4 Fixed-Parameter Algorithms

14 May 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philippps-Universität Marburg

version `2025-05-14 09:58`

Outline

4 Fixed-Parameter Algorithms

Philosophy of FPT

- **Goal:** Principled theory for studying complexity based on two dimensions: input size $n = |x|$ (encoding length) and *some additional parameter k*
  - generalize ideas from $k = MaxInt(x)$
  - investigate influence of k (and n) on running time

Philosophy of FPT

- **Goal:** Principled theory for studying complexity based on two dimensions: input size $n = |x|$ (encoding length) and *some additional parameter* k
  - generalize ideas from $k = MaxInt(x)$
  - investigate influence of k (and n) on running time

$\rightsquigarrow$ Try to find a parameter k such that
(1) the problem can be solved efficiently as long as k is small, and
(2) practical instances have small values of k (even where n gets big).

Motivation: Satisfiability

Consider Satisfiability of CNF formula *the drosophila melanogaster of complexity theory*

- general worst case: NP-complete

$a \to b \equiv \neg a \vee b$

- k = #literals per clause
  - $k \le 2 \rightsquigarrow$ in P 2SAT $x_i \vee \neg x_j \equiv x_j \to x_i$
  - $k \ge 3$ NP-complete $\equiv \neg x_i \to \neg x_j$

Motivation: Satisfiability

Consider Satisfiability of CNF formula *the drosophila melanogaster of complexity theory*

- general worst case: NP-complete
- k = #literals per clause
  - $k \leq 2 \rightsquigarrow$ in P
  - $k \geq 3$ NP-complete
- k = #variables
  - $O(2^k \cdot n)$ time possible (try all assignments)
- k = #clauses?
- k = #literals?
- k = #ones in satisfying assignment
- k = structural property of formula
- for Max-SAT, k = #optimal clauses to satisfy

Parameters

Definition 4.1 (Parameterization)

Let Σ a (finite) alphabet. A *parameterization* (of $\Sigma^\star$) is a mapping $\kappa : \Sigma^\star \to \mathbb{N}$ that is polytime computable. ◀

Definition 4.2 (Parameterized problem)

A *parameterized (decision) problem* is a pair (L, κ) of a language $L \subset \Sigma^\star$ and a parameterization κ of $\Sigma^\star$. ◀

Definition 4.3 (Canonical Parameterizations)

We can often specify a parameterized problem conveniently as a language of *pairs* $L \subset \Sigma^\star \times \mathbb{N}$ with

$$(x, k) \in L \;\wedge\; (x, k') \in L \quad \to \quad k = k'$$

using the *canonical parameterization* $\kappa(x, k) = k$. ◀

Examples

As before: Typically leave encoding implicit.

Definition 4.4 (p-variables-SAT)

Given: formula boolean ϕ (same as before)

Parameter: number of variables

Question: Is there a satisfying assignment $v : [n] \to \{0, 1\}$? ◀

Definition 4.5 (p-Clique)

Given: graph $G = (V, E)$ and $k \in \mathbb{N}$

Parameter: k

Question: $\exists V' \subset V \quad : \quad |V'| \geq k \ \wedge \ \forall u, v \in V' : \{u, v\} \in E$? ◀

Canonical Parameterization

Definition 4.6 (Canonically Parameterized Optimization Problems)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ be an optimization problem.
Then $p\text{-}U$ denotes the *(canonically) parameterized (decision) problem* given by the threshold problem $Lang_U$. ◀

Recall: $Lang_U$ is the set of pairs (x, k) of all instances $x \in L_I$ that have solutions that are weakly "better" than k.

Canonical Parameterization

Definition 4.6 (Canonically Parameterized Optimization Problems)

Let $U = (\Sigma_I, \Sigma_O, L, L_I, M, cost, goal)$ be an optimization problem.
Then p-U denotes the *(canonically) parameterized (decision) problem* given by the threshold problem $Lang_U$.

Recall: $Lang_U$ is the set of pairs (x, k) of all instances $x \in L_I$ that have solutions that are weakly "better" than k.

Examples:

- p-CLIQUE
- p-VERTEX-COVER
- p-GRAPH-COLORING
- ...

