   Thomassen has shown that all graphs of sufficiently large tree width have a cycle whose length is a multiple of 3. □

(A similar argument shows EVEN-CYCLE $\in$ L.)

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

Formulas in Existential Second-Order Logic Describe NP.

Theorem (Fagin, 1974)

A problem is in NP *if, and only if,*
it can be described by a formula in existential second-order logic.

Formulas in Existential Second-Order Logic Describe NP.

Theorem (Fagin, 1974)

A problem is in NP *if, and only if, it can be described by a formula in existential second-order logic.*

- However, $E_1 E_1 E_1 aa$ suffices to describe NP-complete problems.
- What about $E_1 E_1 E_1 ae$?
- What about $E_1 E_1 aa$?
- What about $E_1 aa$?
- …

Can we refine Fagin's Theorem?

An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

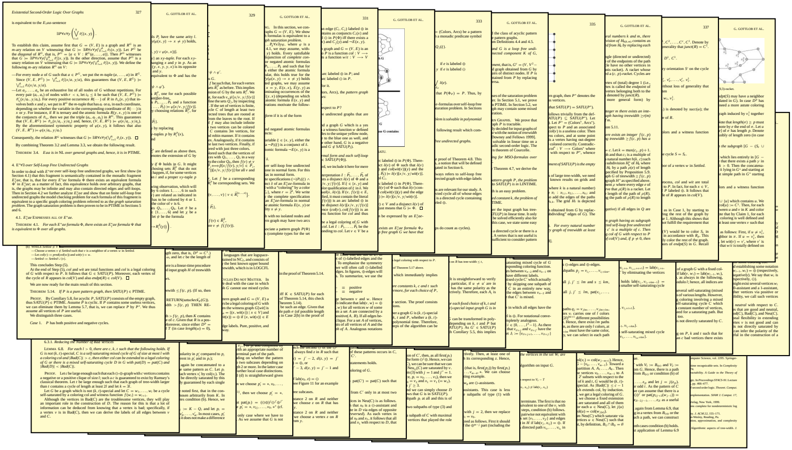
E_1^*ae formulas describe problems in P over undirected loop-free graphs.

An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P *over undirected loop-free graphs.*

Proof.



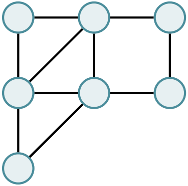
An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P *over undirected loop-free graphs.*

Proof.

Original question:

 $\models^?$ $\exists R \exists G \exists B \forall x \exists y\, (\varphi)$ $\square$

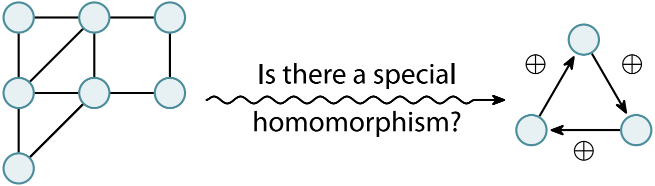
An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P *over undirected loop-free graphs.*

Proof.

New question:



□

An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P over undirected loop-free graphs.

Proof.

1. Preprocess the input graph.
2. Check whether the *tree width is small*.
3. If so, we get a polynomial time algorithm by *Courcelle's Theorem*.
4. If not, depending on the target graph,
   4.1 we can either *just say "yes"* or
   4.2 test whether a special cycle of *constant length* exists.

□

An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P over undirected loop-free graphs.

Proof.

1. Preprocess the input graph.
2. Check whether the *tree width is small*.
3. If so, we get a polynomial time algorithm by *Courcelle's Theorem*.
4. If not, depending on the target graph,
   - 4.1 we can either *just say "yes"* or
   - 4.2 test whether a special cycle of *constant length* exists.

□

Key Insight

All steps can also be done in *logspace*, in particular also *applying Courcelle's Theorem* because of its logspace version.

An Very Hard-To-Prove Upper Bound

Theorem (Gottlob, Kolaitis, Schwentick, 2004)

E_1^*ae formulas describe problems in P over undirected loop-free graphs.

Theorem (T., 2015)

E_1^*ae formulas describe problems in L over undirected loop-free graphs.

Corollary

EVEN-CYCLE $\in$ L.

