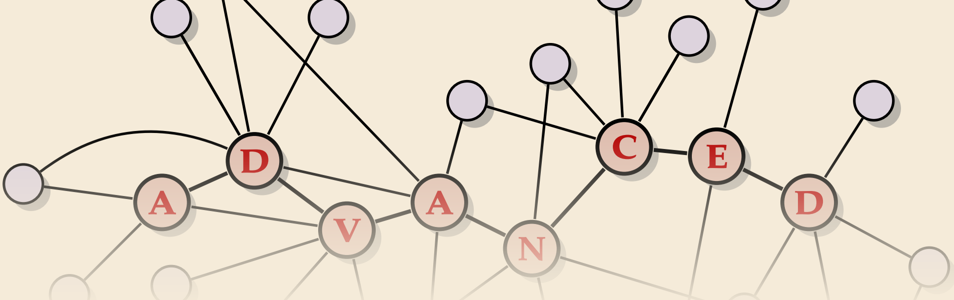


4 Fixed-Parameter Algorithms

14 May 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version `2025-05-20 17:26`

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 $k = \mathit{MaxInt}(x)$
  - 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 $k = \mathit{MaxInt}(x)$
  - 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
  - $k \leq 2 \rightsquigarrow$ in P 2SAT $x_i \vee \neg x_j \equiv x_j \to x_i$
  - $k \geq 3$ 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-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
  - $\leadsto$ $O(2^k \cdot n)$ running time

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

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

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

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

$k=1 \leadsto$ 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 ∈ 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 $\in$ 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
```

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($\varphi, \mathcal{X} = \{x_1, \ldots, x_k\}$)
    if $k == 0$
        if $\varphi == \mathit{true}$ **return** $\emptyset$ **else** `UNSATISFIABLE`
    for *value* in $\{\mathit{true}, \mathit{false}\}$ **do**
        $A := \{x_1 \mapsto \mathit{value}\}$
        $\psi := \varphi[x_1/\mathit{value}]$ // *Substitute value for* x_1
        $B :=$ bruteForceSat($\psi, \{x_2, \ldots, x_k\}$)
        if $B \neq$ `UNSATISFIABLE`
            return $A \cup B$

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: $\forall t\; \exists n \;:\; g(n) \geq t$

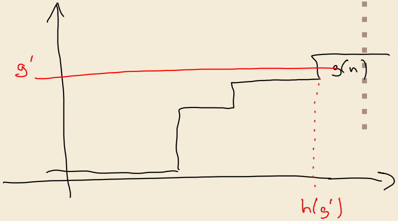
Proof: L decidable $\leadsto$ $\exists$ algorithm to decide L in time $\leq T(|x|)$

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 $\Rightarrow$ h well-defined

(2) h weakly increasing

(3) g computable $\Rightarrow$ h computable

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

time to decide whether $x \in \Sigma^*$ is in L $\qquad$ $n = |x|$

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

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

∎

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**:

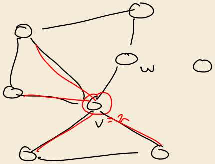
        $G_u := \big(V \setminus \{u\}, E \setminus \{\{u, x\} \in E\}\big)$ *// Remove u from G*

        $C_u :=$ 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.

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

Guessing the parameter

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

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

$\rightsquigarrow$ 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: $\sum_{k'=0}^{k} O(2^{k'}(n+m)) = O(2^k(n+m))$

⇝ 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.

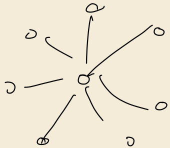
  (never needed as part of optimal VC)
- Here: reduction rules that provably shrink an instance to size $g(k)$

Buss's Reduction Rule for VC

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

"deg > k" Rule: 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

Buss's Reduction for Vertex Cover: (repeatedly apply until no more changes)

- $\deg > k$ rule
- Remove degree 0 and 1 vertices

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.

Buss's Reduction for Vertex Cover: (repeatedly apply until no more changes)

- $\deg > k$ rule
- Remove degree 0 and 1 vertices

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

Buss's Reduction for Vertex Cover: (repeatedly apply until no more changes)

- $\deg > k$ rule
- Remove degree 0 and 1 vertices

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 $\forall v \in V : 2 \le \deg(v) \le k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\le k$ covers $\le k^2$ edges.

Buss's Kernel

Buss's Reduction for Vertex Cover: (repeatedly apply until no more changes)

- $\deg > k$ rule
- Remove degree 0 and 1 vertices

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 $\forall v \in V : 2 \le \deg(v) \le k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\le k$ covers $\le k^2$ 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

Buss's Reduction for Vertex Cover: (repeatedly apply until no more changes)

- $\deg > k$ rule
- Remove degree 0 and 1 vertices

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 $\forall v \in V : 2 \le \deg(v) \le k$.
(Note that the rule might reduce the parameter k).
In the resulting graph, any VC of size $\le k$ covers $\le k^2$ 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 $m \le k^2$, 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 $K(x) = x'$ 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 $O(f(k) + \text{poly}(n))$

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

Max-SAT Kernel

k = # clauses to satisfy

Theorem 4.14 (Kernel for Max-SAT)

p-MAX-SAT has a problem kernel of size $O(k^2)$ which can be constructed in linear time. ◀

Proof:

$$(x \vee y \vee \bar{z}) \wedge (x \vee y \vee \bar{z}) \wedge (\bar{x} \vee z) \wedge (\bar{y} \vee z)$$

$$\rightsquigarrow \{ \{x, y, \bar{z}\}, \{x, y, \bar{z}\}, \{\bar{x}, z\}, \{\bar{y}, z\} \}$$

assumption: each variables occurs at most once per clause
($x \vee \bar{x} \rightsquigarrow$ delete clause)

m = # clauses (n total input size # literals in all clauses

Case 1: $k \leq \lfloor \frac{m}{2} \rfloor$ (output Yes)

pick arbitrary assignment A of all variables

under A, ℓ clauses are satisfied $\ell \geq k$ ✓

if $\ell < k \leq \lfloor \frac{m}{2} \rfloor$ $\rightarrow$ then $\bar{A}$ (inverse assignment) satisfies $m - \ell \geq \lfloor \frac{m}{2} \rfloor \geq k$ 21

Max-SAT Kernel [2]

Proof (cont.):

Case $k > \lfloor \frac{m}{2} \rfloor \Rightarrow k > \frac{m}{2} \Rightarrow \boxed{m < 2k}$

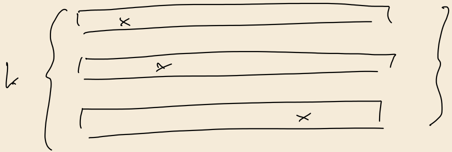
$\Rightarrow$ few clauses, but they could be big

consider $F_L = \{ \text{clauses } C : C \text{ has} \geq k \text{ literals} \}$

$F_S = \{ \text{clauses } C : \text{———} < k \text{———} \}$

$|F_L| = \boxed{L \geq k}$

$\Rightarrow$ Yes instance



each has k variables

can pick unique variable per clause to satisfy it

$\boxed{L < k}$ consider $(F_S, k-L)$ if that is a Yes instance

$\Leftrightarrow (F, k)$ is a Yes instance

Max-SAT Kernel [3]

Proof (cont.): "$\Rightarrow$" If A satisfies $k-L$ clauses in F_S

each of the clauses contains a true literal $\leadsto$ $k-L$

$\leadsto$ A only "fixes" $k-L$ variables

$\Rightarrow$ for L long clauses can find L unique variables

that are not fixed in A

"$\Leftarrow$" trivial.

$\Rightarrow$ reduced problem to $(F_S, k-L)$

at most $m-L \leq m$ clauses each has $< k$ literals

$< 2k$

$\Rightarrow$ # literals $= O(k^2)$ (encoding $O(k^2 \log k)$)

Corollary 4.15

p-MAX-SAT $\in$ FPT

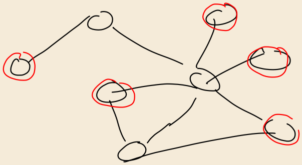
4.4 Depth-Bounded Search II: Planar Independent Set

Deeper results (towards more shallow trees)

- Our previous examples of depth-bounded search were basically brute force
- Here we will see two more examples that exploit the problem structure in more interesting ways

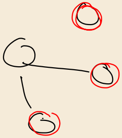
Independent Set on Planar Graphs

~~Recall~~ We will see: general problem p-Independent-Set is $\mathcal{W}[1]$-hard.



Definition 4.16 (p-Planar-Independent-Set)

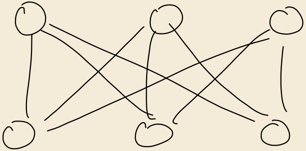
Given: a *planar* graph $G = (V, E)$ and $k \in \mathbb{N}$
Parameter: k
Question: $\exists V' \subset V \; : \; |V'| \geq k \wedge \forall u, v \in V' : \{u, v\} \notin E$?



planar graph G :

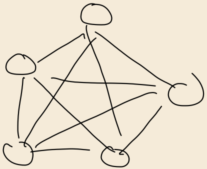
$\exists$ embedding (placement) of vertices in $\mathbb{R}^2$

and a drawing of edges without crossings



$K_{3,3}$ not planar

K_5 not planar



Independent Set on Planar Graphs

Recall: general problem p-Independent-Set is $\mathcal{W}[1]$-hard.

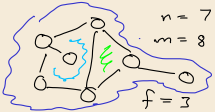
Definition 4.16 (p-Planar-Independent-Set)

Given: a *planar* graph $G = (V, E)$ and $k \in \mathbb{N}$
Parameter: k
Question: $\exists V' \subset V \quad : \quad |V'| \geq k \ \wedge \ \forall u, v \in V' : \{u, v\} \notin E$? ◀

Theorem 4.17 (Depth-Bounded Search for Planar Independent Set)

p-Planar-Independent-Set is in FPT and can be solved in time $O(6^k n)$. ◀

Elementary Knowledge on Planar Graphs



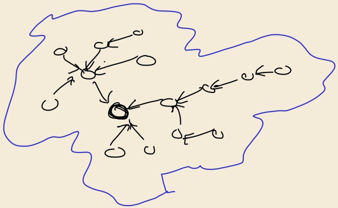
Theorem 4.18 (Euler's formula)

faces

In any finite, connected planar graph G with n nodes, m edges, f holds $n - m + f = 2$. ◀

$7 - 8 + 3 = 2$

Proof idea: Induction on f



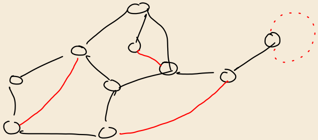
faces = regions of $\mathbb{R}^2$

IB $f = 1$ $\Rightarrow$ G is a tree

$\Rightarrow$ $n = m + 1$

IS "add a new face"

m++ f++



Elementary Knowledge on Planar Graphs

Theorem 4.18 (Euler's formula)

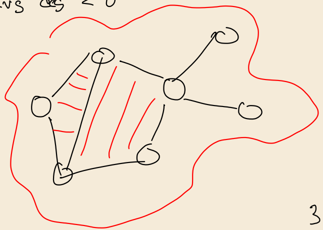
In any finite, connected planar graph G with n nodes, m edges f holds $n - m + f = 2$.

Corollary 4.19

A simple planar graph G on $n \geq 3$ nodes has $m \leq 3n - 6$ edges.
The average degree in G is < 6.

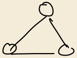
$\sum_v \deg(v) = 2m \leq 6n - 12$

avg deg < 6



↯ not simple

↯ not simple



simple $\Rightarrow$ every face is delimited by ≥ 3 edges

$3f$ double counts each edge at most twice

$3f \leq 2m$

$3(2 - n + m) = 6 - 3n + 3m$ $\quad\Big|\; -2m \;\; +3n - 6$

$m \leq 3n - 6$

avg deg < 6 $\Rightarrow$ $\boxed{\text{in any planar graph, } \exists v : \deg(v) \leq 5}$

"degeneracy" $d = 5$

always find vertex of degree $\leq d$ in G

and in any induced subgraph

induced subgraph

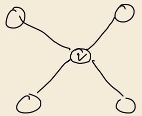
$G = (V, E)$ $\quad G[V'] = \left(V', \{\{u, v\} : u, v \in V', \{u, v\} \in E\}\right)$

$V' \subseteq V$

Depth-Bounded Search for Planar Independent Set

---

procedure planarIndependentSet($G = (V, E)$, k):
  if $k == 0$ **then return** $\emptyset$
  if $k > |V|$ **then return** `NOT_POSSIBLE` // *truncate search*
  Choose $v \in V$ with minimal degree; let $w_1, \ldots, w_d$ be v's neighbors
  // *By planarity, we know* $d \leq 5$.
  for u in $\{v, w_1, \ldots, w_d\}$ **do**
    $D := \{u\} \cup N(u)$ — neighbors of u $G_u = G[V \setminus D]$
    $G_u := \big(V \setminus D, E \setminus \{\{x,y\} \in E : x \in D\}\big)$ // *Delete* u *and its neighbors*
    $I_u := \{u\} \cup$ planarIndependentSet(G_u, $k-1$)
  return largest I_u or `NOT_POSSIBLE` **if** none exists



any maximal indep. set — can't add more vertices to this set

(if none of v's neighbors in the set, could add v)

≤ 6 recursive calls

in w.c. recurse until $k=0$

$\Rightarrow 6^k$ recursive calls in total

each take $\Theta(n+m) = \Theta(n)$

$\Rightarrow$ total time $O(6^k \cdot n)$

Summary Planar Independent Set

- Note: IndependentSet is NP-hard on planar graphs even with vertex degrees at most 3
- planarIndependentSet will often be faster than $O(6^k n)$
- works unchanged in $O((d+1)^k n)$ time for any degeneracy-d graph

every (induced) subgraph has vertex of degree at most d

4.5 Depth-Bounded Search III: Closest String

Closest String

Definition 4.20 (p-Closest-String)

Given: S set of m strings $s_1, s_2, \dots, s_m$ of length L over alphabet Σ and a $k \in \mathbb{N}$.
Parameter: k
Question: Is there a string s for which $d_H(s, s_i) \leq k$ holds for all $i = 1, \dots, m$?

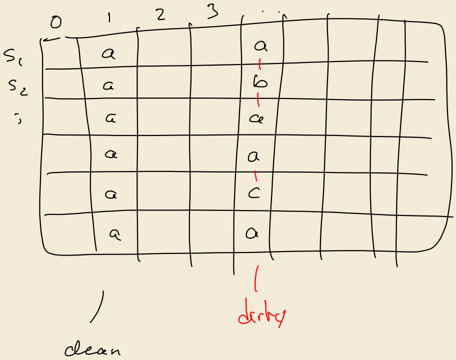
$d_H = 4$

mismatched positions

Dirty Columns

Definition 4.21 (Dirty Column)

A column of the $m \times L$ matrix corresponding to m strings of length L is called *dirty* if it contains at least 2 different symbols.



Dirty Columns

Definition 4.21 (Dirty Column)

A column of the $m \times L$ matrix corresponding to m strings of length L is called *dirty* if it contains at least 2 different symbols. ◀

Lemma 4.22 (Many Dirty Columns → No)

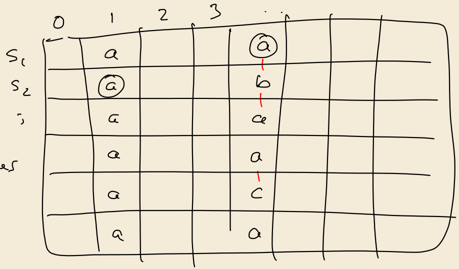
Let an instance to Closest-String with m strings of length L and parameter k be given.
If the corresponding $m \times L$ matrix contains more than $m \cdot k$ dirty columns, then no solution for the given instance exists. ◀

If we have $> m \cdot k$ dirty cols

no matter what s,

one s_i must have $\geq k+1$ mismatches

↯



Depth-Bounded Search for Closest String

procedure closestStringFpt(s, d):
  if $d < 0$ **then return** `NOT_POSSIBLE`
  if $d_H(s, s_i) > k + d$ **for an** $i \in \{1, \ldots, m\}$ **then**
    return `NOT_POSSIBLE`
  if $d_H(s, s_i) \le k$ **for all** $i = 1, \ldots, m$ **then return** s
  Choose $i \in \{1, \ldots, m\}$ arbitrarily with $d_H(s, s_i) > k$
    $P := \{p : s[p] \neq s_i[p]\}$
    Choose arbitrary $P' \subseteq P$ with $|P'| = k + 1$
    for p in P' **do**
      $s' := s$
      $s'[p] := s_i[p]$
      $s_{ret} :=$ closestStringFpt(s', $d - 1$)
      if $s_{ret} \neq$ `NOT_POSSIBLE` **then return** s_{ret}
  return `NOT_POSSIBLE`

next slide

search space $(k+1)^k = O(k^k)$

- initial call closestStringFpt(s_1, k)

Too Much Dirt

Lemma 4.23 (Pair Too Different → No)

Let $S = \{s_1, s_2, \ldots, s_m\}$ a set of strings and $k \in \mathbb{N}$. If there are $i, j \in \{1, \ldots, m\}$ with $d_H(s_i, s_j) > 2k$, then there is no string s with $\max_{1 \le i \le m} d_H(s, s_i) \le k$. ◀