Naming convention for other parameters:
p-*clause*-CNF-SAT: CNF-SAT with parameter "number of *clauses*"

4.1 Fixed-Parameter Tractability

Exemplary Running Times of Parameterized Problems

- p-*variables*-SAT (consider simplest brute-force methods for problems)
  - k variables, n length of formula
  - $\rightsquigarrow$ $O(2^k \cdot n)$ running time

Exemplary Running Times of Parameterized Problems

(consider simplest brute-force methods for problems)

- p-*variables*-SAT
  - k variables, n length of formula

  $\rightsquigarrow$ $O(2^k \cdot n)$ running time

- p-CLIQUE
  - k threshold (clique size); n vertices, m edges in graph

  $\rightsquigarrow$ $\binom{n}{k}$ candidates to check, each takes time $O(k^2)$ to check

  $\rightsquigarrow$ Total time $O(n^k \cdot k^2)$

$$\binom{n}{k} = \frac{\overbrace{n(n-1)(n-2)\cdots(n-k+1)}^{k}}{k!}$$

$$\sim \frac{n^k}{k!}$$

Exemplary Running Times of Parameterized Problems

(consider simplest brute-force methods for problems)

- p-*variables*-SAT
  - k variables, n length of formula
  - $\rightsquigarrow$ $O(2^k \cdot n)$ running time
- p-Clique
  - k threshold (clique size); n vertices, m edges in graph
  - $\rightsquigarrow$ $\binom{n}{k}$ candidates to check, each takes time $O(k^2)$ to check
  - $\rightsquigarrow$ Total time $O(n^k \cdot k^2)$
- p-VertexCover
  - k threshold (VC size); n vertices, m edges in graph
  - $\rightsquigarrow$ $\binom{n}{k}$ candidates to check, each takes time $O(m)$ to check
  - $\rightsquigarrow$ Total time $O(n^k \cdot m)$

Exemplary Running Times of Parameterized Problems

(consider simplest brute-force methods for problems)

- p-*variables*-SAT
  - k variables, n length of formula
  - $\rightsquigarrow$ $O(\underline{2^k} \cdot n)$ running time

$k = O(\log n) \rightsquigarrow$ poly time

- p-Clique
  - k threshold (clique size); n vertices, m edges in graph
  - $\rightsquigarrow$ $\binom{n}{k}$ candidates to check, each takes time $O(k^2)$ to check
  - $\rightsquigarrow$ Total time $O(\underline{n^k} \cdot k^2)$
- p-VertexCover
  - k threshold (VC size); n vertices, m edges in graph
  - $\rightsquigarrow$ $\binom{n}{k}$ candidates to check, each takes time $O(m)$ to check
  - $\rightsquigarrow$ Total time $O(\underline{n^k} \cdot m)$

$k = O(1) \rightsquigarrow$ poly time

- p-GraphColoring
  - k threshold (#colors); n vertices, m edges in graph
  - $\rightsquigarrow$ k^n candidates to check, each takes time $O(m)$
  - $\rightsquigarrow$ Total time $O(\underline{k^n} \cdot m)$

$k = 1 \rightsquigarrow$ polynomial

$k = 3$ NP-hard

FPT Running Time

Definition 4.7 (fpt-algorithm)

Let κ be a parameterization for $\Sigma^\star$.
A (deterministic) algorithm A (with input alphabet Σ) is a *fixed-parameter tractable algorithm (fpt-algorithm)* w.r.t. κ if its running time on $x \in \Sigma^\star$ with $\kappa(x) = k$ is at most

only depends on k

polynomial

$$f(k) \cdot p(|x|) = O\big(f(k) \cdot |x|^c\big)$$

where p is a polynomial of degree c and f is an **arbitrary** computable function. ◀

FPT Running Time

Definition 4.7 (fpt-algorithm)

Let κ be a parameterization for $\Sigma^\star$.
A (deterministic) algorithm A (with input alphabet Σ) is a *fixed-parameter tractable algorithm (fpt-algorithm)* w.r.t. κ if its running time on $x \in \Sigma^\star$ with $\kappa(x) = k$ is at most

$$f(k) \cdot p(|x|) \quad = \quad O\big(f(k) \cdot |x|^c\big)$$

where p is a polynomial of degree c and f is an **arbitrary** computable function. ◀

Definition 4.8 (FPT)

A parameterized problem (L, κ) is *fixed-parameter tractable* if there is an fpt-algorithm that decides it.
The complexity class of all such problems is denoted by FPT. ◀

Intuitively, FPT plays the role of P.