All Pattern Classes Over Undirected Self-Loop Free Graphs

Theorem

The patterns below yield the classes over undirected self-loop free graphs:

Patterns	Class
$E_{any}^*(ae)^*$; E_1E_1aa, E_1aaa, E_2eaa, E_1eae, E_1aee, E_1aea, E_1aae	NP
E_1e^*aa; E_1eaa	NL
$E_{any}aa$, E_{any}^*ae; E_1aa, E_1E_1ae, E_2ae	L
$(ae)^*$, $E_{any}^*e^*a$, E_1ae	AC^0

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

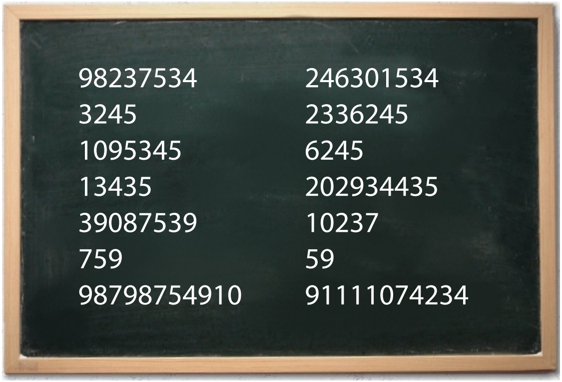
New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

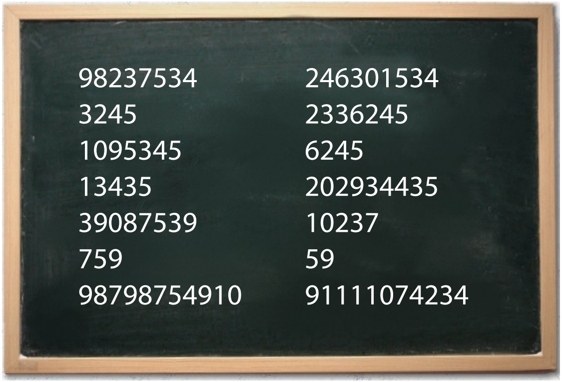
The Subset Sum Problem



The Subset Sum Problem

Circle some numbers so that they add up exactly to 1000000000.

The Subset Sum Problem



The Subset Sum Problem

Circle some numbers so that they add up exactly to 1000000000.

This is a well-known NP-complete problem.

The Subset Sum Problem

The Unary Subset Sum Problem

Cirlce some numbers so that they add up exactly to

.

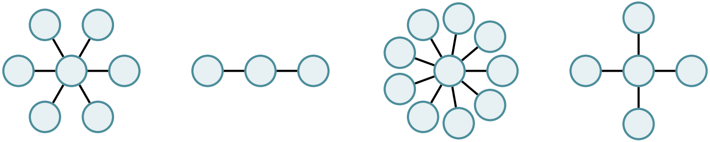
The Unary Version Is In Logspace.

Theorem

UNARY-SUBSET-SUM $\in$ L.

Proof.

For an input like 𝍸II, III, 𝍸𝍸, 𝍸 consider the forest



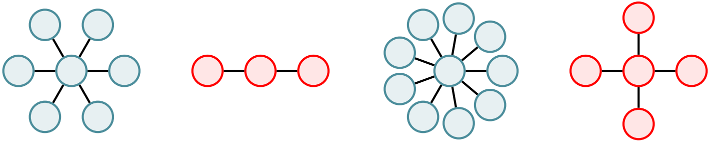
The Unary Version Is In Logspace.

Theorem

UNARY-SUBSET-SUM $\in$ L.

Proof.

For an input like 𝍸𝍷𝍷, 𝍷𝍷𝍷, 𝍸𝍸, 𝍸 consider the forest



1. The size of a set S that is closed under reachability corresponds to a subset sum.
2. The formula $\varphi_{\text{closed}}(X) = \forall u \forall v[(X(u) \wedge E(u,v)) \to X(v)]$ describes "being closed under reachability."
3. New goal: Find out whether φ has a solution of a *certain size*. $\square$

A Stronger Algorithmic Metatheorem

Theorem (Elberfeld, T, Jakoby, 2010)

Let $\varphi(X)$ be an MSO formula with a free variable and let k be a number. Then there is logspace Turing machine that on input of

1. *a graph G of tree width at most k and*
2. *a number s*

computes the number of subsets S with

1. $|S| = s$ *and*
2. $G \models \varphi(S)$.

What We Are Really Interested In: The *Original* Subset Sum Problem.

Let $a_1, \dots, a_n$ be the inputs and s the target sum.

Theorem

Dynamic programming solves the subset sum problem in

- *time $O(ns)$ and*
- *space $O(s)$.*

For larger s, *space* is the bottleneck: 4GB of memory are filled in *one second* for $s = 1{,}000{,}000{,}000$.

