Introduction to solving intractable problems

Serge Gaspers

UNSW

Outline

1. Algorithms for NP-hard problems
2. Exponential Time Algorithms
3. Parameterized Complexity
   - FPT Algorithm for Vertex Cover
   - Algorithms for Vertex Cover
4. Further Reading

Outline

1. Algorithms for NP-hard problems
2. Exponential Time Algorithms
3. Parameterized Complexity
   - FPT Algorithm for Vertex Cover
   - Algorithms for Vertex Cover
4. Further Reading

Central question

P vs. NP

NP-hard problems

- no known polynomial time algorithm for any NP-hard problem
- belief: $\mathsf{P} \neq \mathsf{NP}$
- What to do when facing an NP-hard problem?

Example problem

Monitoring a power grid

Tammy is responsible for fault detection on the power grid of an energy company.

She has access to k monitoring devices. Each one can be placed on a node of the electrical grid and can monitor the power lines that are connected to this node.

Tammy's objective is to place the monitoring devices in such a way that each power line is monitored by at least one monitoring device.

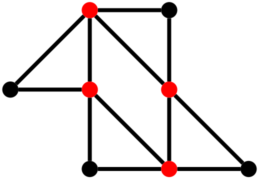
Let us first give an abstraction of this problem and formulate it as a decision problem for graphs.

Example problem: Vertex Cover

A vertex cover in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every edge of G has an endpoint in S.

Vertex Cover	
Input:	Graph G, integer k
Question:	Does G have a vertex cover of size k?

Note: Vertex Cover is NP-complete.



Coping with NP-hardness

- Approximation algorithms
  - There is a polynomial-time algorithm, which, given a graph G, finds a vertex cover of G of size at most $2 \cdot \mathsf{OPT}$, where OPT is the size of a smallest vertex cover of G.
- Exact exponential time algorithms
  - There is an algorithm solving $\textsc{Vertex Cover}$ in time $O(1.1970^n)$, where $n = |V|$ (Xiao and Nagamochi, 2017).
- Fixed parameter algorithms
  - There is an algorithm solving $\textsc{Vertex Cover}$ in time $O(1.2738^k + kn)$ (Chen, Kanj, and Xia, 2010).
- Heuristics
  - The COVER heuristic (COVer Edges Randomly) finds a smaller vertex cover than state-of-the-art heuristics on a suite of hard benchmark instances (Richter, Helmert, and Gretton, 2007).
- Restricting the inputs
  - $\textsc{Vertex Cover}$ can be solved in polynomial time on bipartite graphs, trees, interval graphs, etc. (Golumbic, 2004).
- Quantum algorithms?
  - Not believed to solve NP-hard problems in polynomial time (Aaronson, 2005). Quadratic speedup possible in some cases.

Aims of this course

Design and analyze algorithms for NP-hard problems.

We focus on algorithms that solve NP-hard problems exactly and analyze their worst case running time.

Outline

Running times

Worst case running time of an algorithm.

- An algorithm is polynomial if $\exists c \in \mathbb{N}$ such that the algorithm solves every instance in time $O(n^c)$, where n is the size of the instance.
  Also: $n^{O(1)}$ or $\mathsf{poly}(n)$.
- quasi-polynomial: $2^{O(\log^c n)}$, $c \in O(1)$
- sub-exponential: $2^{o(n)}$
- exponential: $2^{\mathsf{poly}(n)}$
- double-exponential: $2^{2^{\mathsf{poly}(n)}}$

O^*-notation ignores polynomial factors in the input size:

$$O^*(f(n)) \equiv O(f(n) \cdot \mathsf{poly}(n))$$
$$O^*(f(k)) \equiv O(f(k) \cdot \mathsf{poly}(n))$$

Brute-force algorithms for NP-hard problems

Theorem 1

Every problem in NP can be solved in exponential time.

Brute-force algorithms for NP-hard problems

Theorem 1

Every problem in NP can be solved in exponential time.

For a proof, see the lecture on NP-completeness.

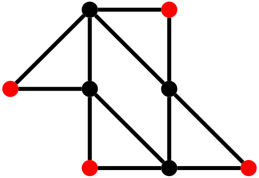
Three main categories for NP-complete problems

- Subset problems
- Permutation problems
- Partition problems

Subset Problem: INDEPENDENT SET

An independent set in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that the vertices in S are pairwise non-adjacent in G.

INDEPENDENT SET
Input: Graph G, integer k
Question: Does G have an independent set of size k?

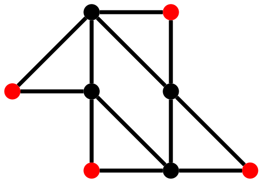


Brute-force:

Subset Problem: Independent Set

An independent set in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that the vertices in S are pairwise non-adjacent in G.

Independent Set	
Input:	Graph G, integer k
Question:	Does G have an independent set of size k?

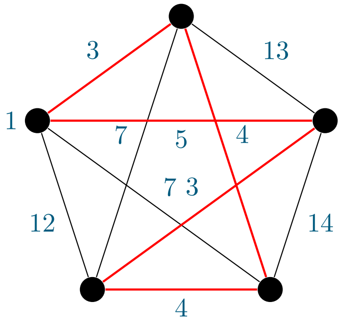


Brute-force: $O^*(2^n)$, where $n = |V(G)|$

Permutation Problem: Traveling SalesPerson

Traveling SalesPerson (TSP)

Input:	a set of n cities, the distance $d(i,j) \in \mathbb{N}$ between every two cities i and j, integer k
Question:	Is there a permutation of the cities (a tour) such that the total distance when traveling from city to city in the specified order, and returning back to the origin, is at most k?



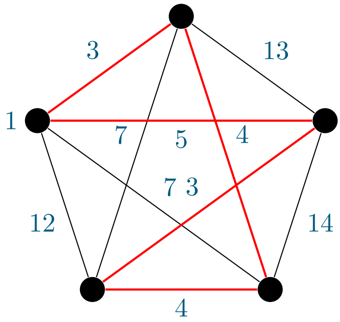
Brute-force:

Permutation Problem: TRAVELING SALESPERSON

TRAVELING SALESPERSON (TSP)

Input: a set of n cities, the distance $d(i,j) \in \mathbb{N}$ between every two cities i and j, integer k

Question: Is there a permutation of the cities (a tour) such that the total distance when traveling from city to city in the specified order, and returning back to the origin, is at most k?

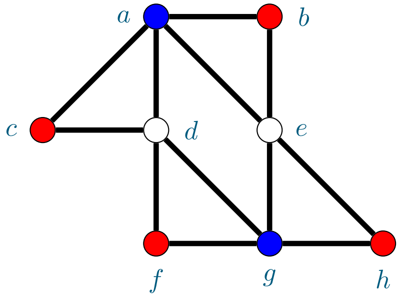


Brute-force: $O^*(n!) \subseteq 2^{O(n \log n)}$

Partition Problem: Coloring

A k-coloring of a graph $G = (V, E)$ is a function $f : V \to \{1, 2, \ldots, k\}$ assigning colors to V such that no two adjacent vertices receive the same color.

Coloring
Input: Graph G, integer k
Question: Does G have a k-coloring?

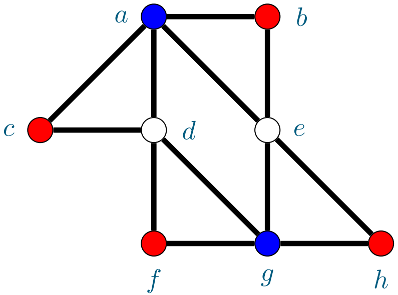


Brute-force:

Partition Problem: Coloring

A k-coloring of a graph $G = (V, E)$ is a function $f : V \to \{1, 2, \ldots, k\}$ assigning colors to V such that no two adjacent vertices receive the same color.

Coloring	
Input:	Graph G, integer k
Question:	Does G have a k-coloring?



Brute-force: $O^*(k^n)$, where $n = |V(G)|$

Exponential Time Algorithms

- natural question in Algorithms:
  design faster (worst-case analysis) algorithms for problems
- might lead to practical algorithms
  - for small instances
    - you don't want to design software where your client/boss can find with better solutions *by hand* than your software
  - subroutines for
    - (sub)exponential time approximation algorithms
    - randomized algorithms with expected polynomial run time

Solve an NP-hard problem

- exhaustive search
  - trivial method
  - try all candidate solutions (certificates) for a ground set on n elements
  - running times for problems in NP
    - Subset Problems: $O^*(2^n)$
    - Permutation Problems: $O^*(n!)$
    - Partition Problems: $O^*(c^{n \log n})$
- faster exact algorithms
  - for some problems, it is possible to obtain provably faster algorithms
  - running times $O(1.0836^n), O(1.4689^n), O(1.9977^n)$

Exponential Time Algorithms in Practice

- How large are the instances one can solve in practice?

Available time nb. of operations	1 s 2^{38}	1 min $\sim 2^{44}$	1 hour $\sim 2^{50}$	3 days $\sim 2^{56}$	6 months $\sim 2^{62}$
n^5	194	446	1,024	2,352	5,404
n^{10}	14	21	32	49	74
1.05^n	540	625	711	796	881
1.1^n	276	320	364	407	451
1.5^n	65	75	85	96	106
2^n	38	44	50	56	62
5^n	16	19	22	24	27
$n!$	14	16	17	19	20

Note: Intel Core i7-8086K executes $\sim 2^{38}$ instructions per second at 5 GHz.

"For every polynomial-time algorithm you have, there is an exponential algorithm that I would rather run."

– Alan Perlis (1922-1990, programming languages, 1st recipient of Turing Award)

Hardware vs. Algorithms

- Suppose a 2^n algorithm enables us to solve instances up to size x
- Faster processors
  - processor speed doubles after 18–24 months (Moore's law)
  - can solve instances up to size $x + 1$
- Faster algorithm
  - design an $O^*(2^{n/2}) \subseteq O(1.4143^n)$ time algorithm
  - can solve instances up to size $2 \cdot x$

Outline

A story

A computer scientist meets a biologist . . .

Eliminating conflicts from experiments

$n = 1000$ experiments,
$k = 20$ experiments failed

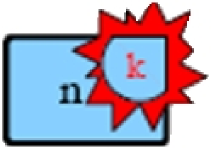
Theoretical	Running Time Number of Instructions	Real
2^n	$1.07 \cdot 10^{301}$	$4.941 \cdot 10^{282}$ years
n^k	10^{60}	$4.611 \cdot 10^{41}$ years
$2^k \cdot n$	$1.05 \cdot 10^9$	0.01526 seconds

Notes

- We assume that 2^{36} instructions are carried out per second.
- The Big Bang happened roughly $13.5 \cdot 10^9$ years ago.

Goal of Parameterized Complexity

Confine the combinatorial explosion to a parameter k.



For which problem–parameter combinations can we find algorithms with running times of the form

$$f(k) \cdot n^{O(1)},$$

where the f is a computable function independent of the input size n?

Examples of Parameters

A Parameterized Problem

Input:	an instance of the problem
Parameter:	a parameter k
Question:	a YES/NO question about the instance and the parameter

- A parameter can be
  - input size (trivial parameterization)
  - solution size
  - related to the structure of the input (maximum degree, treewidth, branchwidth, genus, ...)
  - etc.

Main Complexity Classes

P: class of problems that can be solved in time $n^{O(1)}$
FPT: class of problems that can be solved in time $f(k) \cdot n^{O(1)}$
W[·]: parameterized intractability classes
XP: class of problems that can be solved in time $f(k) \cdot n^{g(k)}$

$$\mathsf{P} \subseteq \mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \cdots \subseteq \mathsf{W}[P] \subseteq \mathsf{XP}$$

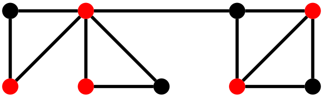
Known: If FPT $=$ W[1], then the Exponential Time Hypothesis fails, i.e. 3-SAT can be solved in time $2^{o(n)}$.

Outline

Vertex Cover

Vertex Cover (VC)

Input:	A graph $G = (V, E)$ on n vertices, an integer k
Parameter:	k
Question:	Is there a set of vertices $C \subseteq V$ of size at most k such that every edge has at least one endpoint in C?



Outline

Brute Force Algorithms

- $2^n \cdot n^{O(1)}$ not FPT
- $n^k \cdot n^{O(1)}$ not FPT

An FPT Algorithm

Algorithm vc1(G, k);

```
if E = ∅ then                                  // all edges are covered
    return Yes
else if k ≤ 0 then                             // we cannot select any vertex
    return No
else
    Select an edge uv ∈ E;
    return vc1(G − u, k − 1) ∨ vc1(G − v, k − 1)
```

Running Time Analysis

- Let us look at an arbitrary execution of the algorithm.
- Recursive calls form a search tree T
  - with depth $\leq k$
  - where each node has ≤ 2 children
- $\Rightarrow T$ has $\leq 2^k$ leaves and $\leq 2^k - 1$ internal nodes
- at each node the algorithm spends time $n^{O(1)}$
- The running time is $O^*(2^k)$

A faster FPT Algorithm

A faster FPT Algorithm

Algorithm vc2(G, k);

if $E = \emptyset$ **then** `// all edges are covered`
  return Yes
else if $k \leq 0$ **then** `// we used too many vertices`
  return No
else if $\Delta(G) \leq 2$ **then** `// ` G `has maximum degree` ≤ 2
  Solve the problem in polynomial time;
else
  Select a vertex v of maximum degree;
  return vc2$(G - v, k - 1) \ \vee\ $ vc2$(G - N[v], k - d(v))$

Running time analysis of vc2

- Number of leaves of the search tree:

$$\begin{aligned} T(k) &\leq T(k-1) + T(k-3) \\ x^k &\leq x^{k-1} + x^{k-3} \\ x^3 - x^2 - 1 &\leq 0 \end{aligned}$$

- The equation $x^3 - x^2 - 1 = 0$ has a unique positive real solution: $x \approx 1.4655\ldots$
- Running time: $1.4656^k \cdot n^{O(1)}$

Outline

1. Algorithms for NP-hard problems
2. Exponential Time Algorithms
3. Parameterized Complexity
   - FPT Algorithm for Vertex Cover
   - Algorithms for Vertex Cover
4. Further Reading

Further Reading

- Exponential-time algorithms
  - Chapter 1, *Introduction*, in (Fomin and Kratsch, 2010).
  - Survey on exponential-time algorithms (Woeginger, 2001).
  - Chapter 1, *Introduction*, in (Gaspers, 2010).
- Parameterized Complexity
  - Chapter 1, *Introduction*, in (Cygan et al., 2015)
  - Chapter 2, *The Basic Definitions*, in (Downey and Fellows, 2013)
  - Chapter I, *Foundations*, in (Niedermeier, 2006)
  - *Preface* in (Flum and Grohe, 2006)

References I

Scott Aaronson (2005). "Guest Column: NP-complete problems and physical reality". In: *SIGACT News* 36.1, pp. 30–52. DOI: 10.1145/1052796.1052804.

Jianer Chen, Iyad A. Kanj, and Ge Xia (2010). "Improved upper bounds for vertex cover". In: *Theoretical Computer Science* 411.40-42, pp. 3736–3756. DOI: 10.1016/j.tcs.2010.06.026.

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Jörg Flum and Martin Grohe (2006). *Parameterized Complexity Theory*. Springer. DOI: 10.1007/3-540-29953-X.

Fedor V. Fomin and Dieter Kratsch (2010). *Exact Exponential Algorithms*. Springer. DOI: 10.1007/978-3-642-16533-7.

Serge Gaspers (2010). *Exponential Time Algorithms: Structures, Measures, and Bounds*. VDM Verlag Dr. Mueller.

References II

Martin Charles Golumbic (2004). *Algorithmic Graph Theory and Perfect Graphs*. Elsevier.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Silvia Richter, Malte Helmert, and Charles Gretton (2007). "A Stochastic Local Search Approach to Vertex Cover". In: *Proceedings of the 30th Annual German Conference on Artificial Intelligence (KI 2007)*. Vol. 4667. Lecture Notes in Computer Science. Springer, pp. 412–426. DOI: 10.1007/978-3-540-74565-5_31.

Gerhard J. Woeginger (2001). "Exact Algorithms for NP-Hard Problems: A Survey". In: *Combinatorial Optimization*, pp. 185–208. DOI: 10.1007/3-540-36478-1_17.

Mingyu Xiao and Hiroshi Nagamochi (2017). "Exact algorithms for maximum independent set". In: *Information and Computation* 255, pp. 126–146. DOI: 10.1016/j.ic.2017.06.001.

NP-completeness

Serge Gaspers

UNSW

Outline

Outline

Polynomial time

Polynomial-time algorithm

Polynomial-time algorithm:
There exists a constant $c \in \mathbb{N}$ such that the algorithm has (worst-case) running-time $O(n^c)$, where n is the size of the input.

Polynomial time

Polynomial-time algorithm

Polynomial-time algorithm:
There exists a constant $c \in \mathbb{N}$ such that the algorithm has (worst-case) running-time $O(n^c)$, where n is the size of the input.

Example

Polynomial: n; $n^2 \log_2 n$; n^3; n^{20}
Super-polynomial: $n^{\log_2 n}$; $2^{\sqrt{n}}$; 1.001^n; 2^n; $n!$

Tractable problems

Central Question

Which computational problems have polynomial-time algorithms?

Million-dollar question

Intriguing class of problems: NP-complete problems.

NP-complete problems

It is unknown whether NP-complete problems have polynomial-time algorithms.

- A polynomial-time algorithm for one NP-complete problem would imply polynomial-time algorithms for all problems in NP.

Gerhard Woeginger's P vs NP page:
http://www.win.tue.nl/~gwoegi/P-versus-NP.htm

Polynomial vs. NP-complete

Polynomial

- Shortest Path: Given a graph G, two vertices a and b of G, and an integer k, does G have a simple a–b-path of length at most k?
- Euler Tour: Given a graph G, does G have a cycle that traverses each edge of G exactly once?
- 2-CNF SAT: Given a propositional formula F in 2-CNF, is F satisfiable?
  A k-CNF formula is a conjunction (AND) of clauses, and each clause is a disjunction (OR) of at most k literals, which are negated or unnegated Boolean variables.