A First FPT Example

Theorem 4.9 (p-variables-SAT is FPT)

p-variables-SAT $\in$ FPT.

A First FPT Example

Theorem 4.9 (p-variables-SAT is FPT)

p-variables-SAT $\in$ FPT.

Proof:

Suffices to use brute force satisfiability for *p-variables*-SAT

```
procedure bruteForceSat(φ, 𝒳 = {x_1, ..., x_k})
    if k == 0
        if φ == true return ∅ else UNSATISFIABLE
    for value in {true, false} do
        A := {x_1 ↦ value}
        ψ := φ[x_1/value] // Substitute value for x_1
        B := bruteForceSat(ψ, {x_2, ..., x_k})
        if B ≠ UNSATISFIABLE
            return A ∪ B
```

. . . but #variables not usually small

A First FPT Example

Theorem 4.9 (p-variables-SAT is FPT)

p-variables-SAT ∈ FPT.

Proof:

Suffices to use brute force satisfiability for *p-variables*-SAT

```
procedure bruteForceSat(φ, 𝒳 = {x_1, ..., x_k})
    if k == 0
        if φ == true return ∅ else UNSATISFIABLE
    for value in {true, false} do
        A := {x_1 ↦ value}
        ψ := φ[x_1/value] // Substitute value for x_1
        B := bruteForceSat(ψ, {x_2, ..., x_k})
        if B ≠ UNSATISFIABLE
            return A ∪ B
```

$f(k)$

Worst case running time: $O(2^k n)$ for $n = |\varphi|$.

2^k recursive calls;

base case needs time $O(|\phi|)$ to check whether formula evaluates to *true* ∎

A First FPT Example

Theorem 4.9 (p-variables-SAT is FPT)

p-variables-SAT $\in$ FPT.

Proof:

Suffices to use brute force satisfiability for *p-variables*-SAT

```
procedure bruteForceSat(φ, 𝒳 = {x_1, ..., x_k})
    if k == 0
        if φ == true return ∅ else UNSATISFIABLE
    for value in {true, false} do
        A := {x_1 ↦ value}
        ψ := φ[x_1/value] // Substitute value for x_1
        B := bruteForceSat(ψ, {x_2, ..., x_k})
        if B ≠ UNSATISFIABLE
            return A ∪ B
```

Worst case running time: $O(2^k\, n)$ for $n = |\varphi|$.

2^k recursive calls;

base case needs time $O(|\phi|)$ to check whether formula evaluates to *true* ∎

…but #variables not usually small

Aren't we all FPT?

Theorem 4.10 (k never decreases → FPT)

Let $g : \mathbb{N} \to \mathbb{N}$ weakly increasing, unbounded and computable, and κ a parameterization with

$$\forall x \in \Sigma^\star : \kappa(x) \geq g(|x|).$$

Then $(L, \kappa) \in \mathsf{FPT}$ for *any* decidable L.

$g(x) = \log \log |x|$
possible

g weakly increasing: $n \leq m \;\to\; g(n) \leq g(m)$
g unbounded: $\forall t \; \exists n \;:\; g(n) \geq t$

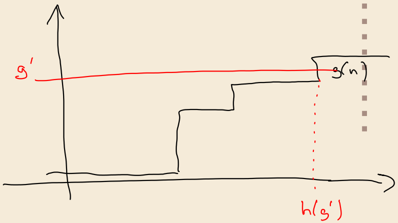
Proof: L decidable $\leadsto$ $\exists$ algorithm to decide L in time $\leq T(|x|)$

w.l.o.g. T weakly increasing

$T(|x|) \geq |x|$

Idea: "hide" $T(|x|)$ in $f(k)$

("g'") $h(g') = \max\{n' \in \mathbb{N} : g(n') \leq g'\} \cup \{1\}$



Aren't we all FPT? – Proof

Proof (cont.):

