Unified Approach to Polynomial Algorithms on Graphs of Bounded (bi-)Rank-width

Petr Hliněný,*

Robert Ganian and Jan Obdržálek

Faculty of Informatics, Masaryk University

Botanická 68a, 602 00 Brno, Czech Republic

e-mail: hlineny@fi.muni.cz http://www.fi.muni.cz/~hlineny

ganian@mail.muni.cz obdrzalek@fi.muni.cz

0 Introduction

In this presentation, we will mix some very general (and abstract) ideas about graph *"width" decompositions* and dynamic programming algorithms on those, with specific applications to efficient algorithms for hard problems running on *rank-decompositions* of graphs.

0 Introduction

In this presentation, we will mix some very general (and abstract) ideas about graph *"width" decompositions* and dynamic programming algorithms on those, with specific applications to efficient algorithms for hard problems running on *rank-decompositions* of graphs.

Talk Outline

1 Measuring Graph "Width"

Motivation: Trees are easy to understand and to handle, so how "tree-like" our graphs are . . . , in some well-defined sense?

- A topic occuring both in pure theory (e.g. Graph Minors),
  and in algorithms (Fixed parameter tractability).

1 Measuring Graph "Width"

Motivation: Trees are easy to understand and to handle, so how "tree-like" our graphs are . . . , in some well-defined sense?

- A topic occuring both in pure theory (e.g. Graph Minors),
  and in algorithms (Fixed parameter tractability).
- Many definitions have been studied so far,
  e.g. *tree-width*, *path-width*, *branch-width*, *DAG-width* . . .

1 Measuring Graph "Width"

Motivation: Trees are easy to understand and to handle, so how "tree-like" our graphs are . . . , in some well-defined sense?

- A topic occuring both in pure theory (e.g. Graph Minors),
  and in algorithms (Fixed parameter tractability).

- Many definitions have been studied so far,
  e.g. *tree-width*, *path-width*, *branch-width*, *DAG-width* . . .

- **Clique-width** – another graph complexity measure [Courcelle and Olariu], defined by operations on vertex–labeled graphs:
  - create a new vertex with label i,
  - take the disjoint union of two labeled graphs,
  - add all edges between vertices of label i and label j,
  - and relabel all vertices with label i to have label j.

Rank-Decompositions (a "better view" of clique-width)

- [Oum and Seymour, 03] Bringing the branch-decomposition approach to measure "complexity" of vertex subsets $X \subseteq V(G)$ via *cut-rank*:

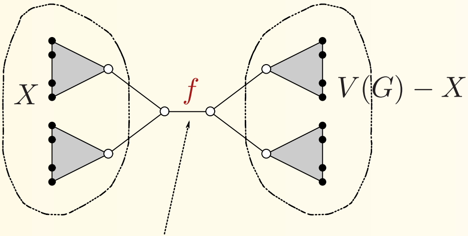
$$\varrho_G(X) = \text{rank of } \begin{array}{cc} & V(G) - X \\ X & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \text{ modulo } 2$$

Rank-Decompositions (a “better view” of clique-width)

- [Oum and Seymour, 03] Bringing the branch-decomposition approach to measure “complexity” of vertex subsets $X \subseteq V(G)$ via *cut-rank*:

$$\varrho_G(\boldsymbol{X}) = \text{rank of} \quad \begin{array}{cc} & V(G)-X \\ X & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \text{ modulo } 2$$

Definition. Decompose $V(G)$ one-to-one into the leaves of a subcubic tree. Then



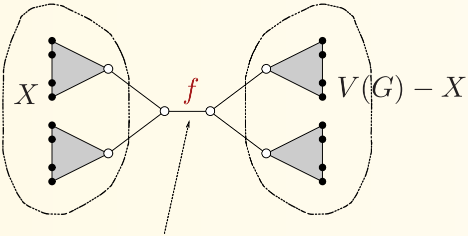
width$(e) = \varrho_G(X)$ where X is displayed by f in the tree.