NP-complete

- Longest Path: Given a graph G and an integer k, does G have a simple path of length at least k?
- Hamiltonian Cycle: Given a graph G, does G have a simple cycle that visits each vertex of G?
- 3-CNF SAT: Given a propositional formula F in 3-CNF, is F satisfiable?
  Example: $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Overview

What's next?

- Formally define P, NP, and NP-complete (NPC)
- (New) skill: show that a problem is NP-complete

Outline

Decision problems and Encodings

<Name of Decision Problem>
Input: <What constitutes an instance>
Question: <Yes/No question>

Decision problems and Encodings

<Name of Decision Problem>
Input: <What constitutes an instance>
Question: <Yes/No question>

We want to know which decision problems can be solved in polynomial time – polynomial in the size of the input n.

- Assume a “reasonable” encoding of the input
- Many encodings are polynomial-time equivalent; i.e., one encoding can be computed from another in polynomial time.
- Important exception: unary versus binary encoding of integers.
  - An integer x takes $\lceil \log_2 x \rceil$ bits in binary and $x = 2^{\log_2 x}$ bits in unary.

Formal-language framework

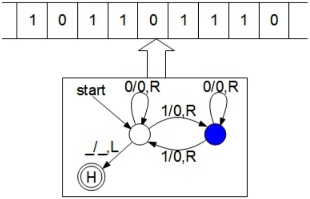
We can view decision problems as languages.

- Alphabet Σ: finite set of symbols. W.l.o.g., $\Sigma = \{0, 1\}$
- Language L over Σ: set of strings made with symbols from Σ: $L \subseteq \Sigma^*$
- Fix an encoding of instances of a decision problem Π into Σ
- Define the language $L_\Pi \subseteq \Sigma^*$ such that

$$x \in L_\Pi \Leftrightarrow x \text{ is a Yes-instance for } \Pi$$

Non-deterministic Turing Machine (NTM)

- input word $x \in \Sigma^*$ placed on an infinite tape (memory)
- read-write head initially placed on the first symbol of x
- computation step: if the machine is in state s and reads a, it can move into state s', writing b, and moving the head into direction $D \in \{L, R\}$ if $((s, a), (s', b, D)) \in \delta$.



- Q: finite, non-empty set of states
- Γ: finite, non-empty set of tape symbols
- $\_ \in \Gamma$: blank symbol (the only symbol allowed to occur on the tape infinitely often)
- $\Sigma \subseteq \Gamma \setminus \{b\}$: set of input symbols
- $q_0 \in Q$: start state
- $A \subseteq Q$: set of accepting (final) states
- $\delta \subseteq (Q \setminus A \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$: transition relation, where L stands for a move to the left and R for a move to the right.

Accepted Language

Definition 1

A NTM accepts a word $x \in \Sigma^*$ if there exists a sequence of computation steps starting in the start state and ending in an accept state.

Definition 2

The language accepted by an NTM is the set of words it accepts.

Acceptance in polynomial time

Definition 3

A language L is accepted in polynomial time by an NTM M if

- L is accepted by M, and
- there is a constant k such that for any word $x \in L$, the NTM M accepts x in $O(|x|^k)$ computation steps.

Deterministic Turing Machine

Definition 4

A Deterministic Turing Machine (DTM) is a Non-deterministic Turing Machine where the transition relation contains at most one tuple $((s,a),(\cdot,\cdot,\cdot))$ for each $s \in Q \setminus A$ and $a \in \Gamma$.

The transition relation δ can be viewed as a function
$\delta : Q \setminus A \times \Gamma \to Q \times \Gamma \times \{L, R\}$.
$\Rightarrow$ For a given input word $x \in \Sigma^*$, there is exactly one sequence of computation steps starting in the start state.

DTM equivalents

Many computational models are polynomial-time equivalent to DTMs:

- Random Access Machine (RAM, used for algorithms in the textbook)
- variants of Turing machines (multiple tapes, infinite only in one direction, ...)
- ...

P and NP

Definition 5 (P)

$\mathsf{P} = \{L \subseteq \Sigma^* : \text{ there is a DTM accepting } L \text{ in polynomial time}\}$

Definition 6 (NP)

$\mathsf{NP} = \{L \subseteq \Sigma^* : \text{ there is a NTM accepting } L \text{ in polynomial time}\}$

Definition 7 (coNP)

$\mathsf{coNP} = \{L \subseteq \Sigma^* : \Sigma^* \setminus L \in \mathsf{NP}\}$

coP?

Theorem 8

If $L \in \mathsf{P}$, then there is a polynomial-time DTM that halts in an accepting state on every word in L and it halts in a non-accepting state on every word not in L.

coP?

Theorem 8

If $L \in \mathsf{P}$, then there is a polynomial-time DTM that halts in an accepting state on every word in L and it halts in a non-accepting state on every word not in L.

Proof sketch.

Suppose $L \in \mathsf{P}$. By the definition of P, there is a DTM M that accepts L in polynomial time.
Idea: design a DTM M' that simulates M for $c \cdot n^k$ steps, where $c \cdot n^k$ is the running time of M and transitions to a non-accepting state if M does not halt in an accepting state.
(Note that this proof is nonconstructive: we might not know the running time of M.) □

NP and certificates

Non-deterministic choices

A NTM for an NP-language L makes a polynomial number of non-deterministic choices on input $x \in L$.
We can encode these non-deterministic choices into a certificate c, which is a polynomial-length word.
Now, there exists a DTM, which, given x and c, verifies that $x \in L$ in polynomial time.

Thus, $L \in$ NP iff there is a DTM V and for each $x \in L$ there exists a polynomial-length certificate c such that $V(x, c) = 1$, but $V(y, \cdot) = 0$ for each $y \notin L$.

CNF-SAT is in NP

- A CNF formula is a propositional formula in conjunctive normal form: a conjunction (AND) of clauses; each clause is a disjunction (OR) of literals; each literal is a negated or unnegated Boolean variable.
- An assignment $\alpha : \mathrm{var}(F) \to \{0,1\}$ satisfies a clause C if it sets a literal of C to true, and it satisfies F if it satisfies all clauses in F.

CNF-SAT	
Input:	CNF formula F
Question:	Does F have a satisfying assignment?

Example: $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Lemma 9

CNF-SAT $\in$ NP.

CNF-SAT is in NP

- A CNF formula is a propositional formula in conjunctive normal form: a conjunction (AND) of clauses; each clause is a disjunction (OR) of literals; each literal is a negated or unnegated Boolean variable.
- An assignment $\alpha : \text{var}(F) \to \{0,1\}$ satisfies a clause C if it sets a literal of C to true, and it satisfies F if it satisfies all clauses in F.

CNF-SAT	
Input:	CNF formula F
Question:	Does F have a satisfying assignment?

Example: $(x \vee \neg y \vee z) \wedge (\neg x \vee z) \wedge (\neg y \vee \neg z)$.

Lemma 9

CNF-SAT $\in$ NP.

Proof.

Certificate: assignment α to the variables.
Given a certificate, it can be checked in polynomial time whether all clauses are satisfied. $\square$

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Proof.

Let Π be an arbitrary problem in NP. [Use certificate-based definition of NP]
We know that $\exists$ a polynomial p and a polynomial-time verification algorithm V such that:

- for every $x \in \Pi$ (i.e., every YES-instance for Π) $\exists$ string $c \in \{0,1\}^*$, $|c| \le p(|x|)$, such that $V(x,c) = 1$, and
- for every $x \notin \Pi$ (i.e., every NO-instance for Π) and every string $c \in \{0,1\}^*$, $V(x,c) = 0$.

Brute-force algorithms for problems in NP

Theorem 10

Every problem in NP can be solved in exponential time.

Proof.

Let Π be an arbitrary problem in NP. [Use certificate-based definition of NP]
We know that $\exists$ a polynomial p and a polynomial-time verification algorithm V such that:

- for every $x \in \Pi$ (i.e., every YES-instance for Π) $\exists$ string $c \in \{0,1\}^*$, $|c| \leq p(|x|)$, such that $V(x,c) = 1$, and
- for every $x \notin \Pi$ (i.e., every NO-instance for Π) and every string $c \in \{0,1\}^*$, $V(x,c) = 0$.

Now, we can prove there exists an exponential-time algorithm for Π with input x:

- For each string $c \in \{0,1\}^*$ with $|c| \leq p(|x|)$, evaluate $V(x,c)$ and return YES if $V(x,c) = 1$.
- Return NO.

Running time: $2^{p(|x|)} \cdot n^{O(1)} \subseteq 2^{O(2 \cdot p(|x|))} = 2^{O(p(|x|))}$, but non-constructive. $\square$

Outline

Polynomial-time reduction

Definition 11

A language L_1 is polynomial-time reducible to a language L_2, written $L_1 \leq_P L_2$, if there exists a polynomial-time computable function $f : \Sigma^* \to \Sigma^*$ such that for all $x \in \Sigma^*$,

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

A polynomial time algorithm computing f is a reduction algorithm.

New polynomial-time algorithms via reductions

Lemma 12

If $L_1, L_2 \in \Sigma^*$ are languages such that $L_1 \leq_P L_2$, then $L_2 \in \mathsf{P}$ implies $L_1 \in \mathsf{P}$.

NP-completeness

Definition 13 (NP-hard)

A language $L \subseteq \Sigma^*$ is NP-hard if

$$L' \leq_P L \text{ for every } L' \in \text{NP}.$$

Definition 14 (NP-complete)

A language $L \subseteq \Sigma^*$ is NP-complete (in NPC) if

1. $L \in$ NP, and
2. L is NP-hard.

A first NP-complete problem

Theorem 15

CNF-SAT is NP-complete.

Proved by encoding NTMs into SAT (Cook, 1971; Levin, 1973) and then CNF-SAT (Karp, 1972).

Proving NP-completeness

Lemma 16

If L *is a language such that* $L' \leq_P L$ *for some* $L' \in$ NPC, *then* L *is* NP*-hard. If, in addition,* $L \in$ NP, *then* $L \in$ NPC.

Proving NP-completeness

Lemma 16

If L is a language such that $L' \leq_P L$ for some $L' \in$ NPC, then L is NP-hard. If, in addition, $L \in$ NP, then $L \in$ NPC.

Proof.

For all $L'' \in$ NP, we have $L'' \leq_P L' \leq_P L$.
By transitivity, we have $L'' \leq_P L$.
Thus, L is NP-hard. □

Proving NP-completeness (2)

Method to prove that a language L is NP-complete:

1. Prove $L \in$ NP
2. Prove L is NP-hard.
   - Select a known NP-complete language L'.
   - Describe an algorithm that computes a function f mapping every instance $x \in \Sigma^*$ of L' to an instance $f(x)$ of L.
   - Prove that $x \in L' \Leftrightarrow f(x) \in L$ for all $x \in \Sigma^*$.
   - Prove that the algorithm computing f runs in polynomial time.

Outline

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$. Create $k-3$ new variables $y_1, \dots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \dots, (\neg y_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

3-CNF SAT is NP-hard

Theorem 17

3-CNF SAT is NP-complete.

Proof.

3-CNF SAT is in NP, since it is a special case of CNF-SAT.
To show that 3-CNF SAT is NP-hard, we give a polynomial reduction from CNF-SAT.
Let F be a CNF formula. The reduction algorithm constructs a 3-CNF formula F' as follows. For each clause C in F:

- If C has at most 3 literals, then copy C into F'.
- Otherwise, denote $C = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k)$. Create $k-3$ new variables $y_1, \dots, y_{k-3}$, and add the clauses $(\ell_1 \vee \ell_2 \vee y_1), (\neg y_1 \vee \ell_3 \vee y_2), (\neg y_2 \vee \ell_4 \vee y_3), \dots, (\neg y_{k-3} \vee \ell_{k-1} \vee \ell_k)$.

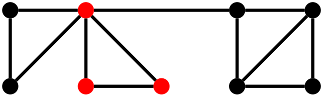
Show that F is satisfiable $\Leftrightarrow$ F' is satisfiable.
Show that F' can be computed in polynomial time (trivial; use a RAM). $\square$

Clique

A clique in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every two vertices of S are adjacent in G.

CLIQUE

Input:	Graph G, integer k
Question:	Does G have a clique of size k?



Theorem 18

CLIQUE *is* NP*-complete.*

Clique (2)

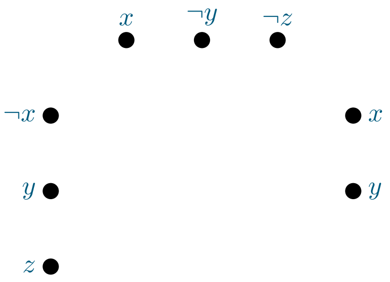
- CLIQUE is in NP

Clique (2)

- $\textsc{Clique}$ is in NP
- Let $F = C_1 \wedge C_2 \wedge \ldots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable

$$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$$

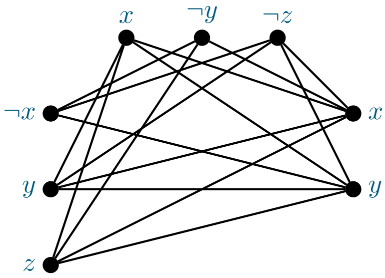
Clique (2)



$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- CLIQUE is in NP
- Let $F = C_1 \wedge C_2 \wedge \ldots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable
- For each clause $C_r = (\ell_1^r \vee \cdots \vee \ell_w^r)$, $1 \leq r \leq k$, create w new vertices $v_1^r, \ldots, v_w^r$

Clique (2)



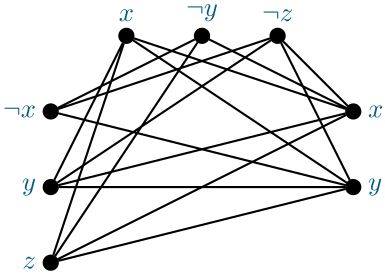
$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- Clique is in NP
- Let $F = C_1 \wedge C_2 \wedge \dots C_k$ be a 3-CNF formula
- Construct a graph G that has a clique of size k iff F is satisfiable
- For each clause $C_r = (\ell_1^r \vee \dots \vee \ell_w^r)$, $1 \le r \le k$, create w new vertices $v_1^r, \dots, v_w^r$
- Add an edge between v_i^r and v_j^s if

$$r \neq s \quad \text{and}$$
$$\ell_i^r \neq \neg \ell_j^s \quad \text{where } \neg\neg x = x.$$

- Check correctness and polynomial running time

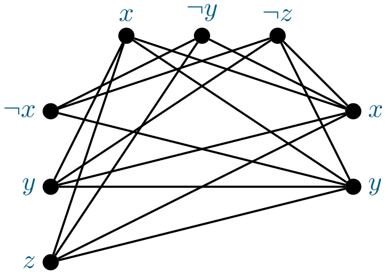
Clique (2)



$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- Correctness: F has a satisfying assignment iff G has a clique of size k.

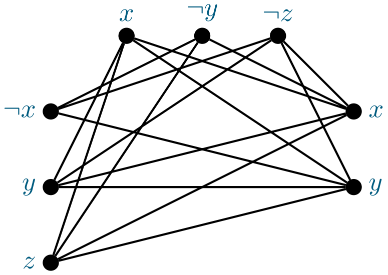
Clique (2)



$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- Correctness: F has a satisfying assignment iff G has a clique of size k.
- $(\Rightarrow)$: Let α be a sat. assignment for F. For each clause C_r, choose a literal ℓ_i^r with $\alpha(\ell_i^r) = 1$, and denote by s^r the corresponding vertex in G. Now, $\{s^r : 1 \leq r \leq k\}$ is a clique of size k in G since $\alpha(x) \neq \alpha(\neg x)$.

Clique (2)



$(\neg x \vee y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (x \vee y)$

- Correctness: F has a satisfying assignment iff G has a clique of size k.
- $(\Rightarrow)$: Let α be a sat. assignment for F. For each clause C_r, choose a literal ℓ_i^r with $\alpha(\ell_i^r) = 1$, and denote by s^r the corresponding vertex in G. Now, $\{s^r : 1 \leq r \leq k\}$ is a clique of size k in G since $\alpha(x) \neq \alpha(\neg x)$.
- $(\Leftarrow)$: Let S be a clique of size k in G. Then, S contains exactly one vertex $s_r \in \{v_1^r, \ldots, v_w^r\}$ for each $r \in \{1, \ldots, k\}$. Denote by l^r the corresponding literal. Now, for any r, r', it is not the case that $l_r = \neg l_{r'}$. Therefore, there is an assignment α to var(F) such that $\alpha(l_r) = 1$ for each $r \in \{1, \ldots, k\}$ and α satisfies F.

Vertex Cover

A vertex cover in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every edge of G has an endpoint in S.

VERTEX COVER	
Input:	Graph G, integer k
Question:	Does G have a vertex cover of size k?

Theorem 19

VERTEX COVER *is* NP*-complete.*

The proof is left as an exercise.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

HAMILTONIAN CYCLE
Input: Graph G
Question: Does G have a Hamiltonian Cycle?

Theorem 20

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

Hamiltonian Cycle	
Input:	Graph G
Question:	Does G have a Hamiltonian Cycle?

Theorem 20

Hamiltonian Cycle *is* NP*-complete.*

Proof sketch.

- Hamiltonian Cycle is in NP: the certificate is a Hamiltonian Cycle of G.

Hamiltonian Cycle

A Hamiltonian Cycle in a graph $G = (V, E)$ is a cycle visiting each vertex exactly once.
(Alternatively, a permutation of V such that every two consecutive vertices are adjacent and the first and last vertex in the permutation are adjacent.)

HAMILTONIAN CYCLE	
Input:	Graph G
Question:	Does G have a Hamiltonian Cycle?

Theorem 20

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch.

- HAMILTONIAN CYCLE is in NP: the certificate is a Hamiltonian Cycle of G.
- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE

...