What We Are Really Interested In: The *Original* Subset Sum Problem.

Let $a_1, \dots, a_n$ be the inputs and s the target sum.

Theorem

Dynamic programming solves the subset sum problem in

- *time $O(ns)$ and*
- *space $O(s)$.*

Theorem (Lokshtanov, Nederlof, 2010)

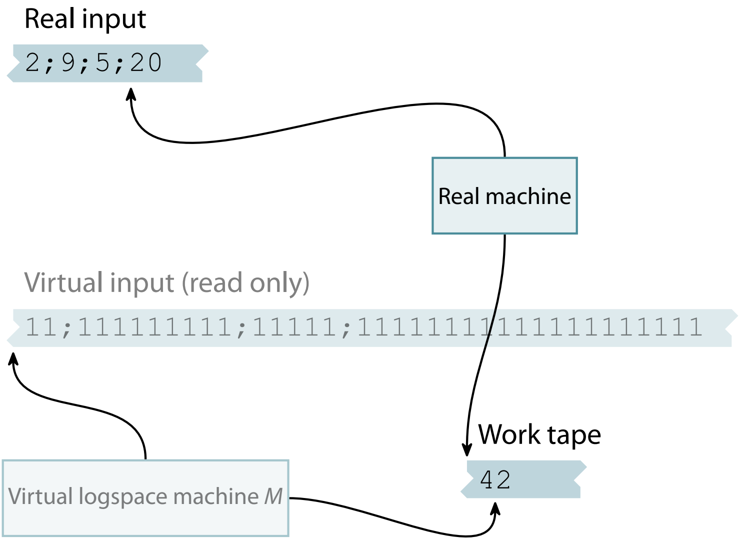
The subset sum problem can be solved in

- *time $n^{O(1)} s^{O(1)}$ and*
- *space $n^{O(1)}$.*

Proof.

Complex analysis of approximation circuits for Fourier transformations. $\square$

A Magic Trick.



A Magic Trick.

A similar argument shows that there are *space efficient* pseudo-polynomial time algorithms for:

- knapsack problems,
- bin packing problems,
- scheduling problems, and
- integer programming for a fixed number of inequalities.

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

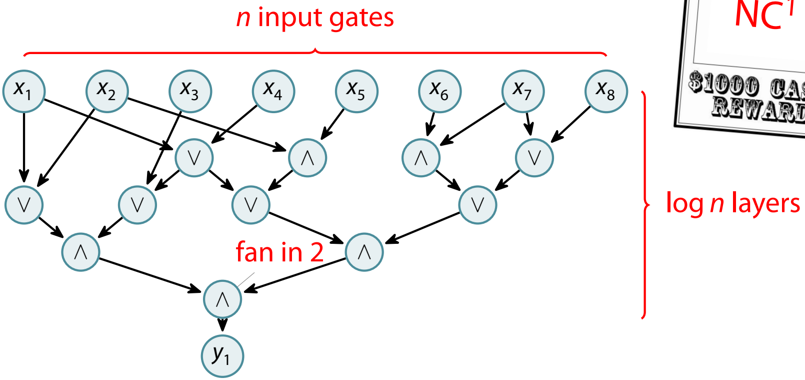
New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

The Class NC^1



- REG $\subseteq$ NC^1 $\subseteq$ L.
- Addition, multiplication, and division all lie in NC^1.
- Tree decompositions cannot be computed in NC^1 unless NC^1 $=$ L.

The NC^1 Version of Courcelle's Theorem.

Theorem (Elberfeld, T, Jakoby, 2012)

Let φ be an MSO formula and k a number. Then there is an NC^1 circuit family that on input of

1. *a graph G*
2. *together with a tree decomposition of G of width k*

decides whether $G \models \varphi$.

(As for logspace, there is also a "counting version".)

