Rank-based Approach on Graphs with Structured Neighborhood

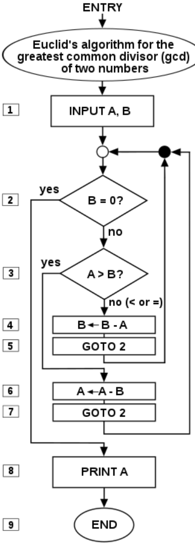
Benjamin Bergougnoux* and Mamadou Moustapha Kanté†

*IRIF, CNRS, Université Paris Diderot

† LIMOS, CNRS, Université Clermont Auvergne

October 11, 2018

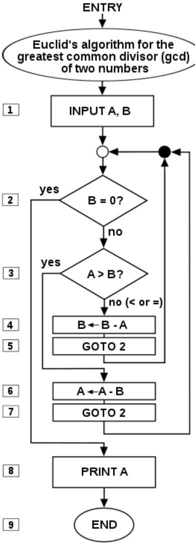
An algorithm



An algorithm:

- take an input
  $\Longrightarrow$ two numbers, a graph, ...

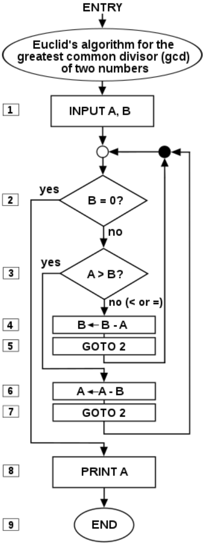
An algorithm



An algorithm:

- take an input
  $\Longrightarrow$ two numbers, a graph, ...
- execute some simple instructions
  $\Longrightarrow$ additions, loops, conditional stuff...

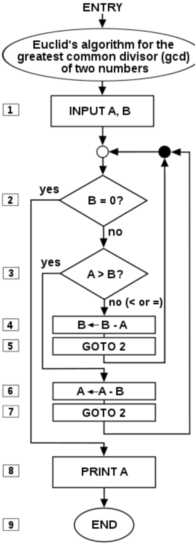
An algorithm



An algorithm:

- take an input
  $\Longrightarrow$ two numbers, a graph, ...
- execute some simple instructions
  $\Longrightarrow$ additions, loops, conditional stuff...
- (may) terminate and return an output
  $\Longrightarrow$ yes, no, a number, ...

An algorithm



An algorithm:

- take an input
  $\Longrightarrow$ two numbers, a graph, ...
- execute some simple instructions
  $\Longrightarrow$ additions, loops, conditional stuff...
- (may) terminate and return an output
  $\Longrightarrow$ yes, no, a number, ...
- solves a problem

Introduction to Complexity

Computers are $\mathrm{EVERYWHERE} \Rightarrow$ we need efficient algorithms !

- Efficient $=$ it uses few resources
  $\Longrightarrow$ time, memory,...

Introduction to Complexity

Computers are EVERYWHERE $\Rightarrow$ we need efficient algorithms !

- Efficient $=$ it uses few resources
  $\Longrightarrow$ time, memory,...
- Time is very important !

Introduction to Complexity

Computers are $\mathrm{EVERYWHERE} \Rightarrow$ we need efficient algorithms !

- Efficient = it uses few resources
  $\Longrightarrow$ time, memory,...
- Time is very important !
- We measure running times in function of input size in the worst case

Introduction to Complexity

Computers are $\mathrm{EVERYWHERE} \Rightarrow$ we need efficient algorithms !

- Efficient = it uses few resources
  $\Longrightarrow$ time, memory,...
- Time is very important !
- We measure running times in function of input size in the worst case
- In theory, constant factor does not matter.
  $\Longrightarrow$ We use big O notation