$\square$

Hamiltonian Cycle (2)

Theorem 21

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE

Hamiltonian Cycle (2)

Theorem 21

Hamiltonian Cycle is NP-complete.

Proof sketch (continued).

- Let us show: Vertex Cover $\leq_P$ Hamiltonian Cycle
- Let $(G = (V, E), k)$ be an instance for Vertex Cover (VC).
- We will construct an equivalent instance G' for Hamiltonian Cycle (HC).

Hamiltonian Cycle (2)

Theorem 21

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Let us show: VERTEX COVER $\leq_P$ HAMILTONIAN CYCLE
- Let $(G = (V, E), k)$ be an instance for VERTEX COVER (VC).
- We will construct an equivalent instance G' for HAMILTONIAN CYCLE (HC).
- Intuition: Non-deterministic choices
  - for VC: which vertices to select in the vertex cover
  - for HC: which route the cycle takes

...

□

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)
- Each edge of G will be represented by a gadget (subgraph) of G'
- s.t. the set of edges covered by a vertex x in G corresponds to a partial cycle going through all gadgets of G' representing these edges.

Hamiltonian Cycle (3)

Theorem 22

HAMILTONIAN CYCLE *is* NP*-complete.*

Proof sketch (continued).

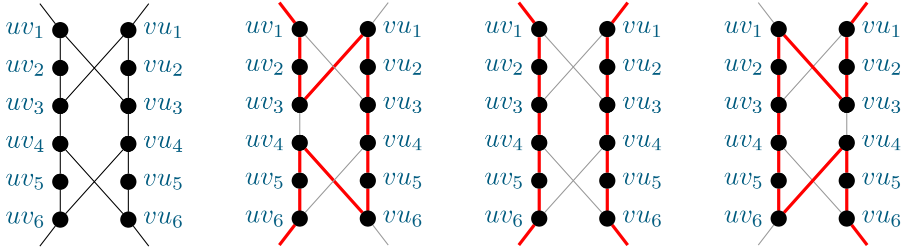
- Add k vertices $s_1, \ldots, s_k$ to G' (*selector vertices*)
- Each edge of G will be represented by a gadget (subgraph) of G'
- s.t. the set of edges covered by a vertex x in G corresponds to a partial cycle going through all gadgets of G' representing these edges.
- Attention: we need to allow for an edge to be covered by both endpoints

...

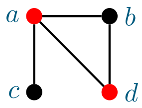
□

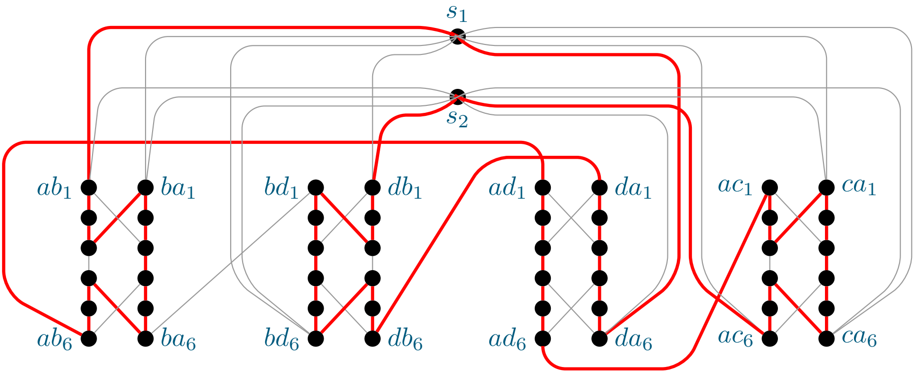
Gadget representing the edge $\{u, v\} \in E$

Its states: 'covered by u', 'covered by u and v', 'covered by v'



Hamiltonian Cycle (5)





Outline

Further Reading

- Chapter 34, **NP-Completeness**, in (Cormen et al., 2009)
- Garey and Johnson's influential reference book (Garey and Johnson, 1979)

References I

Stephen A. Cook (1971). "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971)*, pp. 151–158.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein (2009). *Introduction to Algorithms*. 3rd ed. The MIT Press.

Michael R. Garey and David S. Johnson (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co.

Richard M. Karp (1972). "Reducibility among combinatorial problems". In: *Complexity of computer computations (Proc. Sympos., IBM Thomas J. Watson Res. Center, Yorktown Heights, N.Y., 1972)*. New York: Plenum, pp. 85–103.

Leonid Levin (1973). "Universal sequential search problems". In: *Problems of Information Transmission* 9.3, pp. 265–266.

Kernelization

Serge Gaspers

UNSW

Outline

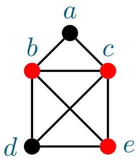
Outline

1. Vertex Cover
   - Simplification rules
   - Preprocessing algorithm
2. Kernelization algorithms
3. Kernel for HAMILTONIAN CYCLE
4. Kernel for EDGE CLIQUE COVER
5. Kernels and Fixed-parameter tractability
6. Further Reading

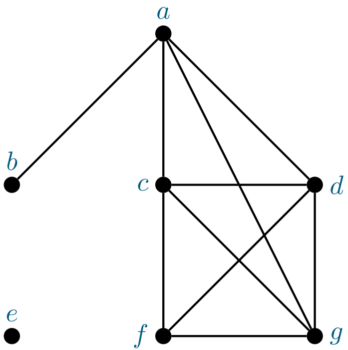
Vertex cover

A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that for each edge $\{u, v\} \in E$, we have $u \in S$ or $v \in S$.

VERTEX COVER	
Input:	A graph $G = (V, E)$ and an integer k
Parameter:	k
Question:	Does G have a vertex cover of size at most k?



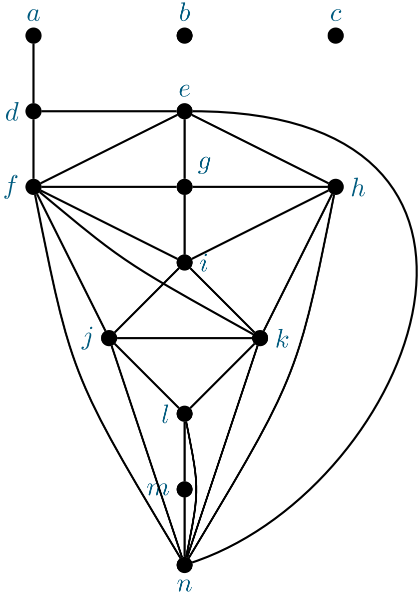
Exercise 1



$k = 4$

Is this a YES-instance for VERTEX COVER?
(Is there $S \subseteq V$ with $|S| \leq 4$, such that $\forall\ uv \in E$, $u \in S$ or $v \in S$?)

Exercise 2



$k = 7$

Outline

Simplification rules for Vertex Cover

(Degree-0)

If $\exists v \in V$ such that $d_G(v) = 0$, then set $G \leftarrow G - v$.

Simplification rules for VERTEX COVER

(Degree-0)

If $\exists v \in V$ such that $d_G(v) = 0$, then set $G \leftarrow G - v$.

Proving correctness. A simplification rule is sound if for every instance, it produces an equivalent instance. Two instances I, I' are equivalent if they are both YES-instances or they are both NO-instances.

Lemma 1

(Degree-0) is sound.

Simplification rules for Vertex Cover

(Degree-0)

If $\exists v \in V$ such that $d_G(v) = 0$, then set $G \leftarrow G - v$.

Proving correctness. A simplification rule is sound if for every instance, it produces an equivalent instance. Two instances I, I' are equivalent if they are both Yes-instances or they are both No-instances.

Lemma 1

(Degree-0) is sound.

Proof.

First, suppose $(G - v, k)$ is a Yes-instance. Let S be a vertex cover for $G - v$ of size at most k. Then, S is also a vertex cover for G since no edge of G is incident to v. Thus, (G, k) is a Yes-instance.

Now, suppose $(G - v, k)$ is a No-instance. For the sake of contradiction, assume (G, k) is a Yes-instance. Let S be a vertex cover for G of size at most k. But then, $S \setminus \{v\}$ is a vertex cover of size at most k for $G - v$; a contradiction. $\square$

Simplification rules for Vertex Cover

(Degree-1)

If $\exists v \in V$ such that $d_G(v) = 1$, then set $G \leftarrow G - N_G[v]$ and $k \leftarrow k - 1$.

Simplification rules for VERTEX COVER

(Degree-1)

If $\exists v \in V$ such that $d_G(v) = 1$, then set $G \leftarrow G - N_G[v]$ and $k \leftarrow k - 1$.

Lemma 1

(Degree-1) is sound.

Proof.

Let u be the neighbor of v in G. Thus, $N_G[v] = \{u, v\}$.
If S is a vertex cover of G of size at most k, then $S \setminus \{u, v\}$ is a vertex cover of $G - N_G[v]$ of size at most $k - 1$, because $u \in S$ or $v \in S$.
If S' is a vertex cover of $G - N_G[v]$ of size at most $k - 1$, then $S' \cup \{u\}$ is a vertex cover of G of size at most k, since all edges that are in G but not in $G - N_G[v]$ are incident to u. $\square$

Simplification rules for Vertex Cover

(Large Degree)

If $\exists v \in V$ such that $d_G(v) > k$, then set $G \leftarrow G - v$ and $k \leftarrow k - 1$.

Simplification rules for Vertex Cover

(Large Degree)

If $\exists v \in V$ such that $d_G(v) > k$, then set $G \leftarrow G - v$ and $k \leftarrow k - 1$.

Lemma 1

(Large Degree) is sound.

Proof.

Let S be a vertex cover of G of size at most k. If $v \notin S$, then $N_G(v) \subseteq S$, contradicting that $|S| \leq k$. □

Simplification rules for VERTEX COVER

(Number of Edges)

If $d_G(v) \leq k$ for each $v \in V$ and $|E| > k^2$ then return NO

Simplification rules for Vertex Cover

(Number of Edges)

If $d_G(v) \le k$ for each $v \in V$ and $|E| > k^2$ then return No

Lemma 1

(Number of Edges) is sound.

Proof.

Assume $d_G(v) \le k$ for each $v \in V$ and $|E| > k^2$.
Suppose $S \subseteq V$, $|S| \le k$, is a vertex cover of G.
We have that S covers at most k^2 edges.
However, $|E| \ge k^2 + 1$.
Thus, S is not a vertex cover of G. □

Outline

Preprocessing algorithm for Vertex Cover

VC-preprocess

Input: A graph G and an integer k.

Output: A graph G' and an integer k' such that G has a vertex cover of size at most k if and only if G' has a vertex cover of size at most k'.

$G' \leftarrow G$
$k' \leftarrow k$
repeat
  Execute simplification rules (Degree-0), (Degree-1), (Large Degree), and (Number of Edges) for (G', k')
until *no simplification rule applies*
return (G', k')

Effectiveness of preprocessing algorithms

- How effective is VC-preprocess?
- We would like to study preprocessing algorithms mathematically and quantify their effectiveness.

First try

- Say that a preprocessing algorithm for a problem Π is nice if it runs in polynomial time and for each instance for Π, it returns an instance for Π that is strictly smaller.

First try

- Say that a preprocessing algorithm for a problem Π is nice if it runs in polynomial time and for each instance for Π, it returns an instance for Π that is strictly smaller.
- $\rightarrow$ executing it a linear number of times reduces the instance to a single bit
- $\rightarrow$ such an algorithm would solve Π in polynomial time
- For NP-hard problems this is not possible unless $\mathsf{P} = \mathsf{NP}$
- We need a different measure of effectiveness

Measuring the effectiveness of preprocessing algorithms

- We will measure the effectiveness in terms of the parameter
- How large is the resulting instance in terms of the parameter?

Effectiveness of VC-preprocess

Lemma 2

For any instance (G, k) for VERTEX COVER*, VC-preprocess produces an equivalent instance (G', k') of size $O(k^2)$.*

Proof.

Since all simplification rules are sound, $(G = (V, E), k)$ and $(G' = (V', E'), k')$ are equivalent.
By (Number of Edges), $|E'| \leq (k')^2 \leq k^2$.
By (Degree-0) and (Degree-1), each vertex in V' has degree at least 2 in G'.
Since $\sum_{v \in V'} d_{G'}(v) = 2|E'| \leq 2k^2$, this implies that $|V'| \leq k^2$.
Thus, $|V'| + |E'| \subseteq O(k^2)$. □

Outline

Kernelization: definition

Definition 3

A kernelization for a parameterized problem Π is a **polynomial time** algorithm, which, for any instance I of Π with parameter k, produces an **equivalent** instance I' of Π with parameter k' such that $|I'| \leq f(k)$ and $k' \leq f(k)$ for a computable function f.
We refer to the function f as the size of the kernel.

Note: We do not formally require that $k' \leq k$, but this will be the case for many kernelizations.

VC-preprocess is a quadratic kernelization

Theorem 4

VC-preprocess is a $O(k^2)$ kernelization for VERTEX COVER.

Outline

A Hamiltonian cycle of G is a subgraph of G that is a cycle on $|V(G)|$ vertices.

vc-Hamiltonian Cycle

Input:	A graph $G = (V, E)$.
Parameter:	$k = vc(G)$, the size of a smallest vertex cover of G.
Question:	Does G have a Hamiltonian cycle?

Thought experiment: Imagine a very large instance where the parameter is tiny. How can you simplify such an instance?

Hamiltonian Cycle II

Issue: We do not actually know a vertex cover of size k.
We do not even know the value of k (it is not part of the input).

Hamiltonian Cycle III

- Obtain a vertex cover using an approximation algorithm. We will use a 2-approximation algorithm, producing a vertex cover of size $\leq 2k$ in polynomial time.
- If C is a vertex cover of size $\leq 2k$, then $I = V \setminus C$ is an independent set of size $\geq |V| - 2k$.
- No two consecutive vertices in the Hamiltonian Cycle can be in I.
- A kernel with $\leq 4k$ vertices can now be obtained with the following simplification rule.

(Too-large)

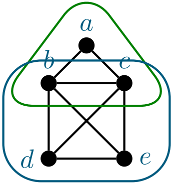
Compute a vertex cover C of size $\leq 2k$ in polynomial time.
If $2|C| < |V|$, then return No

Outline

Definition 5

An edge clique cover of a graph $G = (V, E)$ is a set of cliques in G covering all its edges.
In other words, if $\mathcal{C} \subseteq 2^V$ is an edge clique cover then each $S \in \mathcal{C}$ is a clique in G and for each $\{u, v\} \in E$ there exists an $S \in \mathcal{C}$ such that $u, v \in S$.

Example: $\{\{a, b, c\}, \{b, c, d, e\}\}$ is an edge clique cover for this graph.



Edge Clique Cover

Edge Clique Cover

Input:	A graph $G = (V, E)$ and an integer k
Parameter:	k
Question:	Does G have an edge clique cover of size at most k?

The size of an edge clique cover $\mathcal{C}$ is the number of cliques contained in $\mathcal{C}$ and is denoted $|\mathcal{C}|$.

Helpful properties

Definition 5

A clique S in a graph G is a maximal clique if there is no other clique S' in G with $S \subset S'$.

Lemma 6

A graph G has an edge clique cover $\mathcal{C}$ of size at most k if and only if G has an edge clique cover $\mathcal{C}'$ of size at most k such that each $S \in \mathcal{C}'$ is a maximal clique.

Proof sketch.

$(\Rightarrow)$: Replace each clique $S \in \mathcal{C}$ by a maximal clique S' with $S \subseteq S'$.
$(\Leftarrow)$: Trivial, since $\mathcal{C}'$ is an edge clique cover of size at most k. $\square$

Simplification rules for Edge Clique Cover

Thought experiment: Imagine a very large instance where the parameter is tiny. How can you simplify such an instance?

Simplification rules for EDGE CLIQUE COVER II

The instance could have many degree-0 vertices.

(Isolated)

If there exists a vertex $v \in V$ with $d_G(v) = 0$, then set $G \leftarrow G - v$.

Lemma 7

(Isolated) is sound.

Proof sketch.

Since no edge is incident to v, a smallest edge clique cover for $G - v$ is a smallest edge clique cover for G, and vice-versa. □

Simplification rules for Edge Clique Cover II

The instance could have many degree-0 vertices.

(Isolated)

If there exists a vertex $v \in V$ with $d_G(v) = 0$, then set $G \leftarrow G - v$.

Lemma 7

(Isolated) is sound.

Proof sketch.

Since no edge is incident to v, a smallest edge clique cover for $G - v$ is a smallest edge clique cover for G, and vice-versa. $\square$

(Isolated-Edge)

If $\exists uv \in E$ such that $d_G(u) = d_G(v) = 1$, then set $G \leftarrow G - \{u, v\}$ and $k \leftarrow k - 1$.

Simplification rules for Edge Clique Cover III

(Twins)

If $\exists u, v \in V$, $u \neq v$, such that $N_G[u] = N_G[v]$, then set $G \leftarrow G - v$.

Lemma 8

(Twins) is sound.

Simplification rules for Edge Clique Cover III

(Twins)

If $\exists u, v \in V$, $u \neq v$, such that $N_G[u] = N_G[v]$, then set $G \leftarrow G - v$.

Lemma 8

(Twins) is sound.

Proof.

We need to show that G has an edge clique cover of size at most k if and only if $G - v$ has an edge clique cover of size at most k.
$(\Rightarrow)$: If $\mathcal{C}$ is an edge clique cover of G of size at most k, then $\{S \setminus \{v\} : S \in \mathcal{C}\}$ is an edge clique cover of $G - v$ of size at most k.
$(\Leftarrow)$: Let $\mathcal{C}'$ be an edge clique cover of $G - v$ of size at most k. Partition $\mathcal{C}'$ into $\mathcal{C}'_u = \{S \in \mathcal{C}' : u \in S\}$ and $\mathcal{C}'_{\neg u} = \mathcal{C}' \setminus \mathcal{C}'_u$. Note that each set in $\mathcal{C}_u = \{S \cup \{v\} : S \in \mathcal{C}'_u\}$ is a clique in G since $N_G[u] = N_G[v]$ and that each edge incident to v is contained in at least one of these cliques. Now, $\mathcal{C}_u \cup \mathcal{C}'_{\neg u}$ is an edge clique cover of G of size at most k. □

Simplification rules for EDGE CLIQUE COVER IV

(Size-V)

If the previous simplification rules do not apply and $|V| > 2^k$, then return NO.