Rank-Decompositions (a “better view” of clique-width)

- [Oum and Seymour, 03] Bringing the branch-decomposition approach to measure “complexity” of vertex subsets $X \subseteq V(G)$ via *cut-rank*:

$$\varrho_G(X) = \text{rank of} \quad \begin{array}{cc} & V(G)-X \\ X & \begin{pmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \end{pmatrix} \end{array} \text{ modulo } 2$$

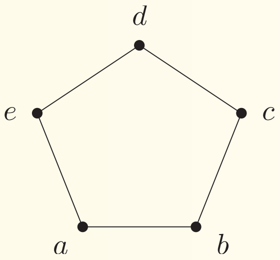
Definition. Decompose $V(G)$ one-to-one into the leaves of a subcubic tree. Then

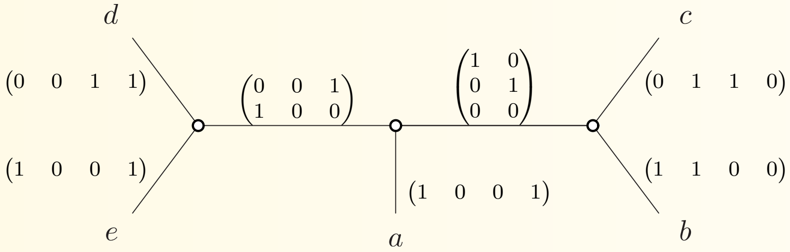


width$(e) = \varrho_G(X)$ where X is displayed by f in the tree.

Rank-width $= \min_{\text{rank-decs. of } G} \max \{\text{width}(f) : f \text{ tree edge}\}$

An example. Cycle C_5 and its *rank-decomposition* of width 2:





Comparing these two

- Rank-width t is related to clique-width k as $\boldsymbol{t \leq k \leq 2^{t+1} - 1}$.
- Both these measures are NP-hard in general.

Comparing these two

- Rank-width t is related to clique-width k as $\boldsymbol{t \leq k \leq 2^{t+1} - 1}$.
- Both these measures are NP-hard in general.
- Clique-width *expressions* seem to be much more "explicit" than *rank-decompositions*, and more suited for design of actual algorithms.

On the other hand, however...

Comparing these two

- Rank-width t is related to clique-width k as $\boldsymbol{t \leq k \leq 2^{t+1} - 1}$.
- Both these measures are NP-hard in general.
- Clique-width *expressions* seem to be much more “explicit” than *rank-decompositions*, and more suited for design of actual algorithms.

On the other hand, however…

- [Corneil and Rotics, 05] Clique-width can really be up to exponentially higher than rank-width.

Comparing these two

- Rank-width t is related to clique-width k as $\boldsymbol{t \leq k \leq 2^{t+1} - 1}$.
- Both these measures are NP-hard in general.
- Clique-width *expressions* seem to be much more "explicit" than *rank-decompositions*, and more suited for design of actual algorithms.

On the other hand, however...

- [Corneil and Rotics, 05] Clique-width can really be up to exponentially higher than rank-width.
- [Oum and PH, 07] There is an *FPT algorithm* for computing an optimal rank-decomposition of a graph in time $O(f(t) \cdot n^3)$.

Comparing these two

- Rank-width t is related to clique-width k as $\boldsymbol{t \leq k \leq 2^{t+1} - 1}$.
- Both these measures are NP-hard in general.
- Clique-width *expressions* seem to be much more "explicit" than *rank-decompositions*, and more suited for design of actual algorithms.

On the other hand, however...

- [Corneil and Rotics, 05] Clique-width can really be up to exponentially higher than rank-width.
- [Oum and PH, 07] There is an *FPT algorithm* for computing an optimal rank-decomposition of a graph in time $O(f(t) \cdot n^3)$.
- And some new results suggest that algorithms designed on rank-decompositions run faster than those designed on clique-width expressions...

2 Dynamic Algorithms and Parse Trees

- A typical idea for a *dynamic algorithm* on a "tree-like" decomposition:
  - Capture all relevant information about the problem on a subtree.
  - Process this information bottom-up in the decomposition.
  - Importantly, this information has limited polynomial size, ideally even constant independent of the input size.

2 Dynamic Algorithms and Parse Trees

- A typical idea for a *dynamic algorithm* on a "tree-like" decomposition:
  - Capture all relevant information about the problem on a subtree.
  - Process this information bottom-up in the decomposition.
  - Importantly, this information has limited polynomial size, ideally even constant independent of the input size.