Outline

Algorithmic Metatheorems

- Classic Variants …
- …and New Variants

New Variants 1: Space Complexity

- New Theorems
- Applications: Cycle Lengths in Graphs
- Applications: Quantifier Prefix Classes
- Applications: Integer Optimization

New Variants 2: Circuit Complexity

- New Theorems
- Applications: Visible Pushdown Languages

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape	(	[	1	+	1	]	+	(	2	+	[	2	+	2	]	)	)

Stack	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

Stack `(` `(` `[`

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

Stack

	[	[
(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

		[	[	[
Stack	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape	(	[	1	+	1	]	+	(	2	+	[	2	+	2	]	)	)

		[	[	[	[
Stack	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

```
        [  [  [  [
Stack (  (  (  (  (  (
```

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape	(	[	1	+	1	]	+	(	2	+	[	2	+	2	]	)	)

		[	[	[	[		
Stack	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

Stack

	[	[	[	[			(
(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

		[	[	[	[			(	(
Stack	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

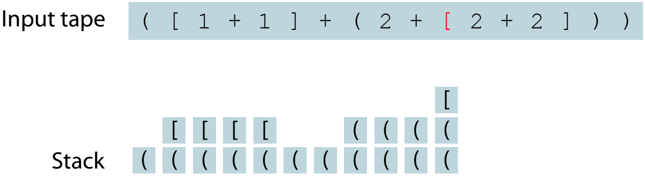
1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

		[	[	[	[			(	(	(
Stack	(	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.



Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

											[	[
		[	[	[	[			(	(	(	(	(
Stack	(	(	(	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

Stack

										[	[	[
	[	[	[	[			(	(	(	(	(	(
(	(	(	(	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

											[	[	[	[
		[	[	[	[			(	(	(	(	(	(	(
Stack	(	(	(	(	(	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

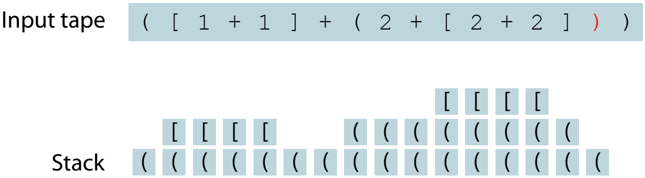
Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

Stack

									[	[	[	[		
	[	[	[	[			(	(	(	(	(	(	(	(
(	(	(	(	(	(	(	(	(	(	(	(	(	(	(

Visible Pushdown Languages

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.



Visible Pushdown Languages

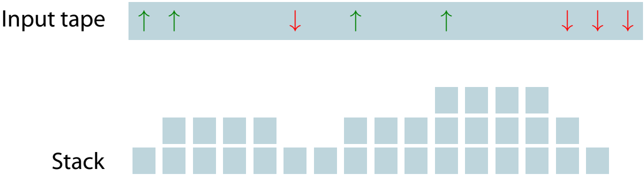
1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.

Input tape `( [ 1 + 1 ] + ( 2 + [ 2 + 2 ] ) )`

											[	[	[	[		
		[	[	[	[			(	(	(	(	(	(	(	(	
Stack	(	(	(	(	(	(	(	(	(	(	(	(	(	(	(	(

We Can Precompute the Stack Outline

1. The input is processed by a *pushdown automaton*.
2. Whether the automaton does a push or a pop may depend only on *the read symbol* and *not* on the state.



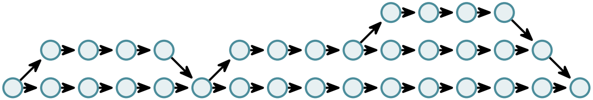
The Complexity of Visible Pushdown Languages

Theorem

All visible pushdown languages lie in NC^1.

Proof.

For an input like ([1 + 1] + (2 + [2 + 2])) we can construct the following graph using "simple counting":



□

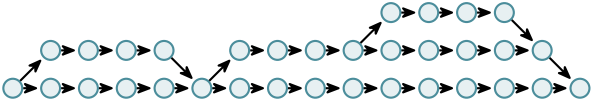
The Complexity of Visible Pushdown Languages

Theorem

All visible pushdown languages lie in NC^1.

Proof.

For an input like $([1 + 1] + (2 + [2 + 2]))$ we can construct the following graph using "simple counting":



Apply the metatheorem for NC^1 to a formula stating *"there exist states and symbols for the nodes that are locally consistent"*. □

Summary

Algorithmic metatheorems have the form

> *If a problem can be described in a certain logic and the input graphs can be decomposed in a certain fashion, then there is a certain kind of algorithm.*

- There are algorithmic metatheorems for logarithmic space.
- There are algorithmic metatheorems for NC^1.
- Algorithmic metatheorems have applications *even when the inputs are not decomposable graphs*.

Outlook

Further Results

- There are algorithmic metatheorems for constant depth circuits (AC^0 and TC^0).
- There are algorithmic metatheorems for pure logic.

Some Open Problems

- Is there an logspace metatheorem for bounded *clique width*?
- Can we *construct* solutions of a certain size?
- How difficult is detecting cycles whose *length modulo* 3 *is* 1?