Lemma 9

(Size-V) is sound.

Simplification rules for Edge Clique Cover IV

(Size-V)

If the previous simplification rules do not apply and $|V| > 2^k$, then return No.

Lemma 9

(Size-V) is sound.

Proof.

For the sake of contradiction, assume neither (Isolated) nor (Twins) are applicable, $|V| > 2^k$, and G has an edge clique cover $\mathcal{C}$ of size at most k. Since $2^{\mathcal{C}}$ (the set of all subsets of $\mathcal{C}$) has size at most 2^k, and every vertex belongs to at least one clique in $\mathcal{C}$ by (Isolated), we have that there exists two vertices $u, v \in V$ such that $\{S \in \mathcal{C} : u \in S\} = \{S \in \mathcal{C} : v \in S\}$. But then, $N_G[u] = \bigcup_{S \in \mathcal{C} : u \in S} S = \bigcup_{S \in \mathcal{C} : v \in S} S = N_G[v]$, contradicting that (Twin) is not applicable. □

Kernel for Edge Clique Cover

Theorem 10 ((Gramm et al., 2008))

Edge Clique Cover *has a kernel with $O(2^k)$ vertices and $O(4^k)$ edges.*

Corollary 11

Edge Clique Cover *is* FPT.

Outline

Kernels and Fixed-parameter tractability

Theorem 12

Let Π be a decidable parameterized problem.
Π has a kernelization algorithm $\Leftrightarrow$ Π is FPT.

Kernels and Fixed-parameter tractability

Theorem 12

Let Π *be a decidable parameterized problem.*
Π *has a kernelization algorithm* $\Leftrightarrow$ Π *is* FPT*.*

Proof.

($\Rightarrow$): An FPT algorithm is obtained by first running the kernelization, and then any brute-force algorithm on the resulting instance.
($\Leftarrow$): Let A be an FPT algorithm for Π with running time $O(f(k)n^c)$.
If $f(k) < n$, then A has running time $O(n^{c+1})$. In this case, the kernelization algorithm runs A and returns a trivial YES- or NO-instance depending on the answer of A.
Otherwise, $f(k) \geq n$. In this case, the kernelization algorithm outputs the input instance. $\square$

Outline

Further Reading

- Chapter 2, *Kernelization* in (Cygan et al., 2015)
- Chapter 4, *Kernelization* in (Downey and Fellows, 2013)
- Chapter 7, *Data Reduction and Problem Kernels* in (Niedermeier, 2006)
- Chapter 9, *Kernelization and Linear Programming Techniques* in (Flum and Grohe, 2006)
- the kernelization book (Fomin et al., 2019)

References

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Jörg Flum and Martin Grohe (2006). *Parameterized Complexity Theory*. Springer. DOI: 10.1007/3-540-29953-X.

Fedor V. Fomin, Daniel Lokshtanov, Saket Saurabh, and Meirav Zehavi (2019). *Kernelization. Theory of Parameterized Preprocessing*. Cambridge University Press.

Jens Gramm, Jiong Guo, Falk Huffner, and Rolf Niedermeier (2008). "Data reduction and exact algorithms for clique cover". In: *ACM J. Exp. Algorithmics* 13. DOI: 10.1145/1412228.1412236.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Approximation Algorithms

Serge Gaspers

UNSW

Outline

Outline

Optimisation problems

Definition 1

An optimisation problem is characterised by

- a set of input instances
- a set of feasible solutions for each input instance
- a value for each feasible solution

In a maximisation problem (resp., a minimisation) problem, the goal is to find a feasible solution with maximum (resp., minimum) value.

Example: In the VERTEX COVER minimisation problem, the input is a graph G, the feasible solutions are all the vertex covers of G, and the value of a vertex cover is its size.

Approximation algorithm

Definition 2

An approximation algorithm A for an optimisation problem Π is a polynomial time algorithm that returns a feasible solution.
Denote by $A(I)$ the value of the feasible solution returned by the approximation algorithm A for an instance I and by $\mathsf{OPT}(I)$ the value of the optimum solution.
If Π is a minimisation problem, then the approximation ratio of A is r if

$$\frac{A(I)}{\mathsf{OPT}(I)} \leq r \text{ for every instance } I.$$

If Π is a maximisation problem, then the approximation ratio of A is r if

$$\frac{\mathsf{OPT}(I)}{A(I)} \leq r \text{ for every instance } I.$$

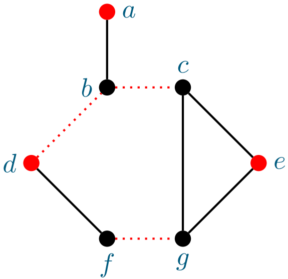
We say that A is an r-approximation algorithm if it has approximation ratio r.

Outline

Problem Definition

Multiway Cut

Input:	A connected graph $G = (V, E)$ and a set of terminals $S = \{s_1, \ldots, s_k\}$
Feasible Solution:	A multiway cut, i.e., an edge subset $X \subseteq E$ such that the graph $(V, E \setminus X)$ has no path between any two distinct terminals
Objective:	Minimize the size of the multiway cut.



Complexity

Multiway Cut is NP-complete, even when $k = 3$ (Dahlhaus et al., 1994).
Multiway Cut can be solved in polynomial time when $k = 2$ by a maximum flow algorithm.

Approximation algorithm

Algorithm Greedy-MC

- For each $i \in \{1, \ldots, k\}$, compute a smallest edge set C_i, separating s_i from the other terminals.
  (This can be done by computing a smallest cut between s_i and s_{-i} in the graph obtained from G by merging all the vertices in $S \setminus \{s_i\}$ into a new vertex s_{-i}.)
- Return $\bigcup_{i \in \{1,\ldots,k\}} C_i$.

Approximation ratio

Theorem 3 ((Dahlhaus et al., 1994))

Greedy-MC is a 2-approximation algorithm for MULTIWAY CUT.

Proof.

First, note that the algorithm runs in polynomial time.
To show that its approximation ratio is at most 2, let us compare the size of the solution it returns, $C = \bigcup_{i \in \{1,\dots,k\}} C_i$, to the size of an optimal solution, A.
The graph $(V, E \setminus A)$ has k connected components $G_1, \dots, G_k$, one for each $s_1, \dots, s_k$.
Let $A_i \subseteq A$ denote the edges with one endpoint in G_i. Observe that $A = \bigcup A_i$.
Since each edge of A is incident to two of the connected components, we have that

$$2 \cdot |A| = \sum_{i=1}^{k} |A_i| \geq \sum_{i=1}^{k} |C_i| \geq |C|$$

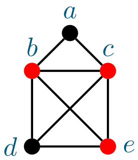
Since $|C| \leq 2 \cdot |A|$, Greedy-MC is a 2-approximation algorithm. □

Outline

Vertex cover

Recall: A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that for each edge $\{u, v\} \in E$, we have $u \in S$ or $v \in S$.

VERTEX COVER	
Input:	A graph $G = (V, E)$ and an integer k
Parameter:	k
Question:	Does G have a vertex cover of size at most k?



Outline

Preprocessing algorithm for Vertex Cover

VC-preprocess
Input: A graph G and an integer k.
Output: A graph G' and an integer k' such that G has a vertex cover of size at most k if and only if G' has a vertex cover of size at most k'.

$G' \leftarrow G$
$k' \leftarrow k$
repeat
  Execute simplification rules (Degree-0), (Degree-1), (Large Degree), and (Number of Edges) for (G', k')
until *no simplification rule applies*
return (G', k')

Claim: It is easy to add some bookkeeping to this preprocessing algorithm so that it outputs a set of $k - k'$ vertices such that any vertex cover S' for G' can be extended to a vertex cover for G by adding these $k - k'$ vertices.

Approximation algorithm for Vertex Cover

Since VC-preprocess returns an equivalent instance (G', k') of size $O(k^2)$, we have that

Corollary 4

The Vertex Cover *optimisation problem has an approximation algorithm with approximation ratio* $O(\text{OPT})$.

Proof sketch.

We start from $k = 0$ and increment k until a solution is returned

- For a given value of k, kernelize.
- If (Number of Edges) does not return No, then return a vertex cover containing all the vertices of the kernelized graph, along with the vertices determined by the bookkeeping of the kernelization procedure.

This procedure returns a vertex cover of size $O(\text{OPT}^2)$. □

Can we obtain a constant approximation ratio?

Outline

Integer Linear Program for VERTEX COVER

The VERTEX COVER problem can be written as an Integer Linear Program (ILP).
For an instance $(G = (V, E), k)$ for VERTEX COVER with $V = \{v_1, \dots, v_n\}$, create a variable x_i for each vertex v_i, $1 \le i \le n$.
Let $X = \{x_1, \dots, x_n\}$.

$$\text{ILP}_{\text{VC}}(G) = \boxed{\begin{aligned} &\text{Minimize} \sum_{i=1}^{n} x_i \\ &x_i + x_j \ge 1 \qquad \text{for each } \{v_i, v_j\} \in E \\ &x_i \in \{0,1\} \qquad \text{for each } i \in \{1, \dots, n\} \end{aligned}}$$

Then, (G, k) is a YES-instance iff the objective value of $\text{ILP}_{\text{VC}}(G)$ is at most k.

Note: Since we just reduced the NP-complete VERTEX COVER problem to ILP, we conclude that ILP is NP-hard.

LP relaxation for Vertex Cover

$$\text{LP}_{\text{VC}}(G) = \boxed{\begin{array}{rl} \text{Minimize} \displaystyle\sum_{i=1}^{n} x_i & \\ x_i + x_j \geq 1 & \text{for each } \{v_i, v_j\} \in E \\ x_i \geq 0 & \text{for each } i \in \{1, \ldots, n\} \end{array}}$$

Note: the value of an optimal solution for the Linear Program $\text{LP}_{\text{VC}}(G)$ is at most the value of an optimal solution for $\text{ILP}_{\text{VC}}(G)$

Note 2: Linear Programs (LP) can be solved in polynomial time (Cohen, Lee, and Song, 2019).

Properties of LP optimal solution

- Let $\alpha : X \to \mathbb{R}_{\geq 0}$ be an optimal solution for $\text{LP}_{\text{VC}}(G)$. Let

$$\begin{aligned} V_- &= \{v_i : \alpha(x_i) < 1/2\} \\ V_{1/2} &= \{v_i : \alpha(x_i) = 1/2\} \\ V_+ &= \{v_i : \alpha(x_i) > 1/2\} \end{aligned}$$

Properties of LP optimal solution

- Let $\alpha : X \to \mathbb{R}_{\geq 0}$ be an optimal solution for $\mathsf{LP}_{\mathsf{VC}}(G)$. Let

$$\begin{aligned} V_- &= \{v_i : \alpha(x_i) < 1/2\} \\ V_{1/2} &= \{v_i : \alpha(x_i) = 1/2\} \\ V_+ &= \{v_i : \alpha(x_i) > 1/2\} \end{aligned}$$

Lemma 5

For each $i, 1 \leq i \leq n$, we have that $\alpha(x_i) \leq 1$.

Lemma 6

V_- *is an independent set.*

Lemma 7

$N_G(V_-) = V_+$.

Properties of LP optimal solution II

Lemma 8

For each $S \subseteq V_+$ *we have that* $|S| \leq |N_G(S) \cap V_-|$.

Proof.

For the sake of contradiction, suppose there is a set $S \subseteq V_+$ such that $|S| > |N_G(S) \cap V_-|$.
Let $\epsilon = \min_{v_i \in S}\{\alpha(x_i) - 1/2\}$ and $\alpha' : X \to \mathbb{R}_{\geq 0}$ s.t.

$$\alpha'(x_i) = \begin{cases} \alpha(x_i) & \text{if } v_i \notin S \cup (N_G(S) \cap V_-) \\ \alpha(x_i) - \epsilon & \text{if } v_i \in S \\ \alpha(x_i) + \epsilon & \text{if } v_i \in N_G(S) \cap V_- \end{cases}$$

Note that α' is an improved solution for $\mathrm{LP}_{\mathrm{VC}}(G)$, contradicting that α is optimal. □

Properties of LP optimal solution III

Theorem 9 (Hall's marriage theorem)

A bipartite graph $G = (V \uplus U, E)$ *has a matching saturating* $S \subseteq V$

$$\Leftrightarrow$$

for every subset $W \subseteq S$ *we have* $|W| \leq |N_G(W)|$. [1]

[1] A matching M in a graph G is a set of edges such that no two edges in M have a common endpoint. A matching saturates a set of vertices S if each vertex in S is an end point of an edge in M.

Properties of LP optimal solution III

Theorem 9 (Hall's marriage theorem)

A bipartite graph $G = (V \uplus U, E)$ *has a matching saturating* $S \subseteq V$

$$\Leftrightarrow$$

for every subset $W \subseteq S$ *we have* $|W| \leq |N_G(W)|$. [1]

Consider the bipartite graph $B = (V_- \uplus V_+, \{\{u,v\} \in E : u \in V_-, v \in V_+\})$.

Lemma 10

There exists a matching M *in* B *of size* $|V_+|$.

Proof.

The lemma follows from the previous lemma and Hall's marriage theorem. □

[1] A matching M in a graph G is a set of edges such that no two edges in M have a common endpoint. A matching saturates a set of vertices S if each vertex in S is an end point of an edge in M.

Crown Decomposition: Definition

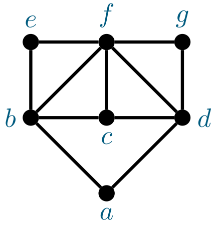
Definition 11 (Crown Decomposition)

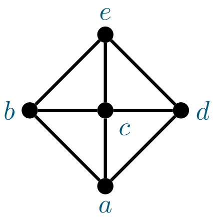
A crown decomposition (C, H, B) of a graph $G = (V, E)$ is a partition of V into sets C, H, and B such that

- the crown C is a non-empty independent set,
- the head $H = N_G(C)$,
- the body $B = V \setminus (C \cup H)$, and
- there is a matching of size $|H|$ in $G[H \cup C]$.

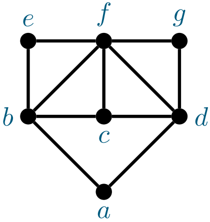
By the previous lemmas, we obtain a crown decomposition $(V_-, V_+, V_{1/2})$ of G if $V_- \neq \emptyset$.

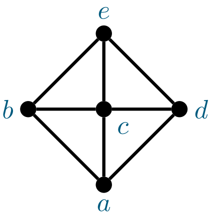
Crown Decomposition: Examples





Crown Decomposition: Examples





crown decomposition
$(\{a,e,g\},\{b,d,f\},\{c\})$

has no crown decomposition

Using the crown decomposition

Lemma 12

Suppose that $G = (V, E)$ *has a crown decomposition* (C, H, B). *Then,*

$$\mathit{vc}(G) \leq k \quad \Leftrightarrow \quad \mathit{vc}(G[B]) \leq k - |H|,$$

where $\mathit{vc}(G)$ *denotes the size of the smallest vertex cover of* G.

Using the crown decomposition

Lemma 12

Suppose that $G = (V, E)$ *has a crown decomposition* (C, H, B). *Then,*

$$\mathit{vc}(G) \leq k \quad \Leftrightarrow \quad \mathit{vc}(G[B]) \leq k - |H|,$$

where $\mathit{vc}(G)$ *denotes the size of the smallest vertex cover of* G.

Proof.

$(\Rightarrow)$: Let S be a vertex cover of G with $|S| \leq k$. Since S contains at least one vertex for each edge of a matching, $|S \cap (C \cup H)| \geq |H|$. Therefore, $S \cap B$ is a vertex cover for $G[B]$ of size at most $k - |H|$.
$(\Leftarrow)$: Let S be a vertex cover of $G[B]$ with $|S| \leq k - |H|$. Then, $S \cup H$ is a vertex cover of G of size at most k, since each edge that is in G but not in $G[B]$ is incident to a vertex in H. □

Nemhauser-Trotter

Corollary 13 ((Nemhauser and Trotter Jr., 1974))

There exists a smallest vertex cover S *of* G *such that* $S \cap V_- = \emptyset$ *and* $V_+ \subseteq S$.

Corollary 14 ((Nemhauser and Trotter Jr., 1974))

Vertex Cover *has a 2-approximation algorithm.*

Crown reduction

(Crown Reduction)

If solving LP$_{VC}(G)$ gives an optimal solution with $V_- \neq \emptyset$, then return $(G - (V_- \cup V_+), k - |V_+|)$.

Crown reduction

(Crown Reduction)

If solving $\text{LP}_{VC}(G)$ gives an optimal solution with $V_- \neq \emptyset$, then return $(G - (V_- \cup V_+), k - |V_+|)$.

(Number of Vertices)

If solving $\text{LP}_{VC}(G)$ gives an optimal solution with $V_- = \emptyset$ and $|V| > 2k$, then return $\textsc{No}$.

Crown reduction

(Crown Reduction)

If solving $\text{LP}_{VC}(G)$ gives an optimal solution with $V_- \neq \emptyset$, then return $(G - (V_- \cup V_+), k - |V_+|)$.

(Number of Vertices)

If solving $\text{LP}_{VC}(G)$ gives an optimal solution with $V_- = \emptyset$ and $|V| > 2k$, then return No.

Lemma 15

(Crown Reduction) and (Number of Vertices) are sound.

Proof.

(Crown Reduction) is sound by previous Lemmas.
Let α be an optimal solution for $\text{LP}_{VC}(G)$ and suppose $V_- = \emptyset$. The value of this solution is at least $|V|/2$. Thus, the value of an optimal solution for $\text{ILP}_{\text{VC}}(G)$ is at least $|V|/2$. Since G has no vertex cover of size less than $|V|/2$, we have a No-instance if $k < |V|/2$. □

Linear vertex-kernel for Vertex Cover

Theorem 16

Vertex Cover *has a kernel with* $2k$ *vertices and* $O(k^2)$ *edges.*

This is the smallest known kernel for Vertex Cover.
See http://fpt.wikidot.com/fpt-races for the current smallest kernels for various problems.