(1) g weakly incr. & unbounded $\implies$ h well-defined

(2) h weakly increasing

(3) g computable $\implies$ h computable

(4) $h(g(n)) \geq n$

time to decide whether $x \in \Sigma^*$ is in L

$n = |x|$

$k = \kappa(x) \geq g(n)$

$$\leq T(n) \underset{\substack{T \text{ incr.} \\ (4)}}{\leq} T(h(g(n))) \underset{T,\, h \text{ incr.}}{\leq} T(h(k)) =: f(k)$$

Back to “sensible” parameters

$\rightsquigarrow$ always check if parameter is reasonable (can be expected to be small)

- if not, FPT might not even mean in NP!

Back to "sensible" parameters

$\rightsquigarrow$ always check if parameter is reasonable (can be expected to be small)

  - if not, FPT might not even mean in NP!

- but now, for some positive examples!

4.2 Depth-Bounded Exhaustive Search I

FPT Design Pattern

- The simplest FPT algorithms use exhaustive search
- but with a search tree bounded by $f(k)$

FPT Design Pattern

- The simplest FPT algorithms use exhaustive search
- but with a search tree bounded by $f(k)$

- bruteforceSat was a typical example!
- does this work on other problems?

Depth-Bounded Search for Vertex Cover

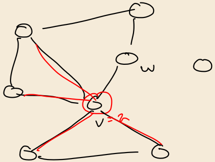
Let's try p-VERTEXCOVER. brute force $\binom{n}{k} \cdot \mathrm{poly}(n) = \Theta(n^k \, \mathrm{poly}(n)) \neq$ fpt running time

Key insight: for every edge $\{v, w\}$, any vertex cover must contain v or w

Depth-Bounded Search for Vertex Cover

Let's try p-VERTEXCOVER.
Key insight: for every edge $\{v, w\}$, any vertex cover must contain v or w

procedure simpleFptVertexCover($G = (V, E)$, k):
    if $E == \emptyset$ **then return** $\emptyset$
    if $k == 0$ **then return** `NOT_POSSIBLE` *// truncate search*
    Choose $\{v, w\} \in E$ (arbitrarily)
    for u in $\{v, w\}$ **do**:
        $G_u := \big(V \setminus \{u\}, E \setminus \{\{u, x\} \in E\}\big)$ *// Remove u from G*
        $C_u :=$ simpleFptVertexCover(G_u, $k - 1$)
    if $C_v ==$ `NOT_POSSIBLE` **then return** $C_w \cup \{w\}$
    if $C_w ==$ `NOT_POSSIBLE` **then return** $C_v \cup \{v\}$
    if $|C_v| \le |C_w|$ **then return** $C_v \cup \{v\}$ **else return** $C_w \cup \{w\}$



Depth-Bounded Search for Vertex Cover

Let's try p-VERTEXCOVER.

Key insight: for every edge $\{v, w\}$, any vertex cover must contain v or w

procedure simpleFptVertexCover($G = (V, E)$, k):
  if $E == \emptyset$ **then return** $\emptyset$
  if $k == 0$ **then return** `NOT_POSSIBLE` *// truncate search*
  Choose $\{v, w\} \in E$ (arbitrarily)
  for u in $\{v, w\}$ **do**:
    $G_u := \big(V \setminus \{u\}, E \setminus \{\{u, x\} \in E\}\big)$ *// Remove u from G*
    $C_u :=$ simpleFptVertexCover(G_u, $k - 1$)
  if $C_v ==$ `NOT_POSSIBLE` **then return** $C_w \cup \{w\}$
  if $C_w ==$ `NOT_POSSIBLE` **then return** $C_v \cup \{v\}$
  if $|C_v| \le |C_w|$ **then return** $C_v \cup \{v\}$ **else return** $C_w \cup \{w\}$

- Does not need explicit checks of solution candidates!
- runs in time $O\big(2^k(n + m)\big)$ $\rightsquigarrow$ fpt-algorithm for p-VERTEX-COVER $\in$ FPT

Guessing the parameter

- Note: Previous algorithm only uses k to *truncate* branches.

$\rightsquigarrow$ We can *guess* a k and it still works

Guessing the parameter

- Note: Previous algorithm only uses k to *truncate* branches.

⇝ We can *guess* a k and it still works

⇝ Try all k!

```
procedure vertexCoverBfs(G = (V, E))
    for k := 0, 1, ..., |V| do
        C := simpleFptVertexCover(G, k)
        if C ≠ NOT_POSSIBLE return C
```

Guessing the parameter

- Note: Previous algorithm only uses k to *truncate* branches.

⇝ We can *guess* a k and it still works

⇝ Try all k!

```
procedure vertexCoverBfs(G = (V, E))
    for k := 0, 1, ..., |V| do
        C := simpleFptVertexCover(G, k)
        if C ≠ NOT_POSSIBLE return C
```

- Running time: $\displaystyle\sum_{k'=0}^{k} O(2^{k'}(n+m)) = O(2^k(n+m))$

⇝ For exponentially growing cost, trying all values up to k costs only constant factor more

4.3 Problem Kernels

Preprocessing

- Second key fpt technique are *reduction rules*
- **Idea:** Reduce the size of the instance (in polytime) without changing its outcome

Preprocessing

- Second key fpt technique are *reduction rules*
- **Idea:** Reduce the size of the instance (in polytime) without changing its outcome
- Trivial example for SAT:

  If a CNF formula contains a single-literal clause $\{x\}$ resp. $\{\neg x\}$, set x to *true* resp. *false* and remove the clause.

  - doesn't do anything in the worst case ...

Preprocessing

- Second key fpt technique are *reduction rules*
- **Idea:** Reduce the size of the instance (in polytime)
  without changing its outcome

- Trivial example for SAT:

  If a CNF formula contains a single-literal clause $\{x\}$ resp. $\{\neg x\}$,
  set x to *true* resp. *false* and remove the clause.

  - doesn't do anything in the worst case ...
  - special case of resolution calculus rule $\dfrac{a_1 \vee a_2 \vee \cdots \vee x, \quad b_1 \vee b_2 \vee \cdots \vee \neg x}{a_1 \vee a_2 \vee \cdots \vee b_1 \vee b_2 \vee \cdots}$
  - basis of practical SAT solvers

Preprocessing

- Second key fpt technique are *reduction rules*
- **Idea:** Reduce the size of the instance (in polytime) without changing its outcome
- Trivial example for SAT:

  If a CNF formula contains a single-literal clause $\{x\}$ resp. $\{\neg x\}$, set x to *true* resp. *false* and remove the clause.

  - doesn't do anything in the worst case ...
  - special case of resolution calculus rule $\dfrac{a_1 \vee a_2 \vee \cdots \vee x, \quad b_1 \vee b_2 \vee \cdots \vee \neg x}{a_1 \vee a_2 \vee \cdots \vee b_1 \vee b_2 \vee \cdots}$
  - basis of practical SAT solvers
- Trivial example for VertexCover

  Remove vertices of degree 0 or 1.

(never needed as part of optimal VC)

Preprocessing

- Second key fpt technique are *reduction rules*
- **Idea:** Reduce the size of the instance (in polytime)
  without changing its outcome

- Trivial example for SAT:

  > If a CNF formula contains a single-literal clause $\{x\}$ resp. $\{\neg x\}$,
  > set x to *true* resp. *false* and remove the clause.

  - doesn't do anything in the worst case ...
  - special case of resolution calculus rule $\dfrac{a_1 \vee a_2 \vee \cdots \vee x, \quad b_1 \vee b_2 \vee \cdots \vee \neg x}{a_1 \vee a_2 \vee \cdots \vee b_1 \vee b_2 \vee \cdots}$
  - basis of practical SAT solvers
- Trivial example for VertexCover

  > Remove vertices of degree 0 or 1.

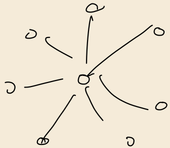
  (never needed as part of optimal VC)
- Here: reduction rules that provably shrink an instance to size $g(k)$

Buss's Reduction Rule for VC

- Given a p-VertexCover instance (G, k)

Buss's reduction: If G contains vertex v of degree $\deg(v) > k$, include v in potential solution and remove it from the graph.

- Can apply this simultaneously to degree $> k$ vertices.
- Either rule applies, or all vertices bounded degree(!)



Kernels

Definition 4.11 (Kernelization)

Let (L, κ) be a parameterized problem. A function $K : \Sigma^\star \to \Sigma^\star$ is *kernelization* of L w.r.t. κ if it maps any $x \in L$ to an instance $x' = K(x)$ with $k' = \kappa(x')$ so that

1. (self-reduction) $x \in L \iff x' \in L$
2. (polytime) K is computable in polytime.
3. (kernel-size) $|x'| \le g(k)$ for some computable function g

We call x' the *(problem) kernel* of x and g the *size of the problem kernel*. ◀

Buss's Kernel

Theorem 4.12 (Buss's Reduction is Kernelization)

Buss' reduction yields a kernelization for p-Vertex-Cover with kernel size $O(k^2)$. ◀

Buss's Kernel

Buss' rule for deg > k and 0/1 deg.

Theorem 4.12 (Buss's Reduction is Kernelization)

Buss' reduction yields a kernelization for p-VERTEX-COVER with kernel size $O(k^2)$.

Proof:

After repeatedly applying Buss's rule as well as the isolated/leaf rule until neither applies further, we have $\forall v \in V : 2 \leq \deg(v) \leq k$.

(Note that the rule might reduce the parameter k).

Buss's Kernel

Theorem 4.12 (Buss's Reduction is Kernelization)

Buss' reduction yields a kernelization for p-Vertex-Cover with kernel size $O(k^2)$. ◀

Proof:

After repeatedly applying Buss's rule as well as the isolated/leaf rule until neither applies further, we have $\forall v \in V : 2 \leq \deg(v) \leq k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\leq k$ covers $\leq k^2$ edges.

Buss's Kernel

Theorem 4.12 (Buss's Reduction is Kernelization)

Buss' reduction yields a kernelization for p-Vertex-Cover with kernel size $O(k^2)$.

Proof:

After repeatedly applying Buss's rule as well as the isolated/leaf rule until neither applies further, we have $\forall v \in V : 2 \leq \deg(v) \leq k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\leq k$ covers $\leq k^2$ edges.
If $m > k^2$, we output a trivial No-instance (e. g., a K_{k+1} a complete graph on $k+1$ vertices).

Buss's Kernel

Theorem 4.12 (Buss's Reduction is Kernelization)
Buss' reduction yields a kernelization for p-VERTEX-COVER with kernel size $O(k^2)$. ◀

Proof:
After repeatedly applying Buss's rule as well as the isolated/leaf rule until neither applies further, we have $\forall v \in V : 2 \leq \deg(v) \leq k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\leq k$ covers $\leq k^2$ edges.
If $m > k^2$, we output a trivial No-instance (e. g., a K_{k+1} a complete graph on $k+1$ vertices).
If $m \leq k^2$, then the input size is now bounded by $g(k) = 2k^2$. ■

FPT iff Kernelization

Theorem 4.13 (FPT ↔ kernel)

A computable, parameterized problem (L, κ) is fixed-parameter tractable if and only if there is a kernelization for L w.r.t. κ.

Proof:

"$\Leftarrow$" kernelization K for (L, κ) given,

L has decider A of running time $T(n)$ (w.l.o.g. weakly increasing)

(1) $x \in \Sigma^*$ to check $x \in L$ $\quad k = \kappa(x) \quad n = |x|$

compute $K(x) = x'$ $\quad$ poly time

$|x'| \le g(k)$

(2) run A on x'

time $T(|x'|) \underset{\text{incr.}}{\le} T(g(k)) =: f(k)$

$\Rightarrow$ algorithm for (L, κ) w/ time $O(f(k) + \mathrm{poly}(n))$

FPT iff Kernelization [2]

Proof (cont.):

"$\Rightarrow$" Given fpt-algorithm A for (L, x) with time $\leq f(k) \cdot n^c$

(1) Simulate A for $\leq n^{c+1}$ steps (poly time)

(2) • If A terminated

if output Yes : output trivial Yes-instance

if " No - - - No - "

• otherwise $n^{c+1} \leq f(k)\, n^c \Rightarrow n \leq f(k)$

$\Rightarrow$ output original input