Parameterized Complexity

Josef Eisl

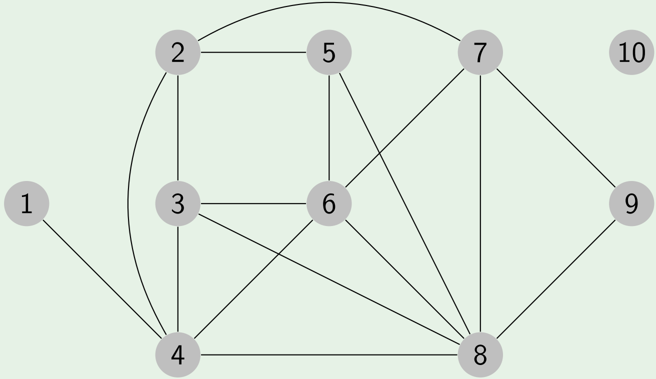
Seminar in Algorithms
186.182

November 20, 2012

Table of contents

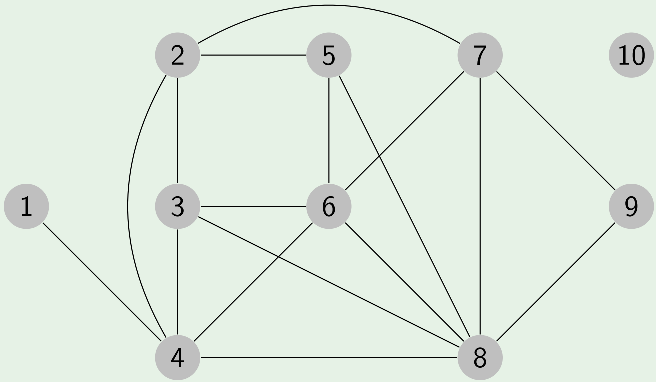
Motivation

Example



Motivation

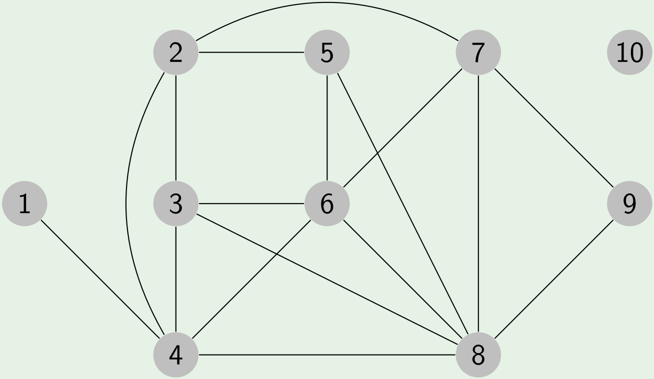
Example



- Is there a path between vertex 1 and vertex 9 of size at most 5?

Motivation

Example



- Is there a path between vertex 1 and vertex 9 of size at most 5?
- Is there a vertex cover of size at most 5?

Complexity Theory

- Analyse how *hard* it is to solve a problem

Complexity Theory

- Analyse how *hard* it is to solve a problem
    - w.r.t. computation time (and space)

Complexity Theory

- Analyse how *hard* it is to solve a problem
    - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems

Complexity Theory

- Analyse how *hard* it is to solve a problem
    - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
    - *Tractable:* can be solved in polynomial time

Complexity Theory

- Analyse how *hard* it is to solve a problem
  - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
  - *Tractable:* can be solved in polynomial time
    - The complexity class P (polynomial-time)

Complexity Theory

- Analyse how *hard* it is to solve a problem
    - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
    - *Tractable:* can be solved in polynomial time
        - The complexity class P (polynomial-time)
        - Runtime bounds: $n^{O(1)}$

Complexity Theory

- Analyse how *hard* it is to solve a problem
  - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
  - *Tractable:* can be solved in polynomial time
    - The complexity class P (polynomial-time)
    - Runtime bounds: $n^{O(1)}$
    - **Example:** REACHABILITY, SORTING, ...

Complexity Theory

- Analyse how *hard* it is to solve a problem
  - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
  - *Tractable:* can be solved in polynomial time
    - The complexity class P (polynomial-time)
    - Runtime bounds: $n^{O(1)}$
    - **Example:** Reachability, Sorting, . . .
  - *Intractable:* no hope for polynomial time algorithm

Complexity Theory

- Analyse how *hard* it is to solve a problem
    - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
    - *Tractable:* can be solved in polynomial time
        - The complexity class P (polynomial-time)
        - Runtime bounds: $n^{O(1)}$
        - **Example:** REACHABILITY, SORTING, ...
    - *Intractable:* no hope for polynomial time algorithm
        - The complexity class NP (nondet.-polynomial time) or higher

Complexity Theory

- Analyse how *hard* it is to solve a problem
  - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
  - *Tractable:* can be solved in polynomial time
    - The complexity class P (polynomial-time)
    - Runtime bounds: $n^{O(1)}$
    - **Example:** REACHABILITY, SORTING, ...
  - *Intractable:* no hope for polynomial time algorithm
    - The complexity class NP (nondet.-polynomial time) or higher
    - Runtime bounds: $2^{n^{O(1)}}$

Complexity Theory

- Analyse how *hard* it is to solve a problem
  - w.r.t. computation time (and space)
- Distinguish easy (*tractable*) and difficult (*intractable*) problems
  - *Tractable:* can be solved in polynomial time
    - The complexity class P (polynomial-time)
    - Runtime bounds: $n^{O(1)}$
    - **Example:** REACHABILITY, SORTING, . . .
  - *Intractable:* no hope for polynomial time algorithm
    - The complexity class NP (nondet.-polynomial time) or higher
    - Runtime bounds: $2^{n^{O(1)}}$
    - **Example:** VERTEX COVER, SAT, DOMINATING SET, . . .

Definition of Problems

Definition (Problem)

A *problem* is a *task/question* together with an *infinite set of instances*.

Definition of Problems

Definition (Problem)

A *problem* is a *task/question* together with an *infinite set of instances*.

Problem (SAT)

Instance: *A Boolean formula* φ.
Question: *Is* φ *satisfiable?*

Definition of Problems

Definition (Problem)

A *problem* is a *task/question* together with an *infinite set of instances*.

Problem (Sat)

Instance: *A Boolean formula φ.*
Question: *Is φ satisfiable?*

Example (Instance of Sat)

$$(A \vee \neg B \vee C) \wedge (D \vee B) \wedge (D \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg A \vee B)$$

Definition of Problems

Definition (Problem)

A *problem* is a *task/question* together with an *infinite set of instances*.

Problem (SAT)

Instance: *A Boolean formula* φ.
Question: *Is* φ *satisfiable?*

Example (Instance of SAT)

$$(A \vee \neg B \vee C) \wedge (D \vee B) \wedge (D \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg A \vee B)$$

- What kind of problems are we interested in?

Definition of Problems

Definition (Problem)

A *problem* is a *task/question* together with an *infinite set of instances*.

Problem (SAT)

Instance: *A Boolean formula φ.*
Question: *Is φ satisfiable?*

Example (Instance of SAT)

$$(A \lor \neg B \lor C) \land (D \lor B) \land (D \lor \neg C) \land (\neg A \lor C) \land (\neg A \lor B)$$

- What kind of problems are we interested in?
    - Decision problems (yes/no answer)

Intractability: The Class NP and beyond

- No hope for polynomial time algorithm

Intractability: The Class NP and beyond

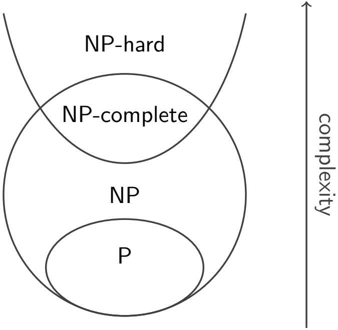
- No hope for polynomial time algorithm
  - NP-hard: at least in the class NP

Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
    - NP-hard: at least in the class NP
    - NP-complete: known to be in the class NP

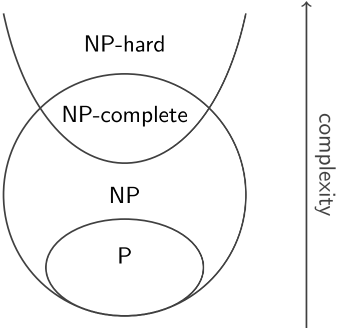
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
  - NP-hard: at least in the class NP
  - NP-complete: known to be in the class NP



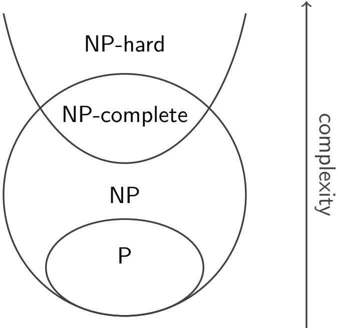
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
    - NP-hard: at least in the class NP
    - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$



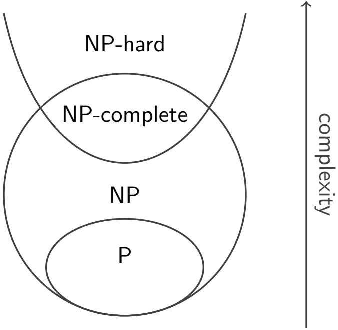
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
    - NP-hard: at least in the class NP
    - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$
- So NP-hard problem can't be solved?



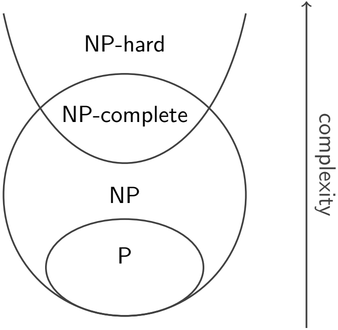
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
  - NP-hard: at least in the class NP
  - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$
- So NP-hard problem can't be solved?
  - Clever algorithms solve many *instances* efficiently:



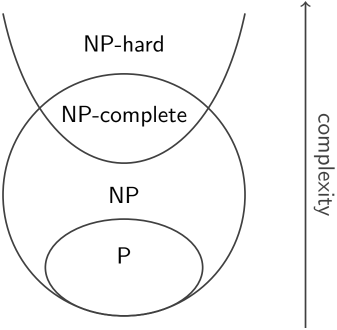
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
  - NP-hard: at least in the class NP
  - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$
- So NP-hard problem can't be solved?
  - Clever algorithms solve many *instances* efficiently:
    - ILP, SAT solver, ...



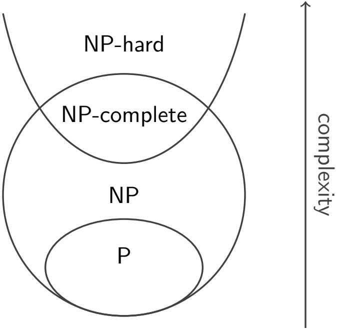
Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
    - NP-hard: at least in the class NP
    - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$
- So NP-hard problem can't be solved?
    - Clever algorithms solve many *instances* efficiently:
        - ILP, SAT solver, ...
    - **But** there is always a *bad* instance



Intractability: The Class NP and beyond

- No hope for polynomial time algorithm
  - NP-hard: at least in the class NP
  - NP-complete: known to be in the class NP
- Combinatorial explosion! $2^{n^{O(1)}}$
- So NP-hard problem can't be solved?
  - Clever algorithms solve many *instances* efficiently:
    - ILP, SAT solver, . . .
  - **But** there is always a *bad* instance



Conjecture

$$P \neq NP$$

The Source of the Hardness

- What are the real world instances?

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *“measure”* this properties?

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view
- Parameter: anything that classifies the problem instances, e.g.:

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view
- Parameter: anything that classifies the problem instances, e.g.:
  - Size of the solution set

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view
- Parameter: anything that classifies the problem instances, e.g.:
  - Size of the solution set
  - *Treewidth* of a graph

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view
- Parameter: anything that classifies the problem instances, e.g.:
  - Size of the solution set
  - *Treewidth* of a graph
  - Max. number of literals in the clauses of a CNF-formula

The Source of the Hardness

- What are the real world instances?
- Why do the worst case exponential algorithms work in practice?
- What properties does separate a *good* instance from a *bad*?
- Can we somehow *"measure"* this properties?

Definition (Parameterized Problem)

A *parameterized problem* is a *task/question* together with an *infinite set of instances* and a *parameter*, often denoted by k.

- Look at the problem from a two-dimensional point of view
- Parameter: anything that classifies the problem instances, e.g.:
    - Size of the solution set
    - *Treewidth* of a graph
    - Max. number of literals in the clauses of a CNF-formula
- Some parameters are useful, most are not!

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter
- In other words: if k is fixed, we can solve the problem in polynomial time

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter
- In other words: if k is fixed, we can solve the problem in polynomial time
    - The problem gets *tractable*

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter
- In other words: if k is fixed, we can solve the problem in polynomial time
    - The problem gets *tractable*
- **Remark 1:** FPT results are always with respect to a parameter!

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter
- In other words: if k is fixed, we can solve the problem in polynomial time
    - The problem gets *tractable*
- **Remark 1:** FPT results are always with respect to a parameter!
- **Remark 2:** There is no bound on $f(k)$ → might be huge!

The Class FPT

Definition

A parameterized problem is *fixed-parameter tractable* (FPT) w.r.t. parameter k if it can be computed in $f(k) \cdot n^{O(1)}$ time where $f(k)$ is only depending on k.

- Shift the combinatorial explosion into the parameter
- In other words: if k is fixed, we can solve the problem in polynomial time
    - The problem gets *tractable*
- **Remark 1:** FPT results are always with respect to a parameter!
- **Remark 2:** There is no bound on $f(k)$ → might be huge!

Parameterized complexity is based on a deal with the devil of intractability. *(*R.G. Downey and M.R. Fellows*)*

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher
    - Hardness proofs by reduction

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher
    - Hardness proofs by reduction
- Parameterized Complexity hierarchy W[t]

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher
    - Hardness proofs by reduction
- Parameterized Complexity hierarchy W[t]
    - Similar to the polynomial hierarchy

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher
    - Hardness proofs by reduction
- Parameterized Complexity hierarchy W[t]
    - Similar to the polynomial hierarchy
- **Example:** $\text{CNF-}\textsc{Sat}$ with parameter $k =$ maximal clause size

The Class W[1] and beyond

- If a problem stays *intractable* w.r.t. a parameter?
    - In the class W[1] or higher
    - Hardness proofs by reduction
- Parameterized Complexity hierarchy W[t]
    - Similar to the polynomial hierarchy
- **Example:** CNF-SAT with parameter $k =$ maximal clause size
    - *intractable* for $k \geq 3$

Table of Contents

The concept of FPT *belongs into the toolkit of every algorithm designer.*
*(*Rolf Niedermeier, Friedrich-Schiller-Universität Jena*)*

Table of Contents

Data Reduction

- Polynomial-time *pre-procession*

Data Reduction

- Polynomial-time *pre-procession*
  - Cut away the *easy parts*

Data Reduction

- Polynomial-time *pre-procession*
    - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve

Data Reduction

- Polynomial-time *pre-procession*
  - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
  - **Note:** the same hardness as the original problem!

Data Reduction

- Polynomial-time *pre-procession*
    - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
    - **Note:** the same hardness as the original problem!
    - Otherwise P = NP

Data Reduction

- Polynomial-time *pre-procession*
    - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
    - **Note:** the same hardness as the original problem!
    - Otherwise P = NP
- Not only important for fixed-parameter algorithms!

Data Reduction

- Polynomial-time *pre-procession*
    - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
    - **Note:** the same hardness as the original problem!
    - Otherwise P = NP
- Not only important for fixed-parameter algorithms!
    - Also other approaches: approximation, heuristics, . . .

Data Reduction

- Polynomial-time *pre-procession*
    - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
    - **Note:** the same hardness as the original problem!
    - Otherwise P = NP
- Not only important for fixed-parameter algorithms!
    - Also other approaches: approximation, heuristics, ...
    - If there are (practical) data reductions then use them!

Data Reduction

- Polynomial-time *pre-procession*
  - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
  - **Note:** the same hardness as the original problem!
  - Otherwise P = NP
- Not only important for fixed-parameter algorithms!
  - Also other approaches: approximation, heuristics, . . .
  - If there are (practical) data reductions then use them!
- Two kinds of rules:

Data Reduction

- Polynomial-time *pre-procession*
  - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
  - **Note:** the same hardness as the original problem!
  - Otherwise P = NP
- Not only important for fixed-parameter algorithms!
  - Also other approaches: approximation, heuristics, . . .
  - If there are (practical) data reductions then use them!
- Two kinds of rules:
  - *parameter-independent*: do not need to know the parameter

Data Reduction

- Polynomial-time *pre-procession*
  - Cut away the *easy parts*
- What remains is a *core* that is difficult to solve
  - **Note:** the same hardness as the original problem!
  - Otherwise P = NP
- Not only important for fixed-parameter algorithms!
  - Also other approaches: approximation, heuristics, . . .
  - If there are (practical) data reductions then use them!
- Two kinds of rules:
  - *parameter-independent*: do not need to know the parameter
  - *parameter-dependent*: need explicit knowledge about the parameter

Weihe's train problem

Problem (WEIHE'S TRAIN PROBLEM)

Instance: *A bipartite graph* $G = (S, T, E)$ *with stations* S *and trains* T *and a positive integer* k.
Question: *Is there a* $S' \subseteq S$ *of size* k *so that every train stops at a station in* S'.

Weihe's train problem

Problem (WEIHE'S TRAIN PROBLEM)

Instance: *A bipartite graph $G = (S, T, E)$ with stations S and trains T and a positive integer k.*
Question: *Is there a $S' \subseteq S$ of size k so that every train stops at a station in S'.*

- Special case of HITTING SET → NP-complete

Weihe's train problem

Problem (WEIHE'S TRAIN PROBLEM)

Instance: *A bipartite graph* $G = (S, T, E)$ *with stations* S *and trains* T *and a positive integer* k.
Question: *Is there a* $S' \subseteq S$ *of size* k *so that every train stops at a station in* S'.

- Special case of HITTING SET → NP-complete

Definition (Weihe's reduction rules)

For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example: Weihe's train problem

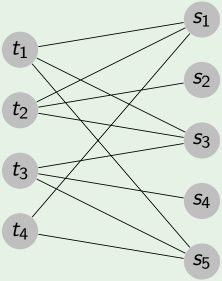
Definition (Weihe's reduction rules)

For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example



Example: Weihe's train problem

Definition (Weihe's reduction rules)

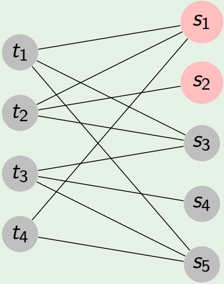
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$



Example: Weihe's train problem

Definition (Weihe's reduction rules)

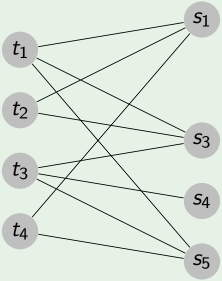
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$
  - delete s_2



Example: Weihe's train problem

Definition (Weihe's reduction rules)

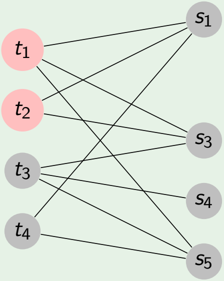
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$
  - delete s_2
- Train Rule:
  $N(t_2) = \{s_1, s_3\} \subseteq N(t_1) = \{s_1, s_3, s_5\}$



Example: Weihe's train problem

Definition (Weihe's reduction rules)

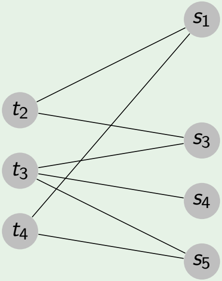
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$
  - delete s_2
- Train Rule:
  $N(t_2) = \{s_1, s_3\} \subseteq N(t_1) = \{s_1, s_3, s_5\}$
  - delete t_1



Example: Weihe's train problem

Definition (Weihe's reduction rules)

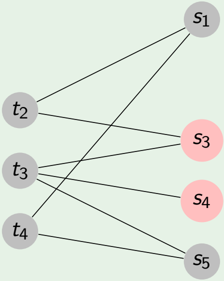
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$
  - delete s_2
- Train Rule:
  $N(t_2) = \{s_1, s_3\} \subseteq N(t_1) = \{s_1, s_3, s_5\}$
  - delete t_1
- Station Rule: $N(s_4) = \{t_3\} \subseteq N(s_3) = \{t_2, t_3\}$



Example: Weihe's train problem

Definition (Weihe's reduction rules)

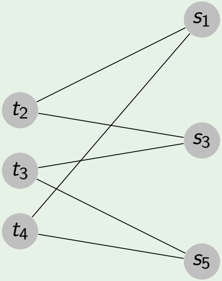
For $s, s' \in S$ and $t, t' \in T$. $N(v)$ denotes the of neighbours of v.

Station Rule $N(s) \subseteq N(s')$ then delete s.

Train Rule $N(t) \subseteq N(t')$ then delete t'.

Example

- Station Rule:
  $N(s_2) = \{t_2\} \subseteq N(s_1) = \{t_1, t_2, t_4\}$
  - delete s_2
- Train Rule:
  $N(t_2) = \{s_1, s_3\} \subseteq N(t_1) = \{s_1, s_3, s_5\}$
  - delete t_1
- Station Rule: $N(s_4) = \{t_3\} \subseteq N(s_3) = \{t_2, t_3\}$
  - delete s_4



Properties: Weihe's train problem

- Works very well in practice:

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)
    - About 10 000 vertices reduced to sub-problems of size ≤ 50

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)
    - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)
    - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules
- Does not find *all* possible solutions

Properties: Weihe's train problem

- Works very well in practice:
  - Evaluation on real data (European train systems)
  - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules
- Does not find *all* possible solutions
- **Drawback:** we can not prove the effectiveness of this reduction!

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)
    - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules
- Does not find *all* possible solutions
- **Drawback:** we can not prove the effectiveness of this reduction!
    - No guarantee that it works on all instances

Properties: Weihe's train problem

- Works very well in practice:
  - Evaluation on real data (European train systems)
  - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules
- Does not find *all* possible solutions
- **Drawback:** we can not prove the effectiveness of this reduction!
  - No guarantee that it works on all instances
- Can we prove the quality of other reductions?

Properties: Weihe's train problem

- Works very well in practice:
    - Evaluation on real data (European train systems)
    - About 10 000 vertices reduced to sub-problems of size ≤ 50
- Only *parameter-independent* rules
- Does not find *all* possible solutions
- **Drawback:** we can not prove the effectiveness of this reduction!
    - No guarantee that it works on all instances
- Can we prove the quality of other reductions?
    - *Yes* → Problem Kernels (next slide)

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,
- (I, k) is a positive instance *iff* (I', k') is one,

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,
- (I, k) is a positive instance *iff* (I', k') is one,
- the transformation from (I, k) to (I', k') must be computable in polynomial time.

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,
- (I, k) is a positive instance *iff* (I', k') is one,
- the transformation from (I, k) to (I', k') must be computable in polynomial time.

- The upper bound of the kernel is independent of the input size!

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,
- (I, k) is a positive instance *iff* (I', k') is one,
- the transformation from (I, k) to (I', k') must be computable in polynomial time.

- The upper bound of the kernel is independent of the input size!
- The solution to (I', k') must not yield a solution to (I, k)

Problem Kernels

Definition (Problem Kernel)

Reduction to a problem kernel means to replace the instance (I, k) by a reduced instance (I', k') such that

- $k' \leq k$ and $|I'| \leq g(k)$ where I is the problem instance, k is the parameter and $g(k)$ is a function solely depending on k,
- (I, k) is a positive instance *iff* (I', k') is one,
- the transformation from (I, k) to (I', k') must be computable in polynomial time.

- The upper bound of the kernel is independent of the input size!
- The solution to (I', k') must not yield a solution to (I, k)
  - But most time it does!

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph* $G = (V, E)$ *and a nonnegative integer* k*.*
Question: *Is there a subset of vertices* $C \subseteq V$ *with* k *or fewer vertices such that each edge in* E *has at least one of its endpoints in* C*.*

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- VERTEX COVER is the most intensively studied problem in the FPT community

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- VERTEX COVER is the most intensively studied problem in the FPT community

Buss's reduction:

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- VERTEX COVER is the most intensively studied problem in the FPT community

Buss's reduction:

VC1 Remove all isolated vertices.

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- VERTEX COVER is the most intensively studied problem in the FPT community

Buss's reduction:

VC1 Remove all isolated vertices.

VC2 For every degree-1 vertex, put the neighbour into the cover and delete both vertices from V.

VERTEX COVER: Buss's reduction to a Problem Kernel

Problem (VERTEX COVER)

Instance: *A graph* $G = (V, E)$ *and a nonnegative integer* k.
Question: *Is there a subset of vertices* $C \subseteq V$ *with* k *or fewer vertices such that each edge in* E *has at least one of its endpoints in* C.

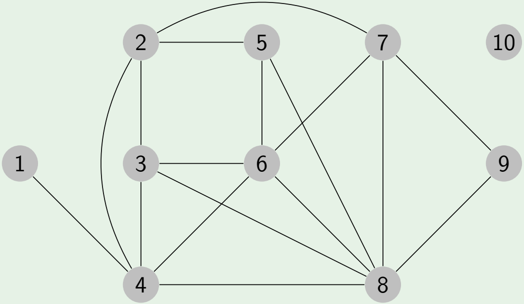
- VERTEX COVER is the most intensively studied problem in the FPT community

Buss's reduction:

- VC1 Remove all isolated vertices.
- VC2 For every degree-1 vertex, put the neighbour into the cover and delete both vertices from V.
- VC3 For a vertex with degree $> k$, put this vertex into the cover and delete it form the graph.

Example: Buss's reduction

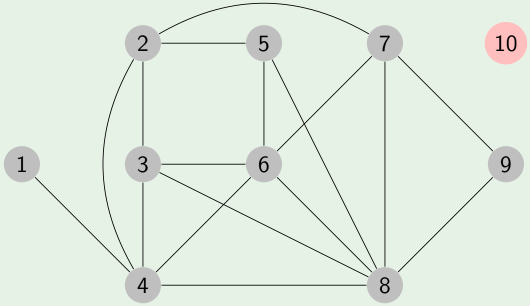
Example



Cover: $\{\}$ $k = 5$

Example: Buss's reduction

Example

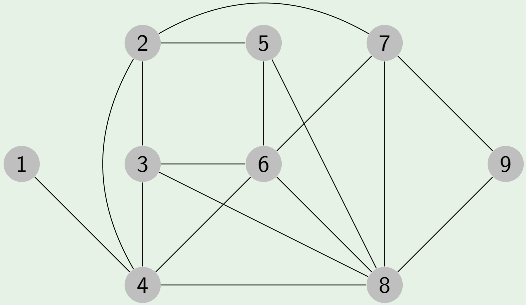


Cover: $\{\}$ $k = 5$

- VC1: vertex 10

Example: Buss's reduction

Example

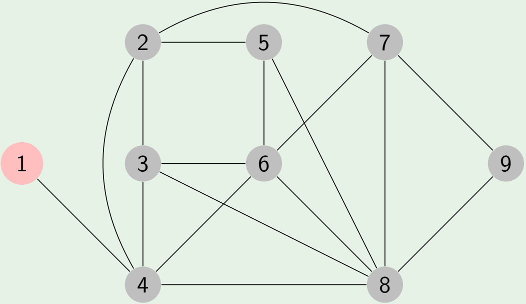


Cover: $\{\}$ $k = 5$

- VC1: vertex 10

Example: Buss's reduction

Example

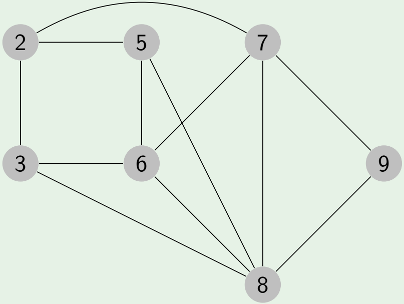


- VC1: vertex 10
- VC2: vertex 1

Cover: {} $k = 5$

Example: Buss's reduction

Example

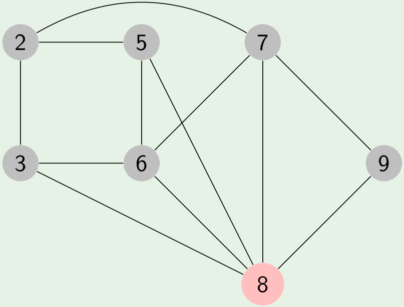


- VC1: vertex 10
- VC2: vertex 1

Cover: $\{4\}$ $k = 4$

Example: Buss's reduction

Example

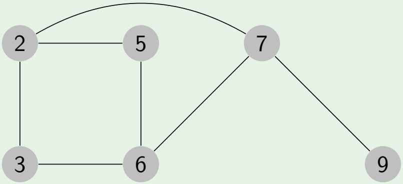


- VC1: vertex 10
- VC2: vertex 1
- VC3: vertex 8

Cover: $\{4\}$ $k = 4$

Example: Buss's reduction

Example

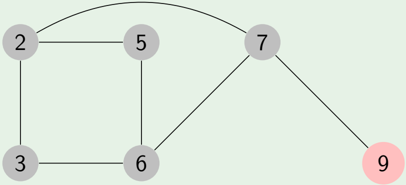


- VC1: vertex 10
- VC2: vertex 1
- VC3: vertex 8

Cover: $\{4, 8\}$ $k = 3$

Example: Buss's reduction

Example

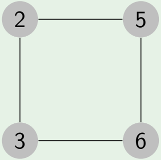


- VC1: vertex 10
- VC2: vertex 1
- VC3: vertex 8
- VC2: vertex 9

Cover: $\{4, 8\}$ $k = 3$

Example: Buss's reduction

Example



- VC1: vertex 10
- VC2: vertex 1
- VC3: vertex 8
- VC2: vertex 9

Cover: $\{4, 8, 7\}$ $k = 2$

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices
  - Only if (G, k) is a positive instances of VERTEX COVER

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices
  - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
    - $\leq k^2$ edges
    - $\leq k^2$ vertices
    - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices
  - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*
- Rule VC3 is *parameter-dependent*

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
    - $\leq k^2$ edges
    - $\leq k^2$ vertices
    - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*
- Rule VC3 is *parameter-dependent*
- Search solution in the remaining graph

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
    - $\leq k^2$ edges
    - $\leq k^2$ vertices
    - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*
- Rule VC3 is *parameter-dependent*
- Search solution in the remaining graph
    - Exhaustive search

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices
  - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*
- Rule VC3 is *parameter-dependent*
- Search solution in the remaining graph
  - Exhaustive search
  - Any other (exact) VERTEX COVER algorithm

Properties: Buss's reduction

- Apply rule VC1-VC3 exhaustively:
  - $\leq k^2$ edges
  - $\leq k^2$ vertices
  - Only if (G, k) is a positive instances of VERTEX COVER
- Can be done in $O(k \cdot |V|)$
- Rule VC1 and VC2 are *parameter-independent*
- Rule VC3 is *parameter-dependent*
- Search solution in the remaining graph
  - Exhaustive search
  - Any other (exact) VERTEX COVER algorithm
- Find at least *one* solution but not all

Conclusion problem kernels

- Data reductions and problem kernels are important

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice
- Some kernelization results are only of theoretical importance

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice
- Some kernelization results are only of theoretical importance
    - Parameter k is too big

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice
- Some kernelization results are only of theoretical importance
    - Parameter k is too big
    - The bound on the kernel size $g(k)$ is useless

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice
- Some kernelization results are only of theoretical importance
    - Parameter k is too big
    - The bound on the kernel size $g(k)$ is useless
- Proven problem kernels provide upper bounds

Conclusion problem kernels

- Data reductions and problem kernels are important
    - Not only for fixed-parameter algorithms
- Some data reduction can not be proven but work well in practice
- Some kernelization results are only of theoretical importance
    - Parameter k is too big
    - The bound on the kernel size $g(k)$ is useless
- Proven problem kernels provide upper bounds

Kernelizations can explain, and prove, why rules work so well in practice.
*(*Rolf Niedermeier, Friedrich-Schiller-Universität Jena*)*

Table of Contents

Search Trees

- Exhaustively search for a solution in a tree-like fashion

Search Trees

- Exhaustively search for a solution in a tree-like fashion
    - Used in many algorithms (e.g. in SAT-solving)

Search Trees

- Exhaustively search for a solution in a tree-like fashion
    - Used in many algorithms (e.g. in SAT-solving)
- Fixed-Parameter Algorithms: depth is bounded by k

Search Trees

- Exhaustively search for a solution in a tree-like fashion
    - Used in many algorithms (e.g. in SAT-solving)
- Fixed-Parameter Algorithms: depth is bounded by k
    - Small k leads to a small search tree

Search Trees

- Exhaustively search for a solution in a tree-like fashion
    - Used in many algorithms (e.g. in SAT-solving)
- Fixed-Parameter Algorithms: depth is bounded by k
    - Small k leads to a small search tree
- Can be combined with data reduction rules

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not
  - Search tree of size $O(2^n)$

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not
  - Search tree of size $O(2^n)$
- Fixed-parameter approach:

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not
  - Search tree of size $O(2^n)$
- Fixed-parameter approach:
  - By definition for each edge $\{v, w\} \in E$ one vertex *must* be in the cover

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph* $G = (V, E)$ *and a nonnegative integer* k*.*
Question: *Is there a subset of vertices* $C \subseteq V$ *with* k *or fewer vertices such that each edge in* E *has at least one of its endpoints in* C*.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not
  - Search tree of size $O(2^n)$
- Fixed-parameter approach:
  - By definition for each edge $\{v, w\} \in E$ one vertex *must* be in the cover
  - Branch on the edges

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
  - Either the vertex is in the cover or not
  - Search tree of size $O(2^n)$
- Fixed-parameter approach:
  - By definition for each edge $\{v, w\} \in E$ one vertex *must* be in the cover
  - Branch on the edges
  - Continue the search for a $k - 1$ cover in $G \setminus \{v\}$ and $G \setminus \{w\}$

Depth-bound search tree: VERTEX COVER

Problem (VERTEX COVER)

Instance: *A graph $G = (V, E)$ and a nonnegative integer k.*
Question: *Is there a subset of vertices $C \subseteq V$ with k or fewer vertices such that each edge in E has at least one of its endpoints in C.*

- Naïve approach: branch on vertex
    - Either the vertex is in the cover or not
    - Search tree of size $O(2^n)$
- Fixed-parameter approach:
    - By definition for each edge $\{v, w\} \in E$ one vertex *must* be in the cover
    - Branch on the edges
    - Continue the search for a $k - 1$ cover in $G \setminus \{v\}$ and $G \setminus \{w\}$
    - Search tree bounded by $O(2^k)$

Impr. Depth-bound search tree: VERTEX COVER

Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)

Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)

Impr. Depth-bound search tree: VERTEX COVER

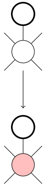
1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:

Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
   - either both neighbours are in the set

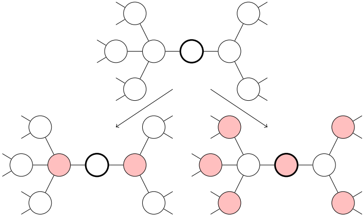
Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
   - either both neighbours are in the set
   - or v together with all the neighbours of the neighbours



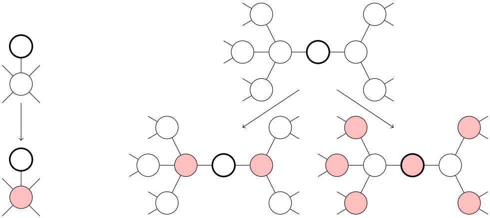
Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
    - either both neighbours are in the set
    - or v together with all the neighbours of the neighbours



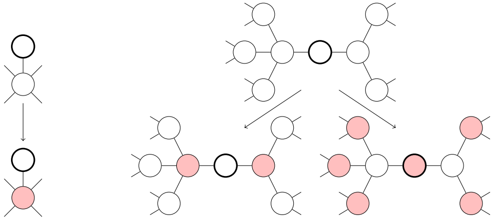
Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
    - either both neighbours are in the set
    - or v together with all the neighbours of the neighbours
3. Vertex v of degree at least 3:



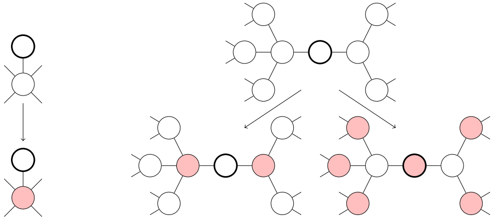
Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
   - either both neighbours are in the set
   - or v together with all the neighbours of the neighbours
3. Vertex v of degree at least 3:
   - either v is in the cover



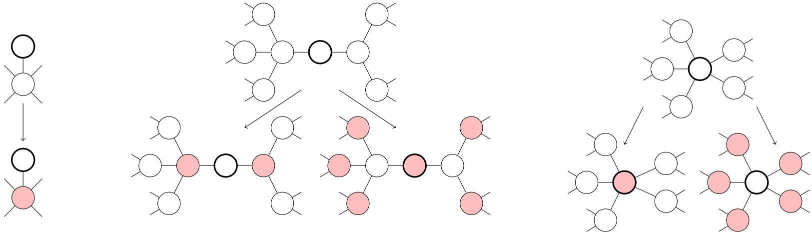
Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
   - either both neighbours are in the set
   - or v together with all the neighbours of the neighbours
3. Vertex v of degree at least 3:
   - either v is in the cover
   - or all its neighbours



Impr. Depth-bound search tree: VERTEX COVER

1. Vertex of degree 1: put the neighbour into the cover (like VC2)
2. Vertex v of degree 2:
   - either both neighbours are in the set
   - or v together with all the neighbours of the neighbours
3. Vertex v of degree at least 3:
   - either v is in the cover
   - or all its neighbours



Impr. Depth-bound search tree: VERTEX COVER (2)

- Finer case distinction

Impr. Depth-bound search tree: VERTEX COVER (2)

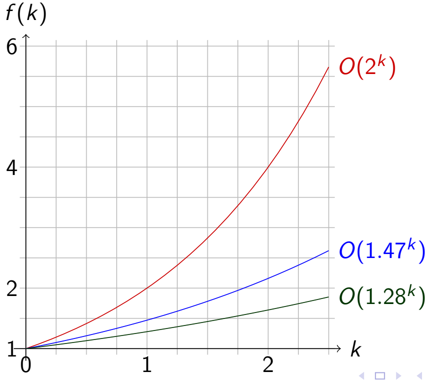
- Finer case distinction
- Search tree size $O(1.47^k)$

Impr. Depth-bound search tree: VERTEX COVER (2)

- Finer case distinction
- Search tree size $O(1.47^k)$
- *Best* search tree known to-date: $O(1.28^k)$

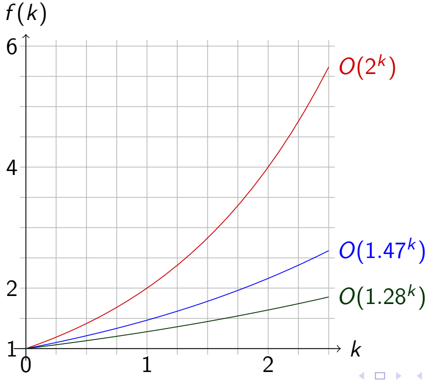
Impr. Depth-bound search tree: VERTEX COVER (2)

- Finer case distinction
- Search tree size $O(1.47^k)$
- *Best* search tree known to-date: $O(1.28^k)$



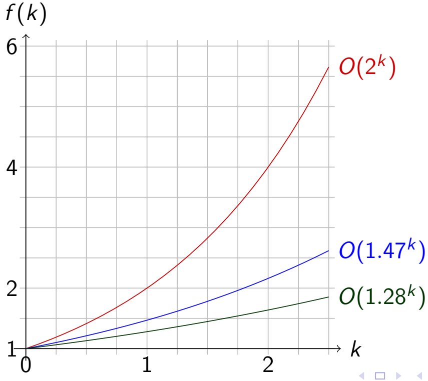
Impr. Depth-bound search tree: VERTEX COVER (2)

- Finer case distinction
- Search tree size $O(1.47^k)$
- *Best* search tree known to-date: $O(1.28^k)$
    - Even more extensive case distinction



Impr. Depth-bound search tree: VERTEX COVER (2)

- Finer case distinction
- Search tree size $O(1.47^k)$
- *Best* search tree known to-date: $O(1.28^k)$
    - Even more extensive case distinction
    - Organisational overhead hidden by $O(\cdot)$ notation



Conclusion search trees

- Branch on a small subset

Conclusion search trees

- Branch on a small subset
    - One of the elements *must* be in the solution

Conclusion search trees

- Branch on a small subset
    - One of the elements *must* be in the solution
- Shrink search tree with more involved case distinctions

Conclusion search trees

- Branch on a small subset
  - One of the elements *must* be in the solution
- Shrink search tree with more involved case distinctions
  - May decrease practical performance

Conclusion search trees

- Branch on a small subset
    - One of the elements *must* be in the solution
- Shrink search tree with more involved case distinctions
    - May decrease practical performance
    - *Computer aided* case distinctions

Conclusion search trees

- Branch on a small subset
    - One of the elements *must* be in the solution
- Shrink search tree with more involved case distinctions
    - May decrease practical performance
    - *Computer aided* case distinctions
- Combining with (interleaved) data reduction is very fruitful

Conclusion search trees

- Branch on a small subset
    - One of the elements *must* be in the solution
- Shrink search tree with more involved case distinctions
    - May decrease practical performance
    - *Computer aided* case distinctions
- Combining with (interleaved) data reduction is very fruitful

> *The art of case distinction.*
> *(*Rolf Niedermeier, Friedrich-Schiller-Universität Jena*)*

Table of Contents

Iterative Compression

Definition (Compression Routine)

A *compression routine* that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size.

Iterative Compression

Definition (Compression Routine)

A *compression routine* that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size.

- To find a solution iteratively call the compression routine

Iterative Compression

Definition (Compression Routine)

A *compression routine* that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size.

- To find a solution iteratively call the compression routine
- If the compression routine is fixed-parameter algorithm

Iterative Compression

Definition (Compression Routine)

A *compression routine* that, given a problem instance and a solution of size k, either calculates a smaller solution or proves that the given solution is of minimum size.

- To find a solution iteratively call the compression routine
- If the compression routine is fixed-parameter algorithm
    - → so is the whole algorithm

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:
    - Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:
    - Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.
    - Call the *compression routine* for $(G[V'], C)$.

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:
    - Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.
    - Call the *compression routine* for $(G[V'], C)$.
3. Output C.

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:
    - Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.
    - Call the *compression routine* for $(G[V'], C)$.
3. Output C.

- *Invariant:* C is always a minimal vertex cover for $G[V']$

Iterative Compression: VERTEX COVER

Algorithm

```
1 Set $V' := \emptyset$ and $C := \emptyset$.
2 For each vertex $v \in V$:
    • Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.
    • Call the compression routine for $(G[V'], C)$.
3 Output $C$.
```

- *Invariant:* C is always a minimal vertex cover for $G[V']$
- $C \cup \{v\}$ is a valid vertex cover for $G[V' \cup \{v\}]$

Iterative Compression: VERTEX COVER

Algorithm

1. Set $V' := \emptyset$ and $C := \emptyset$.
2. For each vertex $v \in V$:
    - Set $V' := V' \cup \{v\}$ and $C := C \cup \{v\}$.
    - Call the *compression routine* for $(G[V'], C)$.
3. Output C.

- *Invariant:* C is always a minimal vertex cover for $G[V']$
- $C \cup \{v\}$ is a valid vertex cover for $G[V' \cup \{v\}]$
- The compression routine yields the optimal solution for the subgraph

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
  - Some vertices remain in the cover $Y \subseteq C$

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
  - Some vertices remain in the cover $Y \subseteq C$
  - Other vertices $S := C \setminus Y$ are replaced

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
    - Y is already in the cover → remaining instance: $G[V' \setminus Y]$

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
    - Y is already in the cover → remaining instance: $G[V' \setminus Y]$
    - We do not take any vertices from S into the cover:

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
    - Y is already in the cover → remaining instance: $G[V' \setminus Y]$
    - We do not take any vertices from S into the cover:
        - If there is an edge with both endpoints in S abort

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
    - Y is already in the cover → remaining instance: $G[V' \setminus Y]$
    - We do not take any vertices from S into the cover:
        - If there is an edge with both endpoints in S abort
        - For all other edges: take the one endpoint that is not in S

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
    - Some vertices remain in the cover $Y \subseteq C$
    - Other vertices $S := C \setminus Y$ are replaced
        - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
    - Y is already in the cover → remaining instance: $G[V' \setminus Y]$
    - We do not take any vertices from S into the cover:
        - If there is an edge with both endpoints in S abort
        - For all other edges: take the one endpoint that is not in S

- Runtime compression routine: $O(2^{|C|}m)$

Compression Routine: VERTEX COVER

Algorithm

Input: cover C and graph $G[V']$

- C' is a *modification* of C
  - Some vertices remain in the cover $Y \subseteq C$
  - Other vertices $S := C \setminus Y$ are replaced
    - $|S| - 1$ new vertices from $V' \setminus C$
- *Idea:* search all $2^{|C|}$ partitions of C into Y and S
- For all partitions:
  - Y is already in the cover → remaining instance: $G[V' \setminus Y]$
  - We do not take any vertices from S into the cover:
    - If there is an edge with both endpoints in S abort
    - For all other edges: take the one endpoint that is not in S

- Runtime compression routine: $O(2^{|C|} m)$
- Runtime fixed-parameter algorithm: $O(2^k m \cdot n)$

Table of Contents

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
    - Table lookups

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
    - Table lookups
    - Bottom up vs. recursive calculation (e.g. *binomial coefficients*)

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
    - Table lookups
    - Bottom up vs. recursive calculation (e.g. *binomial coefficients*)
- **Example:** BINARY KNAPSACK (AD1) with parameter W weight

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
    - Table lookups
    - Bottom up vs. recursive calculation (e.g. *binomial coefficients*)
- **Example:** BINARY KNAPSACK (AD1) with parameter W weight
    - Runtime $O(W \cdot n)$

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
  - Table lookups
  - Bottom up vs. recursive calculation (e.g. *binomial coefficients*)
- **Example:** BINARY KNAPSACK (AD1) with parameter W weight
  - Runtime $O(W \cdot n)$
  - *pseudo-polynomial-time algorithm*

Dynamic Programming

- **Goal:** prevent recomputation by storing intermediate results
  - Table lookups
  - Bottom up vs. recursive calculation (e.g. *binomial coefficients*)
- **Example:** BINARY KNAPSACK (AD1) with parameter W weight
  - Runtime $O(W \cdot n)$
  - *pseudo-polynomial-time algorithm*
- Use dynamic programming to shrink depth-bounded search trees

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
  - e.g. VERTEX COVER, DOMINATING SET, ...

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
    - e.g. VERTEX COVER, DOMINATING SET, . . .
- What makes trees so nice and can this be extended to general graphs?

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
  - e.g. VERTEX COVER, DOMINATING SET, . . .
- What makes trees so nice and can this be extended to general graphs?
- *Treewidth:* measures how *tree-like* a graph is

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
    - e.g. VERTEX COVER, DOMINATING SET, . . .
- What makes trees so nice and can this be extended to general graphs?
- *Treewidth:* measures how *tree-like* a graph is
    - **Remark:** trees have a treewidth of 1

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
    - e.g. VERTEX COVER, DOMINATING SET, ...
- What makes trees so nice and can this be extended to general graphs?
- *Treewidth:* measures how *tree-like* a graph is
    - **Remark:** trees have a treewidth of 1
- Basic approach:

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
  - e.g. VERTEX COVER, DOMINATING SET, ...
- What makes trees so nice and can this be extended to general graphs?
- *Treewidth:* measures how *tree-like* a graph is
  - **Remark:** trees have a treewidth of 1
- Basic approach:
  - Find a *tree decomposition* of a graph

Tree Decomposition

- **Motivation:** many hard graph problems are easy on trees
    - e.g. VERTEX COVER, DOMINATING SET, ...
- What makes trees so nice and can this be extended to general graphs?
- *Treewidth:* measures how *tree-like* a graph is
    - **Remark:** trees have a treewidth of 1
- Basic approach:
    - Find a *tree decomposition* of a graph
    - Solve the problem on this *tree decomposition*

Table of Contents

Conclusion

In parameterized complexity the focus is on the question: What makes the problem computationally difficult? (R.G. Downey and M.R. Fellows)

Conclusion

> *In parameterized complexity the focus is on the question: What makes the problem computationally difficult?* (R.G. Downey and M.R. Fellows)

- Parameterized Complexity Theory can explain where the *hardness* of a problem comes from.

Conclusion

> *In parameterized complexity the focus is on the question: What makes the problem computationally difficult?* (R.G. Downey and M.R. Fellows)

- Parameterized Complexity Theory can explain where the *hardness* of a problem comes from.
- Fixed-parameter algorithms are narrowing the gap between theory and practice.

Conclusion

> *In parameterized complexity the focus is on the question: What makes the problem computationally difficult?* (R.G. Downey and M.R. Fellows)

- Parameterized Complexity Theory can explain where the *hardness* of a problem comes from.
- Fixed-parameter algorithms are narrowing the gap between theory and practice.
- Problem kernels and data reductions are important! Even outside FPAs.

Conclusion

> *In parameterized complexity the focus is on the question: What makes the problem computationally difficult?* (R.G. Downey and M.R. Fellows)

- Parameterized Complexity Theory can explain where the *hardness* of a problem comes from.
- Fixed-parameter algorithms are narrowing the gap between theory and practice.
- Problem kernels and data reductions are important! Even outside FPAs.
- Sometimes they can even explain why algorithms work in practice.

This is a subject that every computer scientist should know about. *(*Foinn Murtagh, University of London*)*

Thank You!

This is a subject that every computer scientist should know about. *(*Foinn Murtagh, University of London*)*