Outline

Crown Decomposition: Definition

Recall:

Definition 17 (Crown Decomposition)

A crown decomposition (C, H, B) of a graph $G = (V, E)$ is a partition of V into sets C, H, and B such that

- the crown C is a non-empty independent set,
- the head $H = N_G(C)$,
- the body $B = V \setminus (C \cup H)$, and
- there is a matching of size $|H|$ in $G[H \cup C]$.

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

To prove the lemma, we need Kőnig's Theorem

Theorem 19 ((Kőnig, 1931))

In every bipartite graph the size of a maximum matching is equal to the size of a minimum vertex cover.

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.

□

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k + 1$ vertices.

$\square$

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k+1$. There is a polynomial time algorithm that either

- *finds a matching of size $k+1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k+1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k+1$ vertices.
Consider the bipartite graph B formed by edges with one endpoint in $V(M)$ and the other in I.

□

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k + 1$ vertices.
Consider the bipartite graph B formed by edges with one endpoint in $V(M)$ and the other in I.
Compute a minimum vertex cover X and a maximum matching M' of B.

□

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k + 1$ vertices.
Consider the bipartite graph B formed by edges with one endpoint in $V(M)$ and the other in I.
Compute a minimum vertex cover X and a maximum matching M' of B.
We know: $|X| = |M'| \leq |M| \leq k$. Hence, $X \cap V(M) \neq \emptyset$.

□

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k + 1$ vertices.
Consider the bipartite graph B formed by edges with one endpoint in $V(M)$ and the other in I.
Compute a minimum vertex cover X and a maximum matching M' of B.
We know: $|X| = |M'| \leq |M| \leq k$. Hence, $X \cap V(M) \neq \emptyset$.
Let $M^* = \{e \in M' : e \cap (X \cap V(M)) \neq \emptyset\}$.

□

Crown Lemma

Lemma 18 (Crown Lemma)

Let $G = (V, E)$ be a graph without isolated vertices and with $|V| \geq 3k + 1$. There is a polynomial time algorithm that either

- *finds a matching of size $k + 1$ in G, or*
- *finds a crown decomposition of G.*

Proof.

Compute a maximum matching M of G. If $|M| \geq k + 1$, we are done.
Note that $I := V \setminus V(M)$ is an independent set with $\geq k + 1$ vertices.
Consider the bipartite graph B formed by edges with one endpoint in $V(M)$ and the other in I.
Compute a minimum vertex cover X and a maximum matching M' of B.
We know: $|X| = |M'| \leq |M| \leq k$. Hence, $X \cap V(M) \neq \emptyset$.
Let $M^* = \{e \in M' : e \cap (X \cap V(M)) \neq \emptyset\}$.
We obtain a crown decomposition with crown $C = V(M^*) \cap I$ and head $H = X \cap V(M) = X \cap V(M^*)$. □

After computing a kernel ...

- ... we can use any algorithm to compute an actual solution.
- Brute-force, faster exponential-time algorithms, parameterized algorithms, often also approximation algorithms

Kernels

- A parameterized problem may not have a kernelization algorithm
  - Example, COLORING[2] parameterized by k has no kernelization algorithm unless P $=$ NP.
  - A kernelization would lead to a polynomial time algorithm for the NP-complete 3-COLORING problem
- Only exponential kernels may be known for a parameterized problem
- There is a theory of kernel lower bounds, establishing exponential lower bounds on the kernel size of certain parameterized problems.

[2] Can one color the vertices of an input graph G with k colors such that no two adjacent vertices receive the same color?

Approximation algorithms

Besides constant factor approximation algorithms, positive results include:

- additive approximation (rare)
- polynomial time approximation schemes (PTAS): able to achive an approximation ratio $1+\epsilon$ for any constant ϵ in polynomial time, but the running time depends on $1/\epsilon$. Restrictions include EPTAS (Efficient PTAS) and FPTAS (Fully PTAS), restricting how the running time may depend on the parameter $1/\epsilon$.

Negative results include

- no factor-c approximation algorithm unless $\mathsf{P} = \mathsf{NP}$ / unless the Unique Games conjecture fails, etc.
- APX-hardness, ruling out PTASs

Outline

Further Reading

- Vazirani's textbook (Vazirani, 2003)
- Fellows et al.'s survey on Vertex Cover kernelization (Fellows et al., 2018)

References I

Michael B. Cohen, Yin Tat Lee, and Zhao Song (2019). "Solving linear programs in the current matrix multiplication time". In: *Proceedings of the 51st Annual ACM SIGACT Symposium on Theory of Computing (STOC 2019)*. ACM, pp. 938–942.

Elias Dahlhaus, David S. Johnson, Christos H. Papadimitriou, Paul D. Seymour, and Mihalis Yannakakis (1994). "The Complexity of Multiterminal Cuts". In: *SIAM Journal on Computing* 23.4, pp. 864–894. DOI: 10.1017/9781107415157.

Michael R. Fellows, Lars Jaffke, Aliz Izabella Király, Frances A. Rosamond, and Mathias Weller (2018). "What Is Known About Vertex Cover Kernelization?" In: *Adventures Between Lower Bounds and Higher Altitudes: Essays Dedicated to Juraj Hromkovič on the Occasion of His 60th Birthday*. Ed. by Hans-Joachim Böckenhauer, Dennis Komm, and Walter Unger. Springer, pp. 330–356.

Dénes Kőnig (1931). "Gráfok és mátrixok". In: *Matematikai és Fizikai Lapok* 38, pp. 116–119.

References II

George L. Nemhauser and Leslie E. Trotter Jr. (1974). "Properties of vertex packing and independence system polyhedra". In: *Math. Program.* 6.1, pp. 48–61.

Vijay V. Vazirani (2003). *Approximation Algorithms*. Springer.

Basics of Parameterized Complexity

Serge Gaspers

UNSW

Outline

1. Introduction
   - Vertex Cover
   - Coloring
   - Clique
   - Δ-Clique

2. Basic Definitions

3. Further Reading

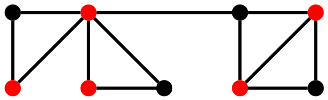
Outline

Outline

Vertex Cover

A vertex cover in a graph $G = (V, E)$ is a subset of its vertices $S \subseteq V$ such that every edge of G has at least one endpoint in S.

VERTEX COVER	
Input:	A graph $G = (V, E)$ and an integer k
Parameter:	k
Question:	Does G have a vertex cover of size k?



Algorithms for Vertex Cover

- brute-force: $O^*(2^n)$
- brute-force: $O^*(n^k)$
- vc1: $O^*(2^k)$
- vc2: $O^*(1.4656^k)$
- (Chen, Kanj, and Xia, 2010): $O(1.2738^k + k \cdot n)$

Running times in practice

$n = 1000$ vertices,
$k = 20$ parameter

	Running Time	
Theoretical	Nb of Instructions	Real
2^n	$1.07 \cdot 10^{301}$	$4.941 \cdot 10^{282}$ years
n^k	10^{60}	$4.611 \cdot 10^{41}$ years
$2^k \cdot n$	$1.05 \cdot 10^9$	15.26 milliseconds
$1.4656^k \cdot n$	$2.10 \cdot 10^6$	0.31 milliseconds
$1.2738^k + k \cdot n$	$2.02 \cdot 10^4$	0.0003 milliseconds

Notes:
– We assume that 2^{36} instructions are carried out per second.
– The Big Bang happened roughly $13.5 \cdot 10^9$ years ago.

Goal of Parameterized Complexity

Confine the combinatorial explosion to a parameter k.

pc.jpg

(1) Which problem–parameter combinations are fixed-parameter tractable (FPT)? In other words, for which problem–parameter combinations are there algorithms with running times of the form

$$f(k) \cdot n^{O(1)},$$

where the f is a computable function independent of the input size n?

(2) How small can we make the $f(k)$?

Examples of Parameters

A Parameterized Problem

Input:	an instance of the problem
Parameter:	a parameter
Question:	a YES–NO question about the instance and the parameter

- A parameter can be
  - solution size
  - input size (trivial parameterization)
  - related to the structure of the input (maximum degree, treewidth, branchwidth, genus, ...)
  - combinations of parameters
  - etc.

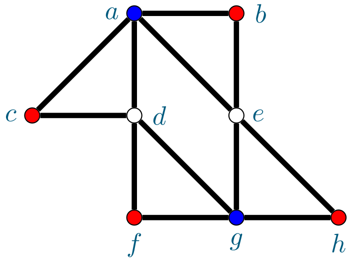
Outline

1. Introduction
   - Vertex Cover
   - Coloring
   - Clique
   - Δ-Clique
2. Basic Definitions
3. Further Reading

Coloring

A k-coloring of a graph $G = (V, E)$ is a function $f : V \to \{1, 2, ..., k\}$ assigning colors to V such that no two adjacent vertices receive the same color.

COLORING
Input: Graph G, integer k
Parameter: k
Question: Does G have a k-coloring?



Brute-force: $O^*(k^n)$, where $n = |V(G)|$.
(Björklund, Husfeldt, and Koivisto, 2009): $O^*(2^n)$ by inclusion-exclusion

Coloring is probably not FPT

- Known: Coloring is NP-complete when $k = 3$
- Suppose there was a $O^*(f(k))$-time algorithm for Coloring
  - Then, 3-Coloring can be solved in $O^*(f(3)) \subseteq O^*(1)$ time
  - Therefore, P $=$ NP
- Therefore, Coloring is not FPT unless P $=$ NP

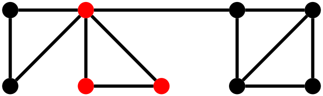
Outline

Clique

A clique in a graph $G = (V, E)$ is a subset of its vertices $S \subseteq V$ such that every two vertices from S are adjacent in G.

Clique

Input:	Graph $G = (V, E)$, integer k
Parameter:	k
Question:	Does G have a clique of size k?



Is Clique NP-complete when k is a fixed constant? Is it FPT?

Algorithm for Clique

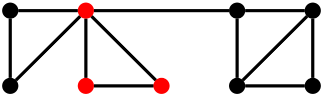
- For each subset $S \subseteq V$ of size k, check whether all vertices of S are adjacent
- Running time: $O^*\left(\binom{n}{k}\right) \subseteq O^*(n^k)$
- When $k \in O(1)$, this is polynomial
- But: we do not currently know an FPT algorithm for Clique
- Since Clique is W[1]-hard, we believe it is not FPT. (See lecture on W-hardness.)

Outline

A different parameter for Clique

Δ-CLIQUE

Input:	Graph $G = (V, E)$, integer k
Parameter:	$\Delta(G)$, i.e., the maximum degree of G
Question:	Does G have a clique of size k?



Is Δ-CLIQUE FPT?

Algorithm for Δ-Clique

Input: A graph G and an integer k.
Output: YES if G has a clique of size k, and NO otherwise.

if $k = 0$ **then**
  return YES
else if $k > \Delta(G) + 1$ **then**
  return NO
else
  /* A clique of size k contains at least one vertex v.

Algorithm for Δ-Clique

Input: A graph G and an integer k.
Output: YES if G has a clique of size k, and NO otherwise.

if $k = 0$ **then**
  return YES
else if $k > \Delta(G) + 1$ **then**
  return NO
else
  `/* A clique of size` k `contains at least one vertex` v`.`
  `For each` $v \in V$`, we check whether` G `has a` k`-clique` S
  `containing` v `(note that` $S \subseteq N_G[v]$ `in this case). */`

Algorithm for Δ-Clique

Input: A graph G and an integer k.
Output: Yes if G has a clique of size k, and No otherwise.

```
if k = 0 then
  return Yes
else if k > Δ(G) + 1 then
  return No
else
  /* A clique of size k contains at least one vertex v.
     For each v ∈ V, we check whether G has a k-clique S
     containing v (note that S ⊆ N_G[v] in this case).        */
  foreach v ∈ V do
    foreach S ⊆ N_G[v] with |S| = k do
      if S is a clique in G then
        return Yes
  return No
```

Algorithm for Δ-Clique

Input: A graph G and an integer k.
Output: YES if G has a clique of size k, and NO otherwise.

if $k = 0$ **then**
  return YES
else if $k > \Delta(G) + 1$ **then**
  return NO
else
  `/* A clique of size` k `contains at least one vertex` v`.`
  `For each` $v \in V$`, we check whether` G `has a` k`-clique` S
  `containing` v `(note that` $S \subseteq N_G[v]$ `in this case). */`
  foreach $v \in V$ **do**
    foreach $S \subseteq N_G[v]$ *with* $|S| = k$ **do**
      if S *is a clique in* G **then**
        return YES
  return NO

Running time: $O^*((\Delta + 1)^k) \subseteq O^*((\Delta + 1)^\Delta)$. (FPT for parameter Δ)

Outline

Main Parameterized Complexity Classes

n: instance size
k: parameter

P: class of problems that can be solved in $n^{O(1)}$ time
FPT: class of parameterized problems that can be solved in $f(k) \cdot n^{O(1)}$ time
XP: class of parameterized problems that can be solved in $f(k) \cdot n^{g(k)}$ time
("polynomial when k is a constant")

$$\mathsf{P} \subseteq \mathsf{FPT} \subseteq \mathsf{W}[1] \subseteq \mathsf{W}[2] \cdots \subseteq \mathsf{W}[P] \subseteq \mathsf{XP}$$

Known: If FPT $=$ W[1], then the Exponential Time Hypothesis fails, i.e. 3-SAT can be solved in $2^{o(n)}$ time, where n is the number of variables.

Note: We assume that f is computable and non-decreasing.

Outline

Further Reading

- Chapter 1, *Introduction* in (Cygan et al., 2015)
- Chapter 2, *The Basic Definitions* in (Downey and Fellows, 2013)
- Chapter I, *Foundations* in (Niedermeier, 2006)
- *Preface* in (Flum and Grohe, 2006)

References I

Andreas Björklund, Thore Husfeldt, and Mikko Koivisto (2009). "Set Partitioning via Inclusion-Exclusion". In: *SIAM Journal on Computing* 39.2, pp. 546–563.

Jianer Chen, Iyad A. Kanj, and Ge Xia (2010). "Improved upper bounds for vertex cover". In: *Theoretical Computer Science* 411.40-42, pp. 3736–3756. DOI: 10.1016/j.tcs.2010.06.026.

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Jörg Flum and Martin Grohe (2006). *Parameterized Complexity Theory*. Springer. DOI: 10.1007/3-540-29953-X.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Parameterized intractability: the W-hierarchy

Serge Gaspers

UNSW

Outline

1. Parameterized Complexity Theory
   - Parameterized reductions
   - Parameterized complexity classes
2. Case study
3. Further Reading

Outline

1. Parameterized Complexity Theory
   - Parameterized reductions
   - Parameterized complexity classes
2. Case study
3. Further Reading

Main Parameterized Complexity Classes

n: instance size
k: parameter

P: class of problems that can be solved in $n^{O(1)}$ time
FPT: class of parameterized problems that can be solved in $f(k) \cdot n^{O(1)}$ time
W[$\cdot$]: parameterized intractability classes
XP: class of parameterized problems that can be solved in $f(k) \cdot n^{g(k)}$ time
("polynomial when k is a constant")

$$\text{P} \subseteq \text{FPT} \subseteq \text{W}[1] \subseteq \text{W}[2] \cdots \subseteq \text{W}[P] \subseteq \text{XP}$$

Note: We assume that f is computable and non-decreasing.

Polynomial-time reductions for parameterized problems?

A vertex cover in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every edge of G has an endpoint in S.

Vertex Cover	
Input:	Graph G, integer k
Parameter:	k
Question:	Does G have a vertex cover of size k?

An independent set in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that there is no edge $uv \in E$ with $u, v \in S$.

Independent Set	
Input:	Graph G, integer k
Parameter:	k
Question:	Does G have an independent set of size k?

Polynomial-time reductions for parameterized problems?

A vertex cover in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that every edge of G has an endpoint in S.

> VERTEX COVER
> Input: Graph G, integer k
> Parameter: k
> Question: Does G have a vertex cover of size k?

An independent set in a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ such that there is no edge $uv \in E$ with $u, v \in S$.

> INDEPENDENT SET
> Input: Graph G, integer k
> Parameter: k
> Question: Does G have an independent set of size k?

- We know: INDEPENDENT SET $\leq_{\mathsf{P}}$ VERTEX COVER
- However: VERTEX COVER $\in$ FPT but INDEPENDENT SET is not known to be in FPT

We will need another type of reductions

- Issue with polynomial-time reductions: parameter can change arbitrarily.

We will need another type of reductions

- Issue with polynomial-time reductions: parameter can change arbitrarily.
- We will want the reduction to produce an instance where the parameter is bounded by a function of the parameter of the original instance.

We will need another type of reductions

- Issue with polynomial-time reductions: parameter can change arbitrarily.
- We will want the reduction to produce an instance where the parameter is bounded by a function of the parameter of the original instance.
- Also: we can allow the reduction to take FPT time instead of only polynomial time.

Outline

1. Parameterized Complexity Theory
   - Parameterized reductions
   - Parameterized complexity classes
2. Case study
3. Further Reading

Parameterized reduction

Definition 1

A parameterized reduction from a parameterized decision problem Π_1 to a parameterized decision problem Π_2 is an algorithm, which, for any instance I of Π_1 with parameter k produces an instance I' of Π_2 with parameter k' such that

- I is a Yes-instance for Π_1 $\Leftrightarrow$ I' is a Yes-instance for Π_2,
- there exists a computable function g such that $k' \leq g(k)$, and
- there exists a computable function f such that the running time of the algorithm is $f(k) \cdot |I|^{O(1)}$.

If there exists a parameterized reduction from Π_1 to Π_2, we write $\Pi_1 \leq_{\mathsf{FPT}} \Pi_2$.