- How to understand words "all relevant information about the problem"?

  Look for inspiration in traditional finite automata theory!

2 Dynamic Algorithms and Parse Trees

- A typical idea for a *dynamic algorithm* on a "tree-like" decomposition:
  - Capture all relevant information about the problem on a subtree.
  - Process this information bottom-up in the decomposition.
  - Importantly, this information has limited polynomial size, ideally even constant independent of the input size.

- How to understand words "all relevant information about the problem"?

  Look for inspiration in traditional finite automata theory!

Theorem. [Myhill–Nerode, folklore]
Finite automaton states (this is our information) $\leftrightarrow$
right congruence classes on the words (of a regular language).

2 Dynamic Algorithms and Parse Trees

- A typical idea for a *dynamic algorithm* on a "tree-like" decomposition:
  - Capture all relevant information about the problem on a subtree.
  - Process this information bottom-up in the decomposition.
  - Importantly, this information has limited polynomial size, ideally even constant independent of the input size.

- How to understand words "all relevant information about the problem"?

  Look for inspiration in traditional finite automata theory!

  Theorem. [Myhill–Nerode, folklore]
  Finite automaton states (this is our information) $\leftrightarrow$
  right congruence classes on the words (of a regular language).

- Combinatorial extensions of this concept appeared e.g. in the works [Abrahamson and Fellows, 93], [PH, 03], or [Ganian and PH, 08].

The concept of a canonical equivalence

How does the right congruence extend
from formal words with the concatention operation
to, say, graphs with a kind of a “join” operation?

The concept of a canonical equivalence

How does the right congruence extend
from formal words with the concatention operation
to, say, graphs with a kind of a "join" operation?

- Consider the universe of graphs $\mathcal{U}_k$ implicitly associated with
  - some (small) distinguished "*boundary of size k*" of each graph, and
  - a *join operation* $G \oplus H$ acting on the boundaries of disjoint G, H.
- Let $\mathcal{P}$ be a graph property we study.

The concept of a canonical equivalence

How does the right congruence extend
from formal words with the concatention operation
to, say, graphs with a kind of a "join" operation?

- Consider the universe of graphs $\mathcal{U}_k$ implicitly associated with
  - some (small) distinguished "*boundary of size k*" of each graph, and
  - a *join operation* $G \oplus H$ acting on the boundaries of disjoint G, H.
- Let $\mathcal{P}$ be a graph property we study.

Definition. The *canonical equivalence* of $\mathcal{P}$ on $\mathcal{U}_k$ is defined:

$\boldsymbol{G_1 \approx_{\mathcal{P},k} G_2}$ for any $G_1, G_2 \in \mathcal{U}_k$ if and only if, for all $H \in \mathcal{U}_k$,

$$G_1 \oplus H \in \mathcal{P} \iff G_2 \oplus H \in \mathcal{P}\,.$$

The concept of a canonical equivalence

How does the right congruence extend
from formal words with the concatention operation
to, say, graphs with a kind of a "join" operation?

- Consider the universe of graphs $\mathcal{U}_k$ implicitly associated with
  - some (small) distinguished "*boundary of size k*" of each graph, and
  - a *join operation* $G \oplus H$ acting on the boundaries of disjoint G, H.
- Let $\mathcal{P}$ be a graph property we study.

Definition. The *canonical equivalence* of $\mathcal{P}$ on $\mathcal{U}_k$ is defined:

$\boldsymbol{G_1 \approx_{\mathcal{P},k} G_2}$ for any $G_1, G_2 \in \mathcal{U}_k$ if and only if, for all $H \in \mathcal{U}_k$,

$$G_1 \oplus H \in \mathcal{P} \iff G_2 \oplus H \in \mathcal{P} .$$

- Informally, the classes of $\approx_{\mathcal{P},k}$ capture all information about the property $\mathcal{P}$ that can "cross" our graph boundary of size k
(regardless of actual meaning of "boundary" and "join").

Parse trees of decompositions

To give a real usable meaning to the above terms "boundary, join, and universe" we set them in the context of tree-shaped decompositions as follows. . .

Parse trees of decompositions

To give a real usable meaning to the above terms "boundary, join, and universe" we set them in the context of tree-shaped decompositions as follows. . .

- Considering a rooted ???-decomposition of a graph G, we build on the following correspondence:

boundary size k	$\leftrightarrow$	restricted bag-size / width / etc in decomposition
join operator $\oplus$	$\leftrightarrow$	the way pieces of G "stick together" in decomp.

Parse trees of decompositions

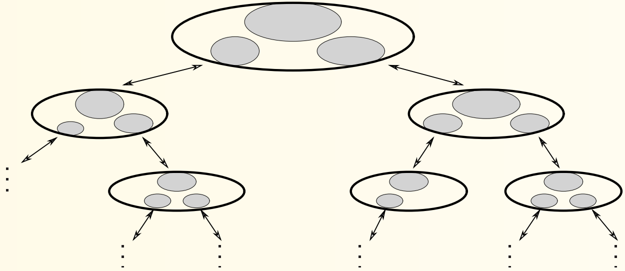
To give a real usable meaning to the above terms "boundary, join, and universe" we set them in the context of tree-shaped decompositions as follows...

- Considering a rooted ???-decomposition of a graph G, we build on the following correspondence:

boundary size k $\leftrightarrow$ restricted bag-size / width / etc in decomposition

join operator $\oplus$ $\leftrightarrow$ the way pieces of G "stick together" in decomp.

- This can be (visually) seen as...



3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

- **Bilinear product** approach of [Courcelle and Kanté, 07]:
  - *boundary* $\sim$ labeling $lab : V(G) \rightarrow 2^{\{1,2,\ldots,t\}}$ (multi-colouring),

3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

- **Bilinear product** approach of [Courcelle and Kanté, 07]:
  - *boundary* $\sim$ labeling $lab : V(G) \to 2^{\{1,2,\dots,t\}}$ (multi-colouring),
  - *join* $\sim$ bilinear form $\mathbf{g}$ over $GF(2)^t$ (i.e. "odd intersection") s.t.

$$\text{edge } uv \;\leftrightarrow\; lab(u) \cdot \mathbf{g} \cdot lab(v) = 1.$$

3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

- **Bilinear product** approach of [Courcelle and Kanté, 07]:
  - *boundary* $\sim$ labeling $lab : V(G) \to 2^{\{1,2,\dots,t\}}$ (multi-colouring),
  - *join* $\sim$ bilinear form $\mathbf{g}$ over $GF(2)^t$ (i.e. "odd intersection") s.t.

$$\text{edge } uv \;\leftrightarrow\; lab(u) \cdot \mathbf{g} \cdot lab(v) = 1.$$

- Join $\to$ a *composition* operator with relabelings f_1, f_2;

$$(G_1, lab^1) \otimes[\mathbf{g} \mid f_1, f_2]\ (G_2, lab^2) \;=\; (H, lab)$$

$\Longrightarrow$ the rank-width **parse tree** [Ganian and PH, 08]:

3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

- **Bilinear product** approach of [Courcelle and Kanté, 07]:
  - *boundary* $\sim$ labeling $lab : V(G) \to 2^{\{1,2,\dots,t\}}$ (multi-colouring),
  - *join* $\sim$ bilinear form $\mathbf{g}$ over $GF(2)^t$ (i.e. "odd intersection") s.t.

    edge $uv \;\leftrightarrow\; lab(u) \cdot \mathbf{g} \cdot lab(v) = 1$.

- Join $\;\to\;$ a *composition* operator with relabelings f_1, f_2;

  $$(G_1, lab^1) \;\otimes[\mathbf{g} \mid f_1, f_2]\; (G_2, lab^2) \;=\; (H, lab)$$

  $\Longrightarrow$ the rank-width **parse tree** [Ganian and PH, 08]:

  t-labeling parse tree for G $\iff$ rank-width of $G \le \boldsymbol{t}$.

3 Parse Trees for Rank-Decompositions

Unlike for branch- or tree-decompositions with obvious parse trees, what is the "boundary" and "join" operation for rank-width?

Our "boundary" includes all vertices, and "join" is just an implicit matrix rank!

- **Bilinear product** approach of [Courcelle and Kanté, 07]:
  - *boundary* $\sim$ labeling $lab : V(G) \to 2^{\{1,2,\dots,t\}}$ (multi-colouring),
  - *join* $\sim$ bilinear form $\mathbf{g}$ over $GF(2)^t$ (i.e. "odd intersection") s.t.

$$\text{edge } uv \;\leftrightarrow\; lab(u) \cdot \mathbf{g} \cdot lab(v) = 1.$$

- Join $\to$ a *composition* operator with relabelings f_1, f_2;

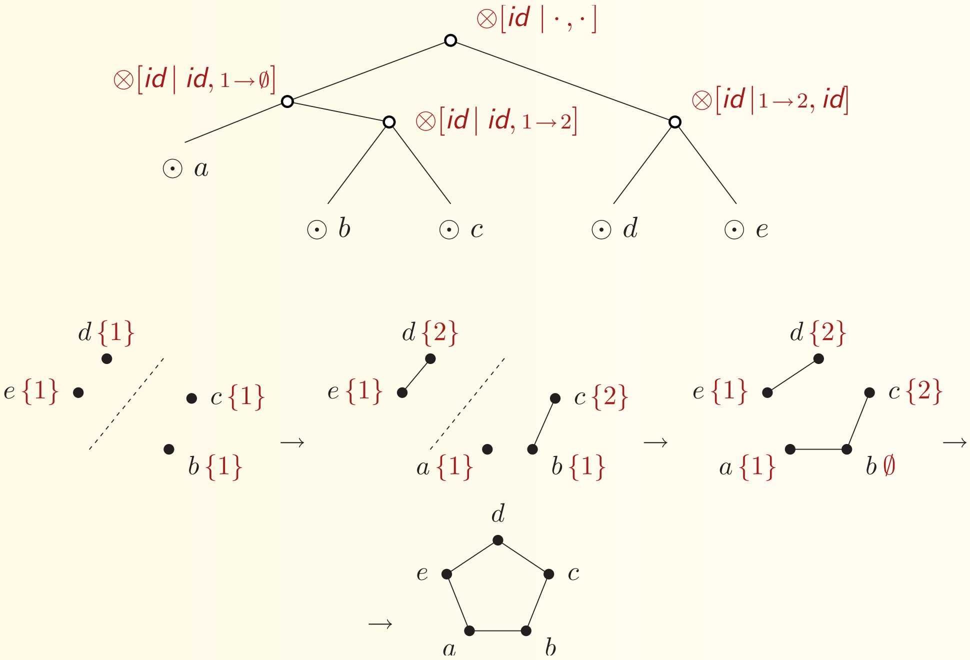
$$(G_1, lab^1) \;\otimes[\mathbf{g} \mid f_1, f_2]\; (G_2, lab^2) \;=\; (H, lab)$$

$\Longrightarrow$ the rank-width **parse tree** [Ganian and PH, 08]:

t-**labeling** parse tree for G $\iff$ rank-width of $G \le \boldsymbol{t}$.

- Independently considered related notion of *R_t-join* decompositions by [Bui-Xuan, Telle, and Vatshelle, 08].

Parse tree. An example generating the cycle C_5 (of rank-width 2):



4 Canonical Equivalence and Algorithms

So, how can one use a canonical equivalence when designing actual algorithms?

4 Canonical Equivalence and Algorithms

So, how can one use a canonical equivalence when designing actual algorithms?

- Let us recall. . .

Theorem. [Myhill–Nerode, folklore]
A finite automaton accepts a given language $\iff$
the number of *right congruence* classes on the words is finite.

4 Canonical Equivalence and Algorithms

So, how can one use a canonical equivalence when designing actual algorithms?

- Let us recall. . .

Theorem. [Myhill–Nerode, folklore]
A finite automaton accepts a given language $\iff$ the number of *right congruence* classes on the words is finite.

- This automaton is constructible and can be emulated in linear time.

4 Canonical Equivalence and Algorithms

So, how can one use a canonical equivalence when designing actual algorithms?

- Let us recall...

Theorem. [Myhill–Nerode, folklore]
A finite automaton accepts a given language $\iff$
the number of *right congruence* classes on the words is finite.

- This automaton is constructible and can be emulated in linear time.

- For parse trees, a straightforward generalization reads:

Theorem. (Analogy of [Myhill–Nerode])
$\mathcal{P}$ is accepted by a finite tree automaton on parse trees of boundary size $\leq k$
$\iff$ the *canonical equivalence* $\approx_{\mathcal{P},k}$ has finitely many classes on $\mathcal{U}_k$.

(Actually, this is a "metatheorem" which requires several more unspoken technical conditions on the parse trees to hold true...)

Extended canonical equivalence

$\boldsymbol{G_1 \approx_{\mathcal{P},k} G_2}$ for any $G_1, G_2 \in \mathcal{U}_k$ if and only if, for all $H \in \mathcal{U}_k$,

$$G_1 \oplus H \models \mathcal{P} \iff G_2 \oplus H \models \mathcal{P}\,.$$

- To apply this concept to predicates $\mathcal{P}(X_1, \dots)$ with free variables, we extend the universe $\mathcal{U}_k$ to *partially-equipped* graphs of boundary $\leq k$.

Extended canonical equivalence

$\boldsymbol{G_1 \approx_{\mathcal{P},k} G_2}$ for any $G_1, G_2 \in \mathcal{U}_k$ if and only if, for all $H \in \mathcal{U}_k$,

$$G_1 \oplus H \models \mathcal{P} \iff G_2 \oplus H \models \mathcal{P} \,.$$

- To apply this concept to predicates $\mathcal{P}(X_1, \dots)$ with free variables, we extend the universe $\mathcal{U}_k$ to *partially-equipped* graphs of boundary $\leq k$.

Theorem. [Ganian and PH, 08]

Suppose ϕ is a formula in the language MS_1. Then the canonical equivalence $\approx_{\phi,t}$ has finite index in the universe of t-labeled partially-equipped graphs.

Extended canonical equivalence

$$\boldsymbol{G_1 \approx_{\mathcal{P},k} G_2} \;\text{ for any } G_1, G_2 \in \mathcal{U}_k \;\text{ if and only if, }\; \text{for all } H \in \mathcal{U}_k,$$
$$G_1 \oplus H \models \mathcal{P} \iff G_2 \oplus H \models \mathcal{P}\,.$$

- To apply this concept to predicates $\mathcal{P}(X_1, \dots)$ with free variables, we extend the universe $\mathcal{U}_k$ to *partially-equipped* graphs of boundary $\leq k$.

Theorem. [Ganian and PH, 08]

Suppose ϕ is a formula in the language MS_1. Then the canonical equivalence $\approx_{\phi,t}$ has finite index in the universe of t-labeled partially-equipped graphs.

- From that one easily concludes an older result:

Theorem. [Courcelle, Makowsky, and Rotics 00]

All *$LinEMSO$ graph optimization* problems (in MS_1 language – only vertices!) on the graphs of bounded rank-width t can be solved in FPT time $\boldsymbol{O(f(t)\cdot n)}$.

Core idea: In dynamic processing of the given parse tree, record optimal representatives of each class of the extended canonical equivalence $\approx_{\phi,t}$ ...

5 Unified Design Style of XP Algorithms

(XP: running in time $O(n^{f(k)})$, not FPT)

Starting point: For many problems $\mathcal{P}$, the number of classes of $\approx_{\mathcal{P},k}$ depends on the input size n ($\rightarrow$ likely no FPT algorithm exists).

5 Unified Design Style of XP Algorithms

(XP: running in time $O(n^{f(k)})$, not FPT)

Starting point: For many problems $\mathcal{P}$, the number of classes of $\approx_{\mathcal{P},k}$ depends on the input size n ($\rightarrow$ likely no FPT algorithm exists).

Yet there are known algorithms for them dynamically processing "information" of polynomial size $O(n^{f(k)})$... How do they work?

We try to give a unified formal description...

5 Unified Design Style of XP Algorithms

(XP: running in time $O(n^{f(k)})$, not FPT)

Starting point: For many problems $\mathcal{P}$, the number of classes of $\approx_{\mathcal{P},k}$ depends on the input size n ($\rightarrow$ likely no FPT algorithm exists).

Yet there are known algorithms for them dynamically processing "information" of polynomial size $O(n^{f(k)})$... How do they work?

We try to give a unified formal description...

In our parse-tree (width k) formalism, we

- assoc. the equiv. classes of $\approx_{\mathcal{P},k}$ with an enum. of suitable "*fragments*",

5 Unified Design Style of XP Algorithms

(XP: running in time $O(n^{f(k)})$, not FPT)

Starting point: For many problems $\mathcal{P}$, the number of classes of $\approx_{\mathcal{P},k}$ depends on the input size n ($\rightarrow$ likely no FPT algorithm exists).

Yet there are known algorithms for them dynamically processing "information" of polynomial size $O(n^{f(k)})$... How do they work?

We try to give a unified formal description...

In our parse-tree (width k) formalism, we

- assoc. the equiv. classes of $\approx_{\mathcal{P},k}$ with an enum. of suitable "*fragments*",
- where the number of distinct "fragments" depends only on k,

5 Unified Design Style of XP Algorithms

(XP: running in time $O(n^{f(k)})$, not FPT)

Starting point: For many problems $\mathcal{P}$, the number of classes of $\approx_{\mathcal{P},k}$ depends on the input size n ($\rightarrow$ likely no FPT algorithm exists).

Yet there are known algorithms for them dynamically processing "information" of polynomial size $O(n^{f(k)})$... How do they work?

We try to give a unified formal description...

In our parse-tree (width k) formalism, we

- assoc. the equiv. classes of $\approx_{\mathcal{P},k}$ with an enum. of suitable "*fragments*",
- where the number of distinct "fragments" depends only on k,
- and we can recombine the fragment enumerations efficiently.

Example 1: Hamiltonian path

XP algorithm wrt. clique-width given by [Espelage, Gurski, and Wanke, 2001].

Theorem. Decide whether a graph G of rank-width t has a *Hamiltonian path* in time

$$O\left(|V(G)|^{\ell(t)}\right) \text{ where } \ell(t) = 4^{t+1} + O(1)$$

Example 1: Hamiltonian path

XP algorithm wrt. clique-width given by [Espelage, Gurski, and Wanke, 2001].

Theorem. Decide whether a graph G of rank-width t has a *Hamiltonian path* in time

$$O\left(|V(G)|^{\ell(t)}\right) \text{ where } \ell(t) = 4^{t+1} + O(1)\,.$$

Proof:

- Considering a *Hamiltonian path* P in the join $G \oplus H$,

Example 1: Hamiltonian path

XP algorithm wrt. clique-width given by [Espelage, Gurski, and Wanke, 2001].

Theorem. Decide whether a graph G of rank-width t has a *Hamiltonian path* in time

$$O\left(|V(G)|^{\ell(t)}\right) \text{ where } \ell(t) = 4^{t+1} + O(1)\,.$$

Proof:

- Considering a *Hamiltonian path* P in the join $G \oplus H$,
- the "fragments" are the subpaths $P_i \subseteq P$ on the G-side

Example 1: Hamiltonian path

XP algorithm wrt. clique-width given by [Espelage, Gurski, and Wanke, 2001].

Theorem. Decide whether a graph G of rank-width t has a *Hamiltonian path* in time

$$O\left(|V(G)|^{\ell(t)}\right) \text{ where } \ell(t) = 4^{t+1} + O(1)\,.$$

Proof:

- Considering a *Hamiltonian path* P in the join $G \oplus H$,
- the "fragments" are the subpaths $P_i \subseteq P$ on the G-side
  - identified by labeling pairs of their ends (only 4^t distinct!),
  - and enumerated at every parse tree node as one multiset.

Example 1: Hamiltonian path

XP algorithm wrt. clique-width given by [Espelage, Gurski, and Wanke, 2001].

Theorem. Decide whether a graph G of rank-width t has a *Hamiltonian path* in time

$$O\left(|V(G)|^{\ell(t)}\right) \text{ where } \ell(t) = 4^{t+1} + O(1)\,.$$

Proof:

- Considering a *Hamiltonian path* P in the join $G \oplus H$,
- the "fragments" are the subpaths $P_i \subseteq P$ on the G-side
  - identified by labeling pairs of their ends (only 4^t distinct!),
  - and enumerated at every parse tree node as one multiset.
- Straightforward dynamic alg. processing then gives the result.

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Fact. For fixed q, this is an *MSOL partitioning problem*.

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Fact. For fixed q, this is an *MSOL partitioning problem*.

Theorem. The *defective (ℓ, q)-colouring problem* with fixed ℓ parts (i.e. minimizing q) can be solved on a graph G of rank-width t in time

$$O\left(|V(G)|^{k(t,\ell)}\right) \text{ where } k(t,\ell) = 4\ell \cdot 2^t + O(1)$$

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Fact. For fixed q, this is an *MSOL partitioning problem*.

Theorem. The *defective (ℓ, q)-colouring problem* with fixed ℓ parts (i.e. minimizing q) can be solved on a graph G of rank-width t in time

$$O\left(|V(G)|^{k(t,\ell)}\right) \text{ where } k(t,\ell) = 4\ell \cdot 2^t + O(1)\,.$$

Proof:

- Consider separately each one colour class X.

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Fact. For fixed q, this is an *MSOL partitioning problem*.

Theorem. The *defective (ℓ, q)-colouring problem* with fixed ℓ parts (i.e. minimizing q) can be solved on a graph G of rank-width t in time

$$O\left(|V(G)|^{k(t,\ell)}\right) \text{ where } k(t,\ell) = 4\ell \cdot 2^t + O(1)\,.$$

Proof:

- Consider separately each one colour class X.
- A "fragment" – one vertex labeling in X,
  but one needs also to record its max. degree in X !

Example 2: Defective colouring

Defective (ℓ, q)-colouring – partition the vertices into ℓ parts such that
– each part induces a subgr. of max. degree $\leq q$.

Considered recently by [Kolman, Lidický, and Sereni, 2009] wrt. tree-width.

Fact. For fixed q, this is an *MSOL partitioning problem*.

Theorem. The *defective (ℓ, q)-colouring problem* with fixed ℓ parts (i.e. minimizing q) can be solved on a graph G of rank-width t in time

$$O\left(|V(G)|^{k(t,\ell)}\right) \text{ where } k(t, \ell) = 4\ell \cdot 2^t + O(1)\,.$$

Proof:

- Consider separately each one colour class X.
- A "fragment" – one vertex labeling in X,
  but one needs also to record its max. degree in X !
- Slightly out of our formalism, and so deserves a closer look...

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?

- Still, one would like to see an explicit (perhaps logic-based) *framework for XP algorithms*, analogously to the MSOL framework with FPT algorithms, cf. [Courcelle] et al.

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?

- Still, one would like to see an explicit (perhaps logic-based) *framework for XP algorithms*, analogously to the MSOL framework with FPT algorithms, cf. [Courcelle] et al.

  – Our presented unified approach shows this should be possible...

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?

- Still, one would like to see an explicit (perhaps logic-based) *framework for XP algorithms*, analogously to the MSOL framework with FPT algorithms, cf. [Courcelle] et al.

  – Our presented unified approach shows this should be possible...

  – And very recently, [Král', Obdržálek, and PH, 2010] have succeded in finding such a framework.

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?
- Still, one would like to see an explicit (perhaps logic-based) *framework for XP algorithms*, analogously to the MSOL framework with FPT algorithms, cf. [Courcelle] et al.
  - Our presented unified approach shows this should be possible...
  - And very recently, [Král', Obdržálek, and PH, 2010] have succeded in finding such a framework.
- **BTW** (totally unrelated...)

  Have you already heard that the *crossing number of almost planar graphs* is NP-complete? [Cabello and Mohar, 2010]

6 Conclusions

- The power of the *Myhill–Nerode–type* formalism extends beyond the finite-state (i.e. related to finite automata) properties. Nice, isn't it?

- Still, one would like to see an explicit (perhaps logic-based) *framework for XP algorithms*, analogously to the MSOL framework with FPT algorithms, cf. [Courcelle] et al.

  - Our presented unified approach shows this should be possible...
  - And very recently, [Král', Obdržálek, and PH, 2010] have succeded in finding such a framework.

- **BTW** (totally unrelated...)

  Have you already heard that the *crossing number of almost planar graphs* is NP-complete? [Cabello and Mohar, 2010]

THANK YOU FOR YOUR ATTENTION