Running time: example

Seeking an element in a sequence: $O(n)$

Example: looking for 8

11	6	78	1	0	54	7	25	42	42	33	9

Running time: example

If the sequence is sorted: $O(\log(n))$

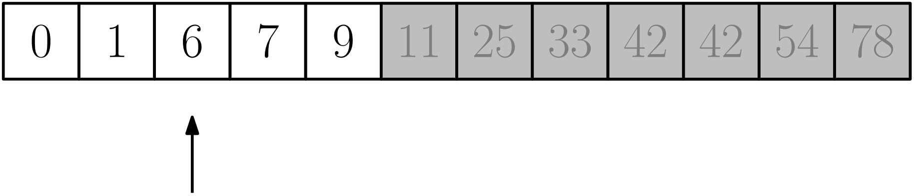
Example: looking for 8

0	1	6	7	9	11	25	33	42	42	54	78

Running time: example

If the sequence is sorted: $O(\log(n))$

Example: looking for 8



Running time: example

If the sequence is sorted: $O(\log(n))$

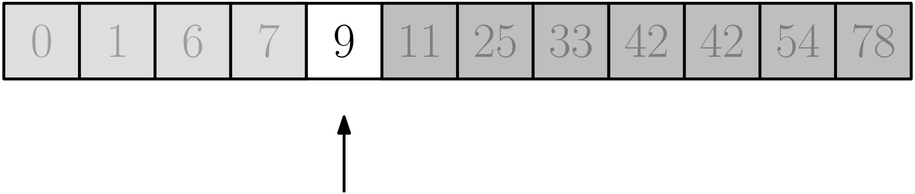
Example: looking for 8

0	1	6	7	9	11	25	33	42	42	54	78

Running time: example

If the sequence is sorted: $O(\log(n))$

Example: looking for 8



Introduction to Complexity

Some problems are harder than others !

- Computing the GCD of two numbers.
  $\Longrightarrow O(n^2)$ polynomial in the input size.

Introduction to Complexity

Some problems are harder than others !

- Computing the GCD of two numbers.
  $\Longrightarrow O(n^2)$ polynomial in the input size.

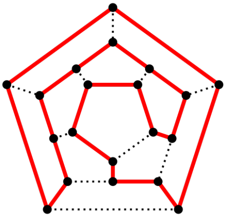
- Finding the divisors of an integer N
  $\Longrightarrow O(\sqrt{N}) = O(2^{n/2})$ exponential in the input size.

Some problems are harder than others !

- Computing the GCD of two numbers.
  $\Longrightarrow O(n^2)$ polynomial in the input size.

- Finding the divisors of an integer N
  $\Longrightarrow O(\sqrt{N}) = O(2^{n/2})$ exponential in the input size.

- Does a graph admit a Hamiltonian Cycle ?
  $\Longrightarrow O(2^n \cdot n^2)$ exponential in the input size.



But...

50 years of intensive research:

- No polynomial time algo. for the Hamiltonian Cycle problem.

But...

50 years of intensive research:

- No polynomial time algo. for the Hamiltonian Cycle problem.
- No intensive research that such an algorithm does not exist.

But...

50 years of intensive research:

- No polynomial time algo. for the Hamiltonian Cycle problem.
- No intensive research that such an algorithm does not exist.

- Solution: comparing problem hardness:

But...

50 years of intensive research:

- No polynomial time algo. for the Hamiltonian Cycle problem.
- No intensive research that such an algorithm does not exist.

- Solution: comparing problem hardness:
  - If we can solve A quickly, then we can solve B quickly too.

But...

50 years of intensive research:

- No polynomial time algo. for the Hamiltonian Cycle problem.
- No intensive research that such an algorithm does not exist.

- Solution: comparing problem hardness:
  - If we can solve A quickly, then we can solve B quickly too.
  - Finding the divisors of an integer harder than computing the GCD.

NP-hardness

NP

Set of all decision (yes-no) problems whose solutions are easily checkable.
$\Longrightarrow$ example: Hamiltonian Cycle problem.

NP-hardness

NP

Set of all decision (yes-no) problems whose solutions are easily checkable.
$\Longrightarrow$ example: Hamiltonian Cycle problem.

NP-hard [Cook / Levin 70s]

A problem is NP-hard if it is at least as hard as every problem in NP:

- If it admits a polynomial time algorithm,
- then every problem in NP admits a polynomial time algorithm !

NP-hardness

NP

Set of all decision (yes-no) problems whose solutions are easily checkable.
$\Longrightarrow$ example: Hamiltonian Cycle problem.

NP-hard [Cook / Levin 70s]

A problem is NP-hard if it is at least as hard as every problem in NP:

- If it admits a polynomial time algorithm,
- then every problem in NP admits a polynomial time algorithm !

Theorem (Karp 1972)

Hamiltonian Cycle *is NP-hard !*

A dead end ?

Theorem (In particular: Garey and Johnson 1979)

Thousands of problems are NP-hard !

No one was able to design a polynomial time algorithm for one of them !

A dead end ?

Theorem (In particular: Garey and Johnson 1979)

Thousands of problems are NP-hard !

No one was able to design a polynomial time algorithm for one of them !

Conjecture

$P \neq NP$: No polynomial algorithms for NP-hard problems.

P vs NP problem: one of the biggest open questions in discrete math.

NP-hard problems everywhere !

We cannot ignore them !

$\Longrightarrow$ They have tons of applications in the real-world in many fields:

- Computer Science,
- Industries, enterprises
  $\Longrightarrow$ Optimization, logistic, planning,...
- Biology
  $\Longrightarrow$ DNA sequencing,...
- Chemistry
- Social choice

In practice:

- There exists efficient softwares for some NP-hard problems.

In practice:

- There exists efficient softwares for some NP-hard problems.
- Wait... what ?

In practice:

- There exists efficient softwares for some NP-hard problems.
- Wait... what ?
- These softwares are not efficient on all instances.
  $\implies$ Do not refute P $\neq$ NP !

In practice:

- There exists efficient softwares for some NP-hard problems.
- Wait... what ?
- These softwares are not efficient on all instances.
  $\Longrightarrow$ Do not refute $\mathsf{P} \neq \mathsf{NP}$!
- NP-hard does not mean hard on every instances

In practice:

- There exists efficient softwares for some NP-hard problems.
- Wait... what ?
- These softwares are not efficient on all instances.
  $\implies$ Do not refute $\mathrm{P} \neq \mathrm{NP}$!
- NP-hard does not mean hard on every instances
- Real-world instances are not arbitrary
  $\implies$ They admit hidden structures

In practice:

- There exists efficient softwares for some NP-hard problems.
- Wait... what ?
- These softwares are not efficient on all instances.
  $\Longrightarrow$ Do not refute $\mathrm{P} \neq \mathrm{NP}$!
- NP-hard does not mean hard on every instances
- Real-world instances are not arbitrary
  $\Longrightarrow$ They admit hidden structures

Question

Can the theory explain these good performances ?
Can we characterize these hidden structures ?

Classical approach

- Consider the instances of a problem $\mathcal{P}$ satisfying some property.

Classical approach

- Consider the instances of a problem $\mathcal{P}$ satisfying some property.
- Design a polynomial time algorithm for these instances.

Classical approach

- Consider the instances of a problem $\mathcal{P}$ satisfying some property.
- Design a polynomial time algorithm for these instances.
- $\mathcal{P}$ is tractable on these instances !

Classical approach

- Consider the instances of a problem $\mathcal{P}$ satisfying some property.
- Design a polynomial time algorithm for these instances.
- $\mathcal{P}$ is tractable on these instances !
- Not very useful in practice

Classical approach

- Consider the instances of a problem $\mathcal{P}$ satisfying some property.
- Design a polynomial time algorithm for these instances.
- $\mathcal{P}$ is tractable on these instances !
- Not very useful in practice
- Real-world instances do not like nice mathematical properties

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:
  - FPT: $O(f(k) \cdot poly(n))$ for some function f.

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:
  - FPT: $O(f(k) \cdot poly(n))$ for some function f.
  - XP: $O(f(k) \cdot n^{g(k)}$ for some functions f and g

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:
  - FPT: $O(f(k) \cdot poly(n))$ for some function f.
  - XP: $O(f(k) \cdot n^{g(k)}$ for some functions f and g
- Distinguish different kind of NP-hard problems

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:
  - FPT: $O(f(k) \cdot poly(n))$ for some function f.
  - XP: $O(f(k) \cdot n^{g(k)}$ for some functions f and g
- Distinguish different kind of NP-hard problems
- Generalize many results of the classical approach

Parameterized Complexity

- Consider some parameter on the instances of $\mathcal{P}$
- Measure the running time in function of this parameter
- Try to design efficient algorithms:
  - FPT: $O(f(k) \cdot poly(n))$ for some function f.
  - XP: $O(f(k) \cdot n^{g(k)}$ for some functions f and g
- Distinguish different kind of NP-hard problems
- Generalize many results of the classical approach
- Good in practice !

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.
- Now a mainstream topic of theoretical computer science.
  $\Longrightarrow$ Thousands of papers !

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.
- Now a mainstream topic of theoretical computer science.
  $\Longrightarrow$ Thousands of papers !
- Expansion in many directions.
  $\Longrightarrow$ Approximation algorithms, Algebra: matroid,...

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.
- Now a mainstream topic of theoretical computer science.
  $\Longrightarrow$ Thousands of papers !
- Expansion in many directions.
  $\Longrightarrow$ Approximation algorithms, Algebra: matroid,...
- Provide tools for designing efficient algorithms.

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.
- Now a mainstream topic of theoretical computer science.
  $\Longrightarrow$ Thousands of papers !
- Expansion in many directions.
  $\Longrightarrow$ Approximation algorithms, Algebra: matroid,...
- Provide tools for designing efficient algorithms.
- But also tools for proving conditional lower bounds.
  $\Longrightarrow$ $W[1]$-hardness: no FPT.

History of Parameterized Complexity

- Introduced in the 90's by Downey and Fellows.
- Now a mainstream topic of theoretical computer science.
  $\Longrightarrow$ Thousands of papers !
- Expansion in many directions.
  $\Longrightarrow$ Approximation algorithms, Algebra: matroid,...
- Provide tools for designing efficient algorithms.
- But also tools for proving conditional lower bounds.
  $\Longrightarrow$ W[1]-hardness: no FPT.
  $\Longrightarrow$ Unless ETH fails, there is no $2^{o(k)} \cdot n^{O(1)}$ algorithm...

Favorite playground: graphs

Graphs offer a wide range of parameters

- maximum degree (the largest number of neighbors of a vertex),

Favorite playground: graphs

Graphs offer a wide range of parameters

- maximum degree (the largest number of neighbors of a vertex),
- the diameter (the largest distance between two vertices),

Favorite playground: graphs

Graphs offer a wide range of parameters

- maximum degree (the largest number of neighbors of a vertex),
- the diameter (the largest distance between two vertices),
- the degeneracy,

Favorite playground: graphs

Graphs offer a wide range of parameters

- maximum degree (the largest number of neighbors of a vertex),
- the diameter (the largest distance between two vertices),
- the degeneracy,
- the genus, etc.

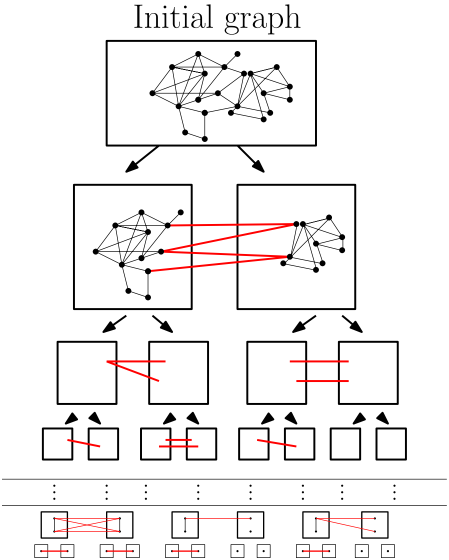
Favorite playground: graphs

Graphs offer a wide range of parameters

- maximum degree (the largest number of neighbors of a vertex),
- the diameter (the largest distance between two vertices),
- the degeneracy,
- the genus, etc.

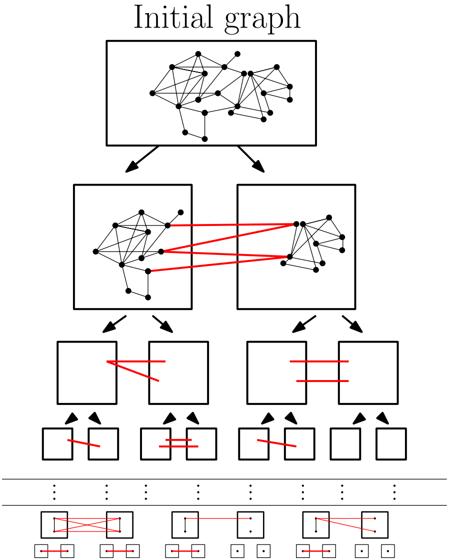
There exists some special kind of parameters: width measures !

Width measures: intuitions



- Divide and Conquer or Dynamic programming:

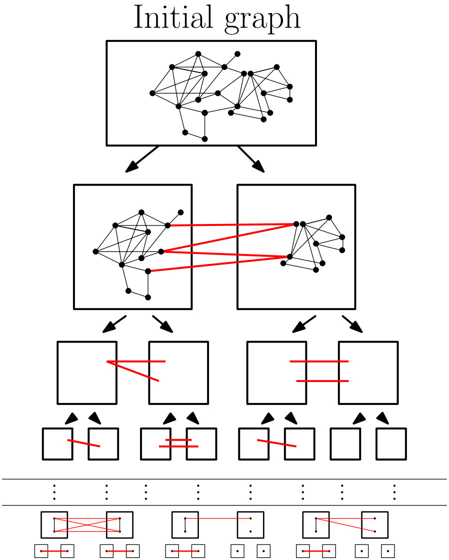
Width measures: intuitions



- Divide and Conquer or Dynamic programming:

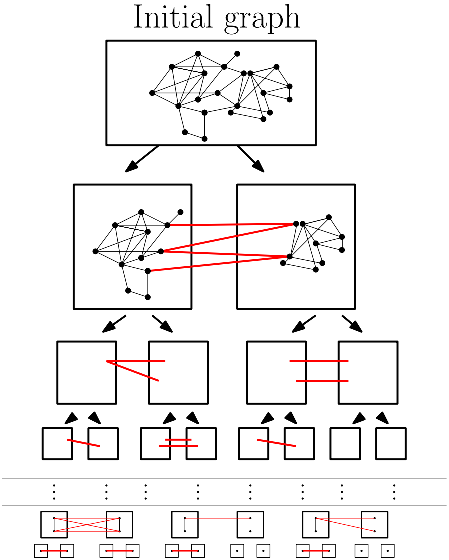
  1. Divide recursively main problem into subproblems

Width measures: intuitions



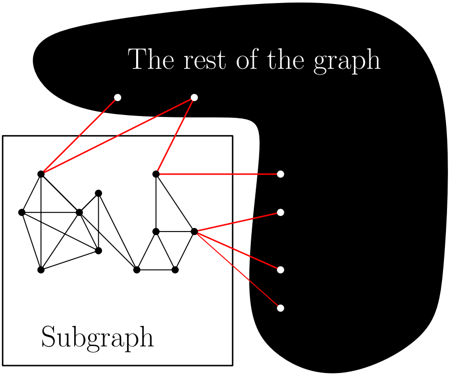
- Divide and Conquer or Dynamic programming:
  1. Divide recursively main problem into subproblems
  2. Stop when subproblems is easy

Width measures: intuitions



- Divide and Conquer or Dynamic programming:
  1. Divide recursively main problem into subproblems
  2. Stop when subproblems is easy
  3. Solves all subproblems recursively (bottom-up)

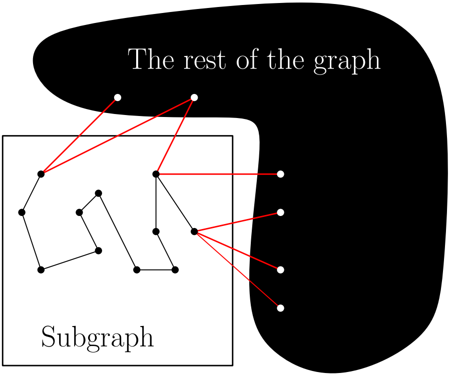
Width measures: intuitions



Example: Hamiltonian Cycle

- How to use the simple boundary:

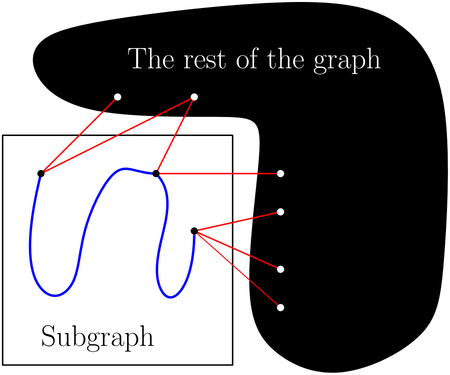
Width measures: intuitions



Example: Hamiltonian Cycle

- How to use the simple boundary:
  1. Observe how partial solutions interact with it.

Width measures: intuitions

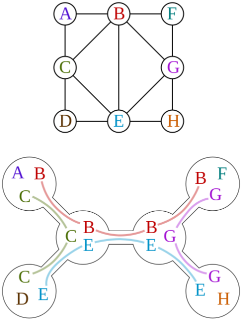


Example: Hamiltonian Cycle

- How to use the simple boundary:
  1. Observe how partial solutions interact with it.
  2. Bound the amount of information we need to store.

Tree-width

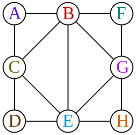
Certainly the most studied and famous graph parameter !

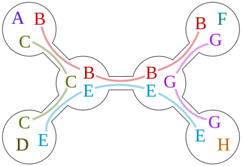


- Measure the tree-likeness.

Tree-width

Certainly the most studied and famous graph parameter !

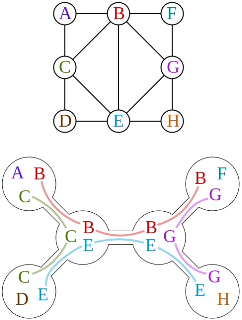




- Measure the tree-likeness.
- Defined from tree-decomposition.

Tree-width

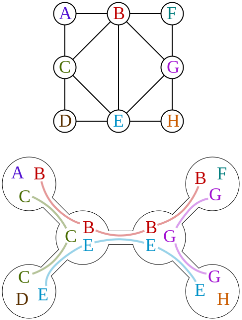
Certainly the most studied and famous graph parameter !



- Measure the tree-likeness.
- Defined from tree-decomposition.
- For many NP-hard problems : $f(\text{tw}) \cdot n$ time algorithm thanks to Courcelle's theorem.

Tree-width

Certainly the most studied and famous graph parameter !



- Measure the tree-likeness.
- Defined from tree-decomposition.
- For many NP-hard problems : $f(\text{tw}) \cdot n$ time algorithm thanks to Courcelle's theorem.
- Computable efficiently: $2^{O(\text{tw})} \cdot n$ constant factor approximation.

Tree-width against NP-hard problems

For problems with a locally checkable property:

- Dominating Set, Independent Set, Vertex Cover, Maximum Induced Matching,...

Tree-width against NP-hard problems

For problems with a locally checkable property:

- Dominating Set, Independent Set, Vertex Cover, Maximum Induced Matching,...
- Naive algorithm: $2^{O(\mathsf{tw})} \cdot n$.

Tree-width against NP-hard problems

For problems with a locally checkable property:

- Dominating Set, Independent Set, Vertex Cover, Maximum Induced Matching,...
- Naive algorithm: $2^{O(\mathsf{tw})} \cdot n$.
- Naive lower bound: $2^{o(\mathsf{tw})} \cdot n^{O(1)}$ unless ETH fails.

Tree-width against NP-hard problems

For problems with a locally checkable property:

- Dominating Set, Independent Set, Vertex Cover, Maximum Induced Matching,...
- Naive algorithm: $2^{O(\mathsf{tw})} \cdot n$.
- Naive lower bound: $2^{o(\mathsf{tw})} \cdot n^{O(1)}$ unless ETH fails.
- Being naive is not that bad for these problems !

Tree-width against NP-hard problems

Problems with a global constraint (connectivity, acyclicity):

- Feedback Vertex Set, Hamiltonian Cycle, Connected Vertex Cover, Connected Dominating Set,...

Tree-width against NP-hard problems

Problems with a global constraint (connectivity, acyclicity):

- Feedback Vertex Set, Hamiltonian Cycle, Connected Vertex Cover, Connected Dominating Set,...
- Naive algorithm: $\mathsf{tw}^{O(\mathsf{tw})} \cdot n$.

Tree-width against NP-hard problems

Problems with a global constraint (connectivity, acyclicity):

- Feedback Vertex Set, Hamiltonian Cycle, Connected Vertex Cover, Connected Dominating Set,...
- Naive algorithm: $\mathsf{tw}^{O(\mathsf{tw})} \cdot n$.
- We need to know the connected components of the partial solutions $\Longrightarrow$ Storing partitions of tw vertices.

Tree-width against NP-hard problems

Problems with a global constraint (connectivity, acyclicity):

- Feedback Vertex Set, Hamiltonian Cycle, Connected Vertex Cover, Connected Dominating Set,...
- Naive algorithm: $\mathsf{tw}^{O(\mathsf{tw})} \cdot n$.
- We need to know the connected components of the partial solutions $\Longrightarrow$ Storing partitions of tw vertices.
- Naive lower bound: $2^{o(\mathsf{tw})} \cdot n^{O(1)}$ unless ETH fails.

Tree-width against NP-hard problems

Problems with a global constraint (connectivity, acyclicity):

- Feedback Vertex Set, Hamiltonian Cycle, Connected Vertex Cover, Connected Dominating Set,...
- Naive algorithm: $\mathsf{tw}^{O(\mathsf{tw})} \cdot n$.
- We need to know the connected components of the partial solutions $\Longrightarrow$ Storing partitions of tw vertices.
- Naive lower bound: $2^{o(\mathsf{tw})} \cdot n^{O(1)}$ unless ETH fails.
- Being naive is not enough.
  $\Longrightarrow$ Can we have $2^{O(\mathsf{tw})} \cdot n^{O(1)}$ algorithm or $\mathsf{tw}^{O(\mathsf{tw})} \cdot n^{O(1)}$ is optimal ?

Tree-width against NP-hard problems

Surprisingly:

Theorem [Cygan et al. FOCS 2011]

Monte Carlo $2^{O(\mathsf{tw}(G))} \cdot n^{O(1)}$ time algorithms for many connectivity problems.

Tree-width against NP-hard problems

Surprisingly:

Theorem [Cygan et al. FOCS 2011]

Monte Carlo $2^{O(\mathsf{tw}(G))} \cdot n^{O(1)}$ time algorithms for many connectivity problems.

Theorem [Bodlaender et al. 2015]

Deterministic $2^{O(\mathsf{tw}(G))} \cdot n$ time algorithms for a wider range of connectivity problems.

Other width measures

Main drawback of tree-width : can be bounded only on sparse graph
$\Longrightarrow$ cliques have tree-width $n-1$.

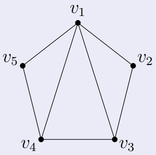
- Dense graphs can be simple too: cliques, distance hereditary graphs,...
- Many NP-hard problems are tractable on these dense graph classes.
- This can be explained through other width measures !
  $\Longrightarrow$ clique-width, rank-width, maximum induced matching width,...
- Most of these width measures are defined through the notion of rooted layout.

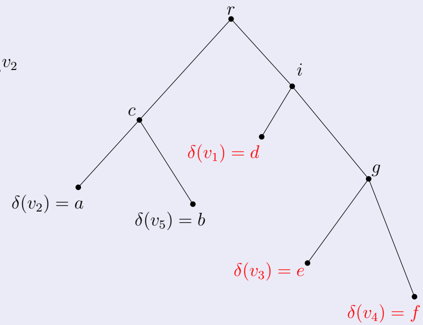
Rooted Layout

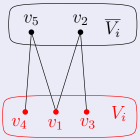
A rooted layout of a graph G is a pair (T, δ) with

- T a rooted tree,
- δ a bijection between the leaves of T and the vertices of G.

Each node x of T is associated with a vertex set V_x of G:





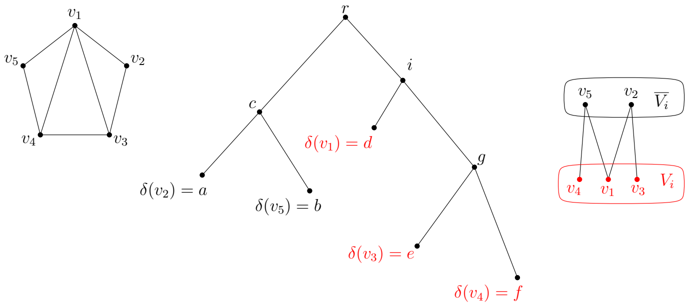


f-width

Given a set function $\mathsf{f} : 2^{V(G)} \to \mathbb{N}$, we define the f-width of

- a rooted layout (T, δ) as $\max_{x \in V(T)} \mathsf{f}(V_x)$,
- $\mathsf{f}(G) := \min \mathsf{f}(\mathcal{L})$ over the rooted layout $\mathcal{L}$ of G.

- f is intend to measure the simplicity of a boundary



Rank-width

The rank-width of G is the rw-width of G where rw(A) is the rank of the adjacency matrix between A and $\overline{A}$.

Mim-width

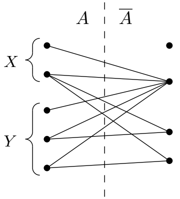
The mim-width of G is the mim-width of G where mim(A) is the size of a maximum induced matching of the bipartite graph between A and $\overline{A}$.

Let G be a graph and $A \subseteq V(G)$.

d-neighbor equivalence

For $d \in \mathbb{N}$, $X, Y \subseteq A$ are d-neighbor equivalent over A if for all $v \in \overline{A}$:

- $\min(d, |N(v) \cap X|) = \min(d, |N(v) \cap Y|)$.
- (If $d = 1$, then $N(X) \cap \overline{A} = N(Y) \cap \overline{A}$.



Number of neighbor in

X	Y
0	0
2	3
2	2
2	2

X and Y are 2-neighbor equivalent but not 3-neighbor equivalent.

d-neighbor width

For every $A \subseteq V(G)$, s-nec$_d(A)$ is the maximum between the number of equivalence classes of

- the d-neighbor equivalence relation over A,
- the d-neighbor equivalence relation over $\overline{A}$.

d-neighbor width

For every $A \subseteq V(G)$, $\text{s-nec}_d(A)$ is the maximum between the number of equivalence classes of

- the d-neighbor equivalence relation over A,
- the d-neighbor equivalence relation over $\overline{A}$.

Given a rooted layout $\mathcal{L}$, we have the following upper bounds on $\text{s-nec}(\mathcal{L})$

Clique-width	Rank-width	Mim-width
$(d+1)^{\mathsf{cw}(\mathcal{L})}$	$2^{d \cdot \mathsf{rw}(\mathcal{L})^2}$	$n^{d \cdot \mathsf{mim}(\mathcal{L})}$

A useful tool for problem with locally checkable property

Thanks to d-neighbor width, we obtained:

- the best (up to a constant in the exponent) algorithm

A useful tool for problem with locally checkable property

Thanks to d-neighbor width, we obtained:

- the best (up to a constant in the exponent) algorithm
- for problems with a locally checkable property

A useful tool for problem with locally checkable property

Thanks to d-neighbor width, we obtained:

- the best (up to a constant in the exponent) algorithm
- for problems with a locally checkable property
- parameterized by: tree-width, clique-width, rank-width, mim-width,...

A useful tool for problem with locally checkable property

Thanks to d-neighbor width, we obtained:

- the best (up to a constant in the exponent) algorithm
- for problems with a locally checkable property
- parameterized by: tree-width, clique-width, rank-width, mim-width,...

A useful tool for problem with locally checkable property

Thanks to d-neighbor width, we obtained:

- the best (up to a constant in the exponent) algorithm
- for problems with a locally checkable property
- parameterized by: tree-width, clique-width, rank-width, mim-width,...

We can design $\text{s-nec}_d(\mathcal{L}) \cdot n^c$ time algorithm for some constants d and c.

What we do

Theorem [Bodlaender et al. 2015]

Deterministic $2^{O(\mathsf{tw}(G))} \cdot n$ time algorithms for a wider range of connectivity problems.

Bodlaender et al. introduced a technique call rank-based approach.

B. and Kanté 2018

We can use the rank-based approach with d-neighbor-width to design:

- $\mathsf{s\text{-}nec}_d(\mathcal{L}) \cdot n^c$ time algorithm
- for the connected variant of problem with locally checkable property $\Longrightarrow$ Connected Dominating Set, Connected Vertex Cover,...

Consequences

Corollary

We obtained algorithms with the following running times

Clique-width	Rank-width	Mim-width
$(d+1)^{\mathsf{cw}(\mathcal{L})} \cdot n$	$2^{d \cdot \mathsf{rw}(\mathcal{L})^2} \cdot n^{O(1)}$	$n^{d \cdot \mathsf{mim}(\mathcal{L})}$

These running times match (up to a constant in the exponent) the best known running times for Vertex Cover and Dominating Set.

Extension to Acyclicity

B. and Kanté

We obtained algorithms with the following running times

Clique-width	Rank-width	Mim-width
$(d+1)^{\mathsf{cw}(\mathcal{L})} \cdot n$	$2^{d \cdot \mathsf{rw}(\mathcal{L})^2} \cdot n^{O(1)}$	$n^{d \cdot \mathsf{mim}(\mathcal{L})}$

for the acyclic variant of problems with locally checkable property
$\Longrightarrow$ Maximum Induced Tree, Feedback Vertex Set, Longest Induced Path,...

- We did not obtained $\mathsf{s\text{-}nec}_d(\mathcal{L}) \cdot n^c$ time algorithm...
- But we heavily rely on the d-neighbor equivalence relation !

Conclusion

- d-neighbor equivalence is incredibly useful.
- Not only for local problems.
- Works also for tree-width.
  $\Longrightarrow$ Maximum matching width !