Note: We can assume that f and g are non-decreasing.

New FPT algorithms via reductions

Lemma 2

If Π_1, Π_2 are parameterized decision problems such that $\Pi_1 \leq_{\mathsf{FPT}} \Pi_2$, then $\Pi_2 \in \mathsf{FPT}$ implies $\Pi_1 \in \mathsf{FPT}$.

Proof sketch.

To obtain an FPT algorithm for Π_1, perform the reduction and then use an FPT algorithm for Π_2 on the resulting instance. $\square$

Outline

1. Parameterized Complexity Theory
   - Parameterized reductions
   - Parameterized complexity classes
2. Case study
3. Further Reading

Boolean Circuits

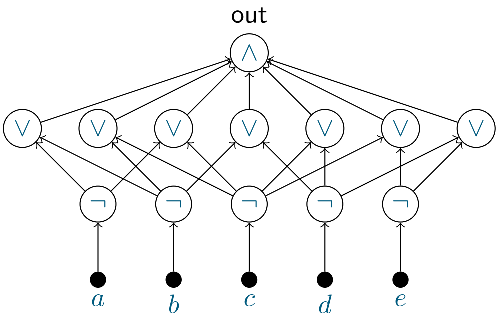
Definition 3

A Boolean circuit is a directed acyclic graph with the nodes labeled as follows:

- every node of in-degree 0 is an input node,
- every node with in-degree 1 is a negation node ($\neg$), and
- every node with in-degree ≥ 2 is either an AND-node ($\wedge$) or an OR-node ($\vee$).

Moreover, exactly one node with out-degree 0 is also labeled the output node.
The depth of the circuit is the maximum length of a directed path from an input node to the output node.
The weft of the circuit is the maximum number of nodes with in-degree ≥ 3 on a directed path from an input node to the output node.

Example



A depth-3, weft-1 Boolean circuit with inputs a, b, c, d, e.

Weighted Circuit Satisfiability

Given an assignment of Boolean values to the input gates, the circuit determines Boolean values at each node in the obvious way.
If the value of the output node is 1 for an input assignment, we say that this assignment satisfies the circuit.
The weight of an assignment is its number of 1s.

WEIGHTED CIRCUIT SATISFIABILITY (WCS)	
Input:	A Boolean circuit C, an integer k
Parameter:	k
Question:	Is there an assignment with weight k that satisfies C?

Exercise: Show that WEIGHTED CIRCUIT SATISFIABILITY $\in$ XP.

WCS for special circuits

Definition 4

The class of circuits $\mathcal{C}_{t,d}$ contains the circuits with weft $\leq t$ and depth $\leq d$.

For any class of circuits $\mathcal{C}$, we can define the following problem.

WCS[$\mathcal{C}$]

Input:	A Boolean circuit $C \in \mathcal{C}$, an integer k
Parameter:	k
Question:	Is there an assignment with weight k that satisfies C?

W classes

Definition 5 (W-hierarchy)

Let $t \in \{1, 2, \dots\}$. A parameterized problem Π is in the parameterized complexity class W$[t]$ if there exists a parameterized reduction from Π to WCS$[\mathcal{C}_{t,d}]$ for some constant $d \geq 1$.

Independent Set and Dominating Set

Theorem 6

INDEPENDENT SET $\in$ W[1].

Theorem 7

DOMINATING SET $\in$ W[2].

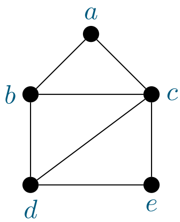
Recall: A dominating set of a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that $N_G[S] = V$.

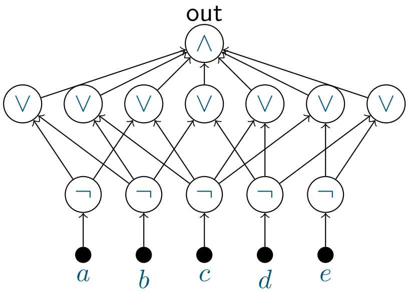
DOMINATING SET

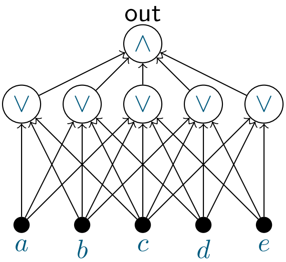
Input:	A graph $G = (V, E)$ and an integer k
Parameter:	k
Question:	Does G have a dominating set of size at most k?

"Proof" by picture

Parameterized reductions from Independent Set to WCS[$\mathcal{C}_{1,3}$] and from Dominating Set to WCS[$\mathcal{C}_{2,2}$].







Setting an input node to 1 corresponds to adding the corresponding vertex to the independent set / dominating set.

W-hardness

Definition 8

Let $t \in \{1, 2, \ldots\}$.
A parameterized decision problem Π is W[t]-hard if for every parameterized decision problem Π' in W[t], there is a parameterized reduction from Π' to Π.
Π is W[t]-complete if $\Pi \in$ W[t] and Π is W[t]-hard.

W-hardness

Definition 8

Let $t \in \{1, 2, \ldots\}$.
A parameterized decision problem Π is W[t]-hard if for every parameterized decision problem Π' in W[t], there is a parameterized reduction from Π' to Π.
Π is W[t]-complete if $\Pi \in$ W[t] and Π is W[t]-hard.

Theorem 9 ((Downey and Fellows, 1995b))

INDEPENDENT SET *is* W[1]*-complete.*

Theorem 10 ((Downey and Fellows, 1995a))

DOMINATING SET *is* W[2]*-complete.*

Proving W-hardness

To show that a parameterized decision problem Π is W[t]-hard:

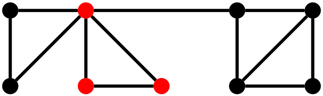
- Select a W[t]-hard problem Π'
- Show that $\Pi' \leq_{\mathsf{FPT}} \Pi$ by designing a parameterized reduction from Π' to Π
  - Design an algorithm, that, for any instance I' of Π' with parameter k', produces an equivalent instance I of Π with parameter k
  - Show that k is upper bounded by a function of k'
  - Show that there exists a function f such that the running time of the algorithm is $f(k') \cdot |I'|^{O(1)}$

Outline

Clique

A clique in a graph $G = (V, E)$ is a subset of its vertices $S \subseteq V$ such that every two vertices from S are adjacent in G.

CLIQUE	
Input:	Graph $G = (V, E)$, integer k
Parameter:	k
Question:	Does G have a clique of size k?



- We will show that CLIQUE is W[1]-hard by a parameterized reduction from INDEPENDENT SET.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\mathsf{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\text{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.
Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\text{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.

Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Equivalence. We need to show that (G, k) is a YES-instance for INDEPENDENT SET if and only if (G', k') is a YES-instance for CLIQUE.

Clique is W[1]-hard

Lemma 11

Independent Set $\leq_{\mathsf{FPT}}$ Clique.

Proof.

Given any instance $(G = (V, E), k)$ for Independent Set, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for Clique such that $k' \leq g(k)$ for some computable function g.

Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Equivalence. We need to show that (G, k) is a Yes-instance for Independent Set if and only if (G', k') is a Yes-instance for Clique.

$(\Rightarrow)$: Let S be an independent set of size k in G. For every two vertices $u, v \in S$, we have that $uv \notin E$. Therefore, $uv \in E(\overline{G})$ for every two vertices in S. We conclude that S is a clique of size k in $\overline{G}$.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\text{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.

Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Equivalence. We need to show that (G, k) is a YES-instance for INDEPENDENT SET if and only if (G', k') is a YES-instance for CLIQUE.

$(\Rightarrow)$: Let S be an independent set of size k in G. For every two vertices $u, v \in S$, we have that $uv \notin E$. Therefore, $uv \in E(\overline{G})$ for every two vertices in S. We conclude that S is a clique of size k in $\overline{G}$.

$(\Leftarrow)$: Let S be a clique of size k in $\overline{G}$. By a similar argument, S is an independent set of size k in G.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\text{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.

Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Equivalence. We need to show that (G, k) is a YES-instance for INDEPENDENT SET if and only if (G', k') is a YES-instance for CLIQUE.

$(\Rightarrow)$: Let S be an independent set of size k in G. For every two vertices $u, v \in S$, we have that $uv \notin E$. Therefore, $uv \in E(\overline{G})$ for every two vertices in S. We conclude that S is a clique of size k in $\overline{G}$.

$(\Leftarrow)$: Let S be a clique of size k in $\overline{G}$. By a similar argument, S is an independent set of size k in G.

Parameter. $k' \leq k$.

Clique is W[1]-hard

Lemma 11

INDEPENDENT SET $\leq_{\text{FPT}}$ CLIQUE.

Proof.

Given any instance $(G = (V, E), k)$ for INDEPENDENT SET, we need to describe an FPT algorithm that constructs an equivalent instance (G', k') for CLIQUE such that $k' \leq g(k)$ for some computable function g.

Construction. Set $k' \leftarrow k$ and $G' \leftarrow \overline{G} = (V, \{uv : u, v \in V, u \neq v, uv \notin E\})$.

Equivalence. We need to show that (G, k) is a YES-instance for INDEPENDENT SET if and only if (G', k') is a YES-instance for CLIQUE.

$(\Rightarrow)$: Let S be an independent set of size k in G. For every two vertices $u, v \in S$, we have that $uv \notin E$. Therefore, $uv \in E(\overline{G})$ for every two vertices in S. We conclude that S is a clique of size k in $\overline{G}$.

$(\Leftarrow)$: Let S be a clique of size k in $\overline{G}$. By a similar argument, S is an independent set of size k in G.

Parameter. $k' \leq k$.

Running time. The construction can clearly be done in FPT time, and even in polynomial time. $\square$

Clique is W[1]-hard II

Corollary 12

CLIQUE *is* W[1]*-hard*

Outline

Further Reading

- Chapter 13, *Fixed-parameter Intractability* in (Cygan et al., 2015)
- Chapter 13, *Parameterized Complexity Theory* in (Niedermeier, 2006)
- Elements of Chapters 20–23 in (Downey and Fellows, 2013)

References I

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (1995a). "Fixed-Parameter Tractability and Completeness I: Basic Results". In: *SIAM Journal on Computing* 24.4, pp. 873–921.

– (1995b). "Fixed-Parameter Tractability and Completeness II: On Completeness for W[1]". In: *Theoretical Computer Science* 141.1&2, pp. 109–131.

– (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Branching algorithms

Serge Gaspers

UNSW

Outline

Outline

1. Branching algorithms
2. Running time analysis
3. Feedback Vertex Set
4. Maximum Leaf Spanning Tree
5. Further Reading

Branching Algorithm

Branching Algorithm

- Selection: Select a local configuration of the problem instance
- Recursion: Recursively solve subinstances
- Combination: Compute a solution of the instance based on the solutions of the subinstances

- Halting rule: 0 recursive calls
- Simplification rule: 1 recursive call
- Branching rule: ≥ 2 recursive calls

Example: Our first VERTEX COVER algorithm

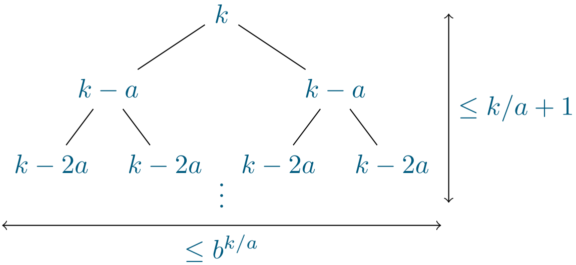
Algorithm vc1(G, k);

```
if E = ∅ then                                     // all edges are covered
    return Yes
else if k ≤ 0 then                                // we cannot select any vertex
    return No
else
    Select an edge uv ∈ E;
    return vc1(G − u, k − 1) ∨ vc1(G − v, k − 1)
```

Outline

Search trees

Recall: A search tree models the recursive calls of an algorithm.
For a b-way branching where the parameter k decreases by a at each recursive call, the number of nodes is at most $b^{k/a} \cdot (k/a + 1)$.



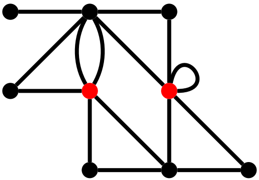
If k/a and b are upper bounded by a function of k, and the time spent at each node is FPT (typically, polynomial), then we get an FPT running time.

Outline

Feedback Vertex Set

A feedback vertex set of a multigraph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that $G - S$ is acyclic.

FEEDBACK VERTEX SET
- Input: Multigraph $G = (V, E)$, integer k
- Parameter: k
- Question: Does G have a feedback vertex set of size at most k?



Simplification Rules

We apply the first applicable[1] simplification/halting rule.

(Finished)

If G is acyclic and $k \geq 0$, then return YES.

(Budget-exceeded)

If $k < 0$, then return NO.

[1] A rule is applicable if it modifies the instance.

Simplification Rules

We apply the first applicable[1] simplification/halting rule.

(Finished)

If G is acyclic and $k \geq 0$, then return Yes.

(Budget-exceeded)

If $k < 0$, then return No.

(Loop)

If G has a loop $vv \in E$, then set $G \leftarrow G - v$ and $k \leftarrow k - 1$.

[1] A rule is applicable if it modifies the instance.

Simplification Rules

We apply the first applicable[1] simplification/halting rule.

(Finished)

If G is acyclic and $k \geq 0$, then return YES.

(Budget-exceeded)

If $k < 0$, then return NO.

(Loop)

If G has a loop $vv \in E$, then set $G \leftarrow G - v$ and $k \leftarrow k - 1$.

(Multiedge)

If E contains an edge uv more than twice, remove all but two copies of uv.

[1] A rule is applicable if it modifies the instance.

Simplification Rules II

(Degree-1)

If $\exists v \in V$ with $d_G(v) \leq 1$, then set $G \leftarrow G - v$.

Simplification Rules III

(Degree-2)

If $\exists v \in V$ with $d_G(v) = 2$, then denote vu, vw its two incident edges and set $G \leftarrow G' = (V \setminus \{v\}, (E \setminus \{vu, vw\}) \cup \{uw\})$.

Simplification Rules III

(Degree-2)

If $\exists v \in V$ with $d_G(v) = 2$, then denote vu, vw its two incident edges and set $G \leftarrow G' = (V \setminus \{v\}, (E \setminus \{vu, vw\}) \cup \{uw\})$.

Lemma 1

(Degree-2) is sound.

Proof.

Suppose S is a feedback vertex set of G of size at most k. Let

$$S' = \begin{cases} S & \text{if } v \notin S \\ (S \setminus \{v\}) \cup \{u\} & \text{if } v \in S. \end{cases}$$

Now, $|S'| \leq k$ and S' is a feedback vertex set of G' since every cycle in G' corresponds to a cycle in G, with, possibly, the edge uw replaced by the walk (u, v, w).

Suppose S' is a feedback vertex set of G' of size at most k. Then, S' is also a feedback vertex set of G. □

Remaining issues

- A select–discard branching decreases k in only one branch
- One could branch on all the vertices of a cycle, but the length of a shortest cycle might not be bounded by any function of k

Remaining issues

- A select–discard branching decreases k in only one branch
- One could branch on all the vertices of a cycle, but the length of a shortest cycle might not be bounded by any function of k

Idea:

- An acyclic graph has average degree < 2
- After applying simplification rules, G has average degree ≥ 3
- The selected feeback vertex set needs to be incident to many edges
- Does a feedback vertex set of size at most k contain at least one vertex among the $f(k)$ vertices of highest degree?

The fvs needs to be incident to many edges

Lemma 2

If S is a feedback vertex set of $G = (V, E)$, then

$$\sum_{v \in S} (d_G(v) - 1) \geq |E| - |V| + 1$$

The fvs needs to be incident to many edges

Lemma 2

If S is a feedback vertex set of $G = (V, E)$, then

$$\sum_{v \in S} (d_G(v) - 1) \geq |E| - |V| + 1$$

Proof.

Since $F = G - S$ is acyclic, $|E(F)| \leq |V| - |S| - 1$.
Since every edge in $E \setminus E(F)$ is incident with a vertex of S, we have

$$\begin{aligned}
|E| &= |E| - |E(F)| + |E(F)| \\
&\leq \left(\sum_{v \in S} d_G(v) \right) + (|V| - |S| - 1) \\
&= \left(\sum_{v \in S} (d_G(v) - 1) \right) + |V| - 1.
\end{aligned}$$

□

The fvs needs to contain a high-degree vertex

Lemma 3

Let G be a graph with minimum degree at least 3 and let H denote a set of $3k$ vertices of highest degree in G.
Every feedback vertex set of G of size at most k contains at least one vertex of H.

The fvs needs to contain a high-degree vertex

Lemma 3

Let G be a graph with minimum degree at least 3 and let H denote a set of $3k$ vertices of highest degree in G.
Every feedback vertex set of G of size at most k contains at least one vertex of H.

Proof.

Suppose not. Let S be a feedback vertex set with $|S| \leq k$ and $S \cap H = \emptyset$. Then,

$$\begin{aligned}
2|E| - |V| &= \sum_{v \in V} (d_G(v) - 1) \\
&= \sum_{v \in H} (d_G(v) - 1) + \sum_{v \in V \setminus H} (d_G(v) - 1) \\
&\geq 3 \cdot (\sum_{v \in S} (d_G(v) - 1)) + \sum_{v \in S} (d_G(v) - 1) \\
&\geq 4 \cdot (|E| - |V| + 1) \\
\Leftrightarrow \quad 3|V| &\geq 2|E| + 4.
\end{aligned}$$

But this contradicts the fact that every vertex of G has degree at least 3. $\square$

Algorithm for Feedback Vertex Set

Theorem 4

Feedback Vertex Set *can be solved in* $O^*((3k)^k)$ *time.*

Proof (sketch).

- Exhaustively apply the simplification rules.
- The branching rule computes H of size $3k$, and branches into subproblems $(G - v, k - 1)$ for each $v \in H$.

□

Outline

Maximum Leaf Spanning Tree

A leaf of a tree is a vertex with degree 1. A spanning tree in a graph $G = (V, E)$ is a subgraph of G that is a tree and has $|V|$ vertices.

MAXIMUM LEAF SPANNING TREE

Input:	connected graph G, integer k
Parameter:	k
Question:	Does G have a spanning tree with at least k leaves?

Property

A k-leaf tree in G is a subgraph of G that is a tree with at least k leaves.
A k-leaf spanning tree in G is a spanning tree in G with at least k leaves.

Lemma 5

Let $G = (V, E)$ be a connected graph.
G has a k-leaf tree $\Leftrightarrow$ G has a k-leaf spanning tree.

Proof.

$(\Leftarrow)$: trivial
$(\Rightarrow)$: Let T be a k-leaf tree in G. By induction on $x := |V| - |V(T)|$, we will show that T can be extended to a k-leaf spanning tree in G.
Base case: $x = 0$ $\checkmark$.
Induction: $x > 0$, and assume the claim is true for all $x' < x$. Choose $uv \in E$ such that $u \in V(T)$ and $v \notin V(T)$. Since $T' := (V(T) \cup \{v\}, E(T) \cup \{uv\})$ has $\geq k$ leaves and $< x$ external vertices, it can be extended to a k-leaf spanning tree in G by the induction hypothesis. $\square$

Strategy

- The branching algorithm will check whether G has a k-leaf tree.
- A tree with ≥ 3 vertices has at least one internal ($=$ non-leaf) vertex.
- "Guess" an internal vertex r, i.e., do a $|V|$-way branching fixing an initial internal vertex r.

Strategy

- The branching algorithm will check whether G has a k-leaf tree.
- A tree with ≥ 3 vertices has at least one internal ($=$ non-leaf) vertex.
- "Guess" an internal vertex r, i.e., do a $|V|$-way branching fixing an initial internal vertex r.
- In any branch, the algorithm has computed
  - T – a tree in G
  - I – the internal vertices of T, with $r \in I$
  - B – a subset of the leaves of T where T may be extended: the boundary set
  - L – the remaining leaves of T
  - X – the external vertices $V \setminus V(T)$

Strategy

- The branching algorithm will check whether G has a k-leaf tree.
- A tree with ≥ 3 vertices has at least one internal ($=$ non-leaf) vertex.
- "Guess" an internal vertex r, i.e., do a $|V|$-way branching fixing an initial internal vertex r.
- In any branch, the algorithm has computed
  - T – a tree in G
  - I – the internal vertices of T, with $r \in I$
  - B – a subset of the leaves of T where T may be extended: the boundary set
  - L – the remaining leaves of T
  - X – the external vertices $V \setminus V(T)$
- The question is whether T can be extended to a k-leaf tree where all the vertices in L are leaves.

Simplification Rules

Apply the first applicable simplification rule:

(Halt-Yes)

If $|L| + |B| \geq k$, then return Yes.

(Halt-No)

If $|B| = 0$, then return No.

(Non-extendable)

If $\exists v \in B$ with $N_G(v) \cap X = \emptyset$, then move v to L.

Branching Lemma

Lemma 6 (Branching Lemma)

Suppose $u \in B$ *and there exists a* k*-leaf tree* T' *extending* T *where* u *is an internal vertex.*
Then, there exists a k*-leaf tree* T'' *extending*
$(V(T) \cup N_G(u), E(T) \cup \{uv : v \in N_G(u) \cap X\})$.

Branching Lemma

Lemma 6 (Branching Lemma)

Suppose $u \in B$ *and there exists a* k*-leaf tree* T' *extending* T *where* u *is an internal vertex.*
Then, there exists a k*-leaf tree* T'' *extending*
$(V(T) \cup N_G(u), E(T) \cup \{uv : v \in N_G(u) \cap X\})$.

Proof.

Start from $T'' \leftarrow T'$ and perform the following operation for each $v \in N_G(u) \cap X$.
If $v \notin V(T')$, then add he vertex v and the edge uv.
Otherwise, add the edge uv, creating a cycle C in T and remove the other edge of C incident to v. This does not decrease the number of leaves, since it only increases the number of edges incident to u, and u was already internal. □

Follow Path Lemma

Lemma 7 (Follow Path Lemma)

Suppose $u \in B$ *and* $|N_G(u) \cap X| = 1$. *Let* $N_G(u) \cap X = \{v\}$.
If there exists a k*-leaf tree extending* T *where* u *is internal, but no* k*-leaf tree extending* T *where* u *is a leaf, then there exists a* k*-leaf tree extending* T *where both* u *and* v *are internal.*

Follow Path Lemma

Lemma 7 (Follow Path Lemma)

Suppose $u \in B$ *and* $|N_G(u) \cap X| = 1$. *Let* $N_G(u) \cap X = \{v\}$.
If there exists a k*-leaf tree extending* T *where* u *is internal, but no* k*-leaf tree extending* T *where* u *is a leaf, then there exists a* k*-leaf tree extending* T *where both* u *and* v *are internal.*

Proof.

Suppose not, and let T' be a k-leaf tree extending T where u is internal and v is a leaf. But then, $T - v$ is a k-leaf tree as well. □

Algorithm

- Apply halting & simplification rules
- Select $u \in B$. Branch into
  - $u \in L$
  - $u \in I$. In this case, add $X \cap N_G(u)$ to B (Branching Lemma).
    - In the special case where $|X \cap N_G(u)| = 1$, denote $\{v\} = X \cap N_G(u)$, make v internal, and add $N_G(v) \cap X$ to B, continuing the same way until reaching a vertex with at least 2 neighbors in X (Follow Path Lemma).
    - In the special case where $|X \cap N_G(u)| = 0$, return No.

Algorithm

- Apply halting & simplification rules
- Select $u \in B$. Branch into
  - $u \in L$
  - $u \in I$. In this case, add $X \cap N_G(u)$ to B (Branching Lemma).
    - In the special case where $|X \cap N_G(u)| = 1$, denote $\{v\} = X \cap N_G(u)$, make v internal, and add $N_G(v) \cap X$ to B, continuing the same way until reaching a vertex with at least 2 neighbors in X (Follow Path Lemma).
    - In the special case where $|X \cap N_G(u)| = 0$, return No.
- In one branch, a vertex moves from B to L; in the other branch, $|B|$ increases by at least 1.

Running time analysis

- Consider the "measure" $\mu := 2k - 2|L| - |B|$
- We have that $0 \leq \mu \leq 2k$
- Branch where $u \in L$:
  - $|B|$ decreases by 1, $|L|$ increases by 1
  - μ decreases by 1
- Branch where $u \in I$.
  - u moves from B to I
  - ≥ 2 vertices move from X to B
  - μ decreases by at least 1
- Binary search tree of height $\leq \mu \leq 2k$

Result for Maximum Leaf Spanning Tree

Theorem 8 ((Kneis, Langer, and Rossmanith, 2011))

MAXIMUM LEAF SPANNING TREE *can be solved in* $O^*(4^k)$ *time.*

Outline

Further Reading

- Chapter 3, *Bounded Search Trees* in (Cygan et al., 2015)
- Chapter 3, *Bounded Search Trees* in (Downey and Fellows, 2013)
- Chapter 8, *Depth-Bounded Search Trees* in (Niedermeier, 2006)

References I

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Joachim Kneis, Alexander Langer, and Peter Rossmanith (2011). "A New Algorithm for Finding Trees with Many Leaves". In: *Algorithmica* 61.4, pp. 882–897.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Measure & Conquer

Serge Gaspers

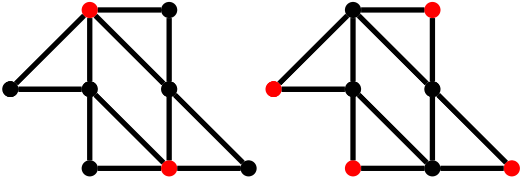
UNSW

Outline

Outline

Recall: Maximal Independent Sets

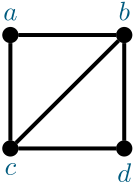
- A vertex set $S \subseteq V$ of a graph $G = (V, E)$ is an independent set in G if there is no edge $uv \in E$ with $u, v \in S$.
- An independent set is maximal if it is not a subset of any other independent set.
- Examples:



Enumeration problem: Enumerate all maximal independent sets

ENUM-MIS

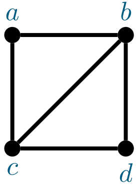
Input:	graph G
Output:	all maximal independent sets of G



Maximal independent sets: $\{a, d\}, \{b\}, \{c\}$

Enumeration problem: Enumerate all maximal independent sets

> **Enum-MIS**
> Input: graph G
> Output: all maximal independent sets of G



Maximal independent sets: $\{a, d\}, \{b\}, \{c\}$

Note: Let v be a vertex of a graph G. Every maximal independent set contains a vertex from $N_G[v]$.

Branching Algorithm for Enum-MIS

Algorithm **enum-mis**(G, I)
Input : A graph $G = (V, E)$, an independent set I of G.
Output: All maximal independent sets of G that are supersets of I.

```
G' ← G − N_G[I]
if V(G') = ∅ then                                   // G' has no vertex
    Output I
else
    Select v ∈ V(G') such that d_{G'}(v) = δ(G')    // v has min degree in G'
    Run enum-mis(G, I ∪ {u}) for each u ∈ N_{G'}[v]
```

Running Time Analysis

Let $L(n) = 2^{\alpha n}$ be an upper bound on the number of leaves in any search tree of **enum-mis** for an instance with $|V(G')| \leq n$.

We minimize α subject to constraints obtained from the branching:

$$
\begin{aligned}
& L(n) \geq (d+1) \cdot L(n-(d+1)) && \text{for each integer } d \geq 0. \\
\Leftrightarrow \quad & 2^{\alpha n} \geq d' \cdot 2^{\alpha \cdot (n-d')} && \text{for each integer } d' \geq 1. \\
\Leftrightarrow \quad & 1 \geq d' \cdot 2^{\alpha \cdot (-d')} && \text{for each integer } d' \geq 1.
\end{aligned}
$$

For fixed d', the smallest value for 2^α satisfying the constraint is ${d'}^{1/d'}$. The function $f(x) = x^{1/x}$ has its maximum value for $x = e$ and for integer x the maximum value of $f(x)$ is when $x = 3$.
Therefore, the minimum value for 2^α for which all constraints hold is $3^{1/3}$. We can thus set $L(n) = 3^{n/3}$.

Running Time Analysis II

Since the height of the search trees is $\leq |V(G')|$, we obtain:

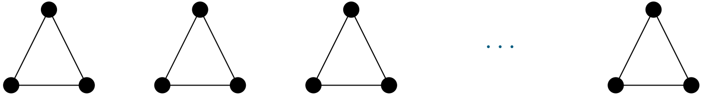
Theorem 1

Algorithm **enum-mis** *has running time* $O^*(3^{n/3}) \subseteq O(1.4423^n)$*, where* $n = |V|$*.*

Corollary 2

A graph on n *vertices has* $O(3^{n/3})$ *maximal independent sets.*

Running Time Lower Bound



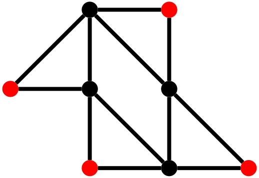
Theorem 3

There is an infinite family of graphs with $\Omega(3^{n/3})$ *maximal independent sets.*

Outline

Maximum Independent Set

Maximum Independent Set
Input: graph G
Output: A largest independent set of G.



Branching Algorithm for Maximum Independent Set

Algorithm **mis**(G)
Input : A graph $G = (V, E)$.
Output: The size of a maximum i.s. of G.

if $\Delta(G) \leq 2$ **then** // G has max degree ≤ 2
  return the size of a maximum i.s. of G in polynomial time
else if $\exists v \in V : d(v) = 1$ **then** // v has degree 1
  return $1 + \textbf{mis}(G - N[v])$
else if G is not connected **then**
  Let G_1 be a connected component of G
  return $\textbf{mis}(G_1) + \textbf{mis}(G - V(G_1))$
else
  Select $v \in V$ s.t. $d(v) = \Delta(G)$ // v has max degree
  return $\max\left(1 + \textbf{mis}(G - N[v]), \textbf{mis}(G - v)\right)$

Correctness

Line 4:

Lemma 4

If $v \in V$ has degree 1, then G has a maximum independent set I with $v \in I$.

Proof.

Let J be a maximum independent set of G.
If $v \in J$ we are done because we can take $I = J$.
If $v \notin J$, then $u \in J$, where u is the neighbor of v, otherwise J would not be maximum.
Set $I = (J \setminus \{u\}) \cup \{v\}$. We have that I is an independent set, and, since $|I| = |J|$, I is a maximum independent set containing v. □

Outline

Simple Analysis I

Lemma 5 (Simple Analysis Lemma)

Let

- A *be a branching algorithm*
- $\alpha > 0,\ c \geq 0$ *be constants*

such that on input I, A *calls itself recursively on instances* $I_1, \ldots, I_k$, *but, besides the recursive calls, uses time* $O(|I|^c)$, *such that*

$$(\forall i : 1 \leq i \leq k) \quad |I_i| \leq |I| - 1, \text{ and} \tag{1}$$

$$2^{\alpha \cdot |I_1|} + \cdots + 2^{\alpha \cdot |I_k|} \leq 2^{\alpha \cdot |I|}. \tag{2}$$

Then A *solves any instance* I *in time* $O(|I|^{c+1}) \cdot 2^{\alpha \cdot |I|}$.

Simple Analysis II

Proof.

By induction on $|I|$.
W.l.o.g., suppose the hypotheses' O statements hide a constant factor $d \geq 0$, and for the base case assume that the algorithm returns the solution to an empty instance in time $d \leq d \cdot |I|^{c+1} 2^{\alpha \cdot |I|}$.
Suppose the lemma holds for all instances of size at most $|I| - 1 \geq 0$, then the running time of algorithm A on instance I is

$$
\begin{aligned}
T_A(I) &\leq d \cdot |I|^c + \sum_{i=1}^{k} T_A(I_i) && \text{(by definition)} \\
&\leq d \cdot |I|^c + \sum d \cdot |I_i|^{c+1} 2^{\alpha \cdot |I_i|} && \text{(by the inductive hypothesis)} \\
&\leq d \cdot |I|^c + d \cdot (|I| - 1)^{c+1} \sum 2^{\alpha \cdot |I_i|} && \text{(by (1))} \\
&\leq d \cdot |I|^c + d \cdot (|I| - 1)^{c+1} 2^{\alpha \cdot |I|} && \text{(by (2))} \\
&\leq d \cdot |I|^{c+1} 2^{\alpha \cdot |I|}.
\end{aligned}
$$

The final inequality uses that $\alpha \cdot |I| > 0$ and holds for any $c \geq 0$. □

Simple Analysis for **mis**

- At each node of the search tree: $O(n^2)$ time
- G disconnected: let $s := |V(G_1)|$
  (1) If $\alpha \cdot s < 1$, then $s < 1/\alpha$, and the algorithm solves G_1 in constant time (provided that $\alpha > 0$). We can view this rule as a simplification rule, removing G_1 and making one recursive call on $G - V(G_1)$.
  (2) If $\alpha \cdot (n - s) < 1$: similar as (1).
  (3) Otherwise,

$$(\forall s : 1/\alpha \leq s \leq n - 1/\alpha) \quad 2^{\alpha \cdot s} + 2^{\alpha \cdot (n-s)} \leq 2^{\alpha \cdot n}. \tag{3}$$

  always satisfied since $2^x + 2^y \leq 2^{x+y}$ if $x, y \geq 1$.
- Branch on vertex of degree $d \geq 3$

$$(\forall d : 3 \leq d \leq n - 1) \quad 2^{\alpha \cdot (n-1)} + 2^{\alpha \cdot (n-1-d)} \leq 2^{\alpha n}. \tag{4}$$

  Dividing all these terms by $2^{\alpha n}$, the constraints become

$$2^{-\alpha} + 2^{\alpha \cdot (-1-d)} \leq 1. \tag{5}$$

Compute optimum α

The minimum α satisfying the constraints is obtained by solving a convex mathematical program minimizing α subject to the constraints (the constraint for $d = 3$ is sufficient as all other constraints are weaker).

Compute optimum α

The minimum α satisfying the constraints is obtained by solving a convex mathematical program minimizing α subject to the constraints (the constraint for $d = 3$ is sufficient as all other constraints are weaker).

Alternatively, set $x := 2^\alpha$, compute the unique positive real root of each of the characteristic polynomials

$$c_d(x) := x^{-1} + x^{-1-d} - 1,$$

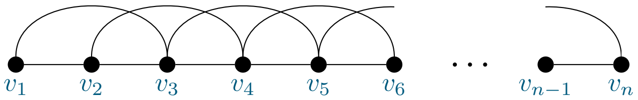
and take the maximum of these roots (Kullmann, 1999).

d	x	α
3	1.3803	0.4650
4	1.3248	0.4057
5	1.2852	0.3620
6	1.2555	0.3282
7	1.2321	0.3011

Simple Analysis: Result

- use the Simple Analysis Lemma with $c = 2$ and $\alpha = 0.464959$
- running time of Algorithm **mis** upper bounded by $O(n^3) \cdot 2^{0.464959 \cdot n} = O(2^{0.4650 \cdot n})$ or $O(1.3803^n)$

Lower bound



$$T(n) = T(n-5) + T(n-3)$$

- for this graph, P_n^2, the worst case running time is $1.1938\ldots^n \cdot \mathsf{poly}(n)$
- Run time of algo **mis** is $\Omega(1.1938^n)$

Worst-case running time — a mystery

Mystery

What is the worst-case running time of Algorithm **mis**?

- lower bound $\Omega(1.1938^n)$
- upper bound $O(1.3803^n)$

Outline

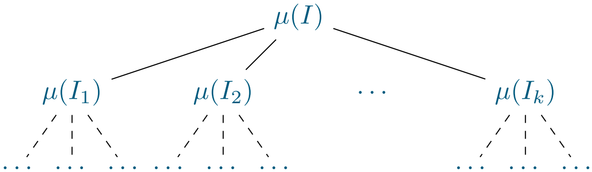
1 Introduction

2 Maximum Independent Set
- Simple Analysis
- Search Trees and Branching Numbers
- Measure & Conquer Analysis
- Optimizing the measure
- Exponential Time Subroutines
- Structures that arise rarely

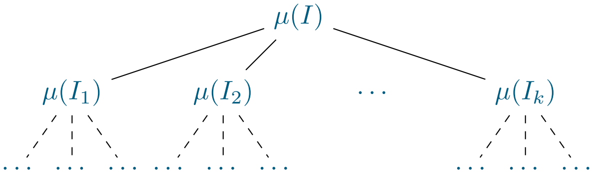
3 Further Reading

Search Trees

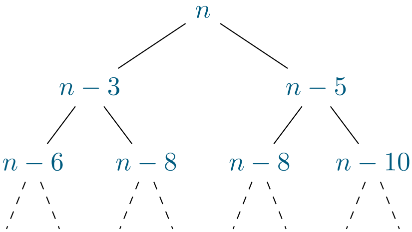
Denote $\mu(I) := \alpha \cdot |I|$.



Denote $\mu(I) := \alpha \cdot |I|$.



Example: execution of **mis** on a P_n^2



Branching number: Definition

Consider a constraint

$$2^{\mu(I)-a_1} + \cdots + 2^{\mu(I)-a_k} \leq 2^{\mu(I)}.$$

Its branching number is

$$2^{-a_1} + \cdots + 2^{-a_k},$$

and is denoted by

$$(a_1, \ldots, a_k).$$

Clearly, any constraint with branching number at most 1 is satisfied.

Branching numbers: Properties

Dominance For any a_i, b_i such that $a_i \geq b_i$ for all $i,\ 1 \leq i \leq k$,

$$(a_1, \ldots, a_k) \leq (b_1, \ldots, b_k)\,,$$

as $2^{-a_1} + \cdots + 2^{-a_k} \leq 2^{-b_1} + \cdots + 2^{-b_k}$.
In particular, for any $a, b > 0$,

$$\text{either} \quad (a, a) \leq (a, b) \quad \text{or} \quad (b, b) \leq (a, b)\,.$$

Balance If $0 < a \leq b$, then for any ε such that $0 \leq \varepsilon \leq a$,

$$(a, b) \leq (a - \varepsilon, b + \varepsilon)$$

by convexity of 2^x.

Outline

Measure & Conquer analysis

- Goal
  - capture more structural changes when branching into subinstances
- How?
  - via a potential-function method called Measure & Conquer (Fomin, Grandoni, and Kratsch, 2009)
- Example: Algorithm **mis**
  - advantage when degrees of vertices decrease

Measure

Instead of using the number of vertices, n, to track the progress of **mis**, let us use a measure μ of G.

Definition 6

A measure μ for a problem P is a function from the set of all instances for P to the set of non negative reals.

Let us use the following measure for the analysis of **mis** on graphs of maximum degree at most 5:

$$\mu(G) = \sum_{i=0}^{5} \omega_i n_i,$$

where $n_i := |\{v \in V : d(v) = i\}|$.

Measure & Conquer Analysis

Lemma 7 (Measure & Conquer Lemma)

Let

- A *be a branching algorithm*
- $c \geq 0$ *be a constant, and*
- $\mu(\cdot), \eta(\cdot)$ *be two measures for the instances of* A,

such that on input I, A *calls itself recursively on instances* $I_1, \ldots, I_k$, *but, besides the recursive calls, uses time* $O(\eta(I)^c)$, *such that*

$$(\forall i) \quad \eta(I_i) \leq \eta(I) - 1, \text{ and} \tag{6}$$

$$2^{\mu(I_1)} + \ldots + 2^{\mu(I_k)} \leq 2^{\mu(I)}. \tag{7}$$

Then A *solves any instance* I *in time* $O(\eta(I)^{c+1}) \cdot 2^{\mu(I)}$.

Analysis of mis for degree at most 5

For $\mu(G) = \sum_{i=0}^{5} \omega_i n_i$ to be a valid measure, we constrain that

$$w_d \geq 0 \qquad \text{for each } d \in \{0, \ldots, 5\}$$

We also constrain that reducing the degree of a vertex does not increase the measure (useful for analysis of the degree-1 simplification rule and the branching rule):

$$-\omega_d + \omega_{d-1} \leq 0 \qquad \text{for each } d \in \{1, \ldots, 5\}$$

Analysis of mis for degree at most 5

For $\mu(G) = \sum_{i=0}^{5} \omega_i n_i$ to be a valid measure, we constrain that

$$w_d \geq 0 \qquad\qquad \text{for each } d \in \{0, \ldots, 5\}$$

We also constrain that reducing the degree of a vertex does not increase the measure (useful for analysis of the degree-1 simplification rule and the branching rule):

$$-\omega_d + \omega_{d-1} \leq 0 \qquad\qquad \text{for each } d \in \{1, \ldots, 5\}$$

Lines 1–2 is a halting rule and we merely need that it takes polynomial time so that we can apply Lemma 7.

if $\Delta(G) \leq 2$ **then** // G has max degree ≤ 2
 return the size of a maximum i.s. of G in polynomial time

Analysis of mis for degree at most 5 (II)

Lines 3–4 of **mis** need to satisfy (7).

else if $\exists v \in V : d(v) = 1$ **then** // v has degree 1
  return $1 + \textbf{mis}(G - N[v])$

The simplification rule removes v and its neighbor u.
We get a constraint for each possible degree of u:

$$
\begin{aligned}
& & 2^{\mu(G)-\omega_1-\omega_d} &\leq 2^{\mu(G)} & &\text{for each } d \in \{1,\dots,5\}\\
&\Leftrightarrow & 2^{-\omega_1-\omega_d} &\leq 2^{0} & &\text{for each } d \in \{1,\dots,5\}\\
&\Leftrightarrow & -\omega_1-\omega_d &\leq 0 & &\text{for each } d \in \{1,\dots,5\}
\end{aligned}
$$

These constraints are always satisfied since $\omega_d \geq 0$ for each $d \in \{0,\dots,5\}$.
Note: the degrees of u's other neighbors (if any) decrease, but this degree change does not increase the measure.

Analysis of mis for degree at most 5 (III)

For lines 5–7 of **mis** we consider two cases.

else if G is not connected **then**

  Let G_1 be a connected component of G

  return $\textbf{mis}(G_1) + \textbf{mis}(G - V(G_1))$

If $\mu(G_1) < 1$ (or $\mu(G - V(G_1)) < 1$, which is handled similarly), then we view this rule as a simplification rule, which takes polynomial time to compute $\textbf{mis}(G_1)$, and then makes a recursive call $\textbf{mis}(G - V(G_1))$. To ensure that instances with measure < 1 can be solved in polynomial time, we constrain that

$$w_d > 0 \qquad\qquad \text{for each } d \in \{3, 4, 5\}$$

and this will be implied by other constraints.

Otherwise, $\mu(G_1) \geq 1$ and $\mu(G - V(G_1)) \geq 1$, and we need to satisfy (7).

Since $\mu(G) = \mu(G_1) + \mu(G - V(G_1))$, the constraints

$$2^{\mu(G_1)} + 2^{\mu(G - V(G_1))} \leq 2^{\mu(G)}$$

are always satisfied since the slope of the function 2^x is at least 1 when $x \geq 1$.

(I.e., we get no new constraints on $\omega_1, \ldots, \omega_5$.)

Analysis of mis for degree at most 5 (IV)

Lines 8–10 of **mis** need to satisfy (7).

else

Select $v \in V$ s.t. $d(v) = \Delta(G)$ `// v has max degree`

return $\max\left(1 + \textbf{mis}(G - N[v]), \textbf{mis}(G - v)\right)$

We know that in $G - N[v]$, some vertex of $N^2[v]$ has its degree decreased (unless G has at most 6 vertices, which can be solved in constant time). Define

$$(\forall d : 2 \leq d \leq 5) \quad h_d := \min_{2 \leq i \leq d} \{w_i - w_{i-1}\}$$

We obtain the following constraints:

$$2^{\mu(G) - w_d - \sum_{i=2}^{d} p_i \cdot (w_i - w_{i-1})} + 2^{\mu(G) - w_d - \sum_{i=2}^{d} p_i \cdot w_i - h_d} \leq 2^{\mu(G)}$$

$$\Leftrightarrow \quad 2^{- w_d - \sum_{i=2}^{d} p_i \cdot (w_i - w_{i-1})} + 2^{- w_d - \sum_{i=2}^{d} p_i \cdot w_i - h_d} \leq 1$$

for all $d, 3 \leq d \leq 5$ (degree of v), and all $p_i, 2 \leq i \leq d,$ such that $\sum_{i=2}^{d} p_i = d$ (number of neighbors of degree i).

Applying the lemma

Our constraints

$$\begin{aligned} w_d &\geq 0 \\ -\omega_d + \omega_{d-1} &\leq 0 \\ 2^{-w_d - \sum_{i=2}^{d} p_i \cdot (w_i - w_{i-1})} + 2^{-w_d - \sum_{i=2}^{d} p_i \cdot w_i - h_d} &\leq 1 \end{aligned}$$

are satisfied by the following values:

Applying the lemma

Our constraints

$$\begin{aligned} w_d &\geq 0 \\ -\omega_d + \omega_{d-1} &\leq 0 \\ 2^{-w_d - \sum_{i=2}^{d} p_i \cdot (w_i - w_{i-1})} + 2^{-w_d - \sum_{i=2}^{d} p_i \cdot w_i - h_d} &\leq 1 \end{aligned}$$

are satisfied by the following values:

i	w_i	h_i
1	0	0
2	0.25	0.25
3	0.35	0.10
4	0.38	0.03
5	0.40	0.02

These values for w_i satisfy all the constraints and $\mu(G) \leq 2n/5$ for any graph of max degree ≤ 5.
Taking $c = 2$ and $\eta(G) = n$, the Measure & Conquer Lemma shows that **mis** has run time $O(n^3)2^{2n/5} = O(1.3196^n)$ on graphs of max degree ≤ 5.

Outline

Compute optimal weights

- By convex programming (Gaspers and Sorkin, 2012)

All constraints are already convex, except conditions for h_d

$$(\forall d : 2 \le d \le 5) \quad h_d := \min_{2 \le i \le d} \{w_i - w_{i-1}\}$$

$$\Downarrow$$

$$(\forall i, d : 2 \le i \le d \le 5) \quad h_d \le w_i - w_{i-1}.$$

Use existing convex programming solvers to find optimum weights.

Convex program in AMPL

```
param maxd integer = 5;
set DEGREES := 0..maxd;
var W {DEGREES} >= 0;  # weight for vertices according to their degrees
var g {DEGREES} >= 0;  # weight for degree reductions from deg i
var h {DEGREES} >= 0;  # weight for degree reductions from deg <= i
var Wmax;              # maximum weight of W[d]

minimize Obj: Wmax;    # minimize the maximum weight

subject to MaxWeight {d in DEGREES}:
  Wmax >= W[d];
subject to gNotation {d in DEGREES : 2 <= d}:
  g[d] <= W[d]-W[d-1];
subject to hNotation {d in DEGREES, i in DEGREES : 2 <= i <= d}:
  h[d] <= W[i]-W[i-1];
subject to Deg3 {p2 in 0..3, p3 in 0..3 : p2+p3=3}:
  2^(-W[3] -p2*g[2] -p3*g[3]) + 2^(-W[3] -p2*W[2] -p3*W[3] -h[3]) <=1;
subject to Deg4 {p2 in 0..4, p3 in 0..4, p4 in 0..4 : p2+p3+p4=4}:
  2^(-W[4] - p2*g[2] - p3*g[3] - p4*g[4])
+ 2^(-W[4] - p2*W[2] - p3*W[3] - p4*W[4] - h[4]) <=1;
subject to Deg5 {p2 in 0..5, p3 in 0..5, p4 in 0..5, p5 in 0..5 :
                 p2+p3+p4+p5=5}:
  2^(-W[5] - p2*g[2] - p3*g[3] - p4*g[4] - p5*g[5])
+ 2^(-W[5] - p2*W[2] - p3*W[3] - p4*W[4] - p5*W[5] - h[5]) <=1;
```

Convex program in Python I

```
import pyomo.environ as pyo # install with > pip install pyomo

maxd = 5                  # maximum vertex degree
degrees = range(0,maxd+1) # set of all possible degrees
m = pyo.ConcreteModel()   # model to be solved

# declare variables
m.W    = pyo.Var(degrees, domain=pyo.NonNegativeReals)
m.Wmax = pyo.Var(domain=pyo.NonNegativeReals)
m.g    = pyo.Var(degrees, domain=pyo.NonNegativeReals)
m.h    = pyo.Var(degrees, domain=pyo.NonNegativeReals)

# set objective function
m.OBJ = pyo.Objective(expr = m.Wmax, sense=pyo.minimize)

# add constraints
def maxweight_rule(m, d):
  return m.Wmax >= m.W[d]
m.maxweight = pyo.Constraint(degrees, rule=maxweight_rule)

def gnotation_rule(m, d):
  return m.g[d] <= m.W[d]-m.W[d-1]
m.gnotation = pyo.Constraint(range(2,maxd+1), rule=gnotation_rule)

def hnotation_rule(m, i, d):
  return m.h[d] <= m.W[i]-m.W[i-1]
```

Convex program in Python II

```
m.hnotation = pyo.Constraint(((i,d) for i in range(2,maxd+1) \
                                    for d in range(2,maxd+1) \
                              if i<=d), rule=hnotation_rule)

def deg3_rule(m, p2, p3):
  return 2**(-m.W[3] -p2*m.g[2] -p3*m.g[3]) \
       + 2**(-m.W[3] -p2*m.W[2] -p3*m.W[3] -m.h[3]) \
       <= 1
m.deg3 = pyo.Constraint(((p2,p3) for p2 in range(0,4) \
                                 for p3 in range(0,4) \
                         if p2+p3==3), rule=deg3_rule)

def deg4_rule(m, p2, p3, p4):
  return 2**(-m.W[4] -p2*m.g[2] -p3*m.g[3] -p4*m.g[4]) \
       + 2**(-m.W[4] -p2*m.W[2] -p3*m.W[3] -p4*m.W[4] -m.h[4]) \
       <= 1
m.deg4 = pyo.Constraint(((p2,p3,p4) for p2 in range(0,5) \
                                    for p3 in range(0,5) \
                                    for p4 in range(0,5) \
                         if p2+p3+p4==4), rule=deg4_rule)

def deg5_rule(m, p2, p3, p4, p5):
  return 2**(-m.W[5] -p2*m.g[2] -p3*m.g[3] -p4*m.g[4] -p5*m.g[5]) \
       + 2**(-m.W[5] -p2*m.W[2] -p3*m.W[3] -p4*m.W[4] -p5*m.W[5] -m.h[5]) \
       <= 1
m.deg5 = pyo.Constraint(((p2,p3,p4,p5) for p2 in range(0,6) \
```

Convex program in Python III

```
                                        for p3 in range(0,6) \
                                        for p4 in range(0,6) \
                                        for p5 in range(0,6) \
                          if p2+p3+p4+p5==5), rule=deg5_rule)

# set up the solver
solver_manager = pyo.SolverManagerFactory('neos') # we are using a remote server here
solver = pyo.SolverFactory('ipopt')               # with the solver ipopt
results = solver_manager.solve(m, opt=solver)     # solve
results.write()                                   # display results
print("Running time: ", 2**m.Wmax.value, "^n")    # display final running time
m.display()                                       # display details
```

Optimal weights

i	w_i	h_i
1	0	0
2	0.206018	0.206018
3	0.324109	0.118091
4	0.356007	0.031898
5	0.358044	0.002037

- use the Measure & Conquer Lemma with $\mu(G) = \sum_{i=1}^{5} w_i n_i \leq 0.358044 \cdot n$, $c = 2$, and $\eta(G) = n$
- **mis** has running time $O(n^3) 2^{0.358044 \cdot n} = O(1.2817^n)$

Outline

Exponential time subroutines

Lemma 8 (Combine Analysis Lemma)

Let

- A *be a branching algorithm and* B *be an algorithm,*
- $c \geq 0$ *be a constant, and*
- $\mu(\cdot), \mu'(\cdot), \eta(\cdot)$ *be three measures for the instances of* A *and* B*,*

such that $\mu'(I) \leq \mu(I)$ *for all instances* I*, and on input* I*,* A *either solves* I *by invoking* B *with running time* $O(\eta(I)^{c+1}) \cdot 2^{\mu'(I)}$*, or calls itself recursively on instances* $I_1, \ldots, I_k$*, but, besides the recursive calls, uses time* $O(\eta(I)^c)$*, such that*

$$(\forall i) \quad \eta(I_i) \leq \eta(I) - 1, \text{ and} \tag{8}$$

$$2^{\mu(I_1)} + \ldots + 2^{\mu(I_k)} \leq 2^{\mu(I)}. \tag{9}$$

Then A *solves any instance* I *in time* $O(\eta(I)^{c+1}) \cdot 2^{\mu(I)}$*.*

Algorithm **mis** on general graphs

- use the Combine Analysis Lemma with $A = B =$ **mis**, $c = 2$, $\mu(G) = 0.35805n$, $\mu'(G) = \sum_{i=1}^{5} w_i n_i$, and $\eta(G) = n$
- for every instance G, $\mu'(G) \leq \mu(G)$ because $\forall i, w_i \leq 0.35805$
- for each $d \geq 6$,

$$(0.35805, (d+1) \cdot 0.35805) \leq 1$$

- Thus, Algorithm **mis** has running time $O(1.2817^n)$ for graphs of arbitrary degrees

Outline

Rare Configurations

- Branching on a local configuration C does not influence overall running time if C is selected only a constant number of times on the path from the root to a leaf of any search tree corresponding to the execution of the algorithm
- Can be proved formally by using measure

$$\mu'(I) := \begin{cases} \mu(I) + c & \text{if } C \text{ may be selected in the current subtree} \\ \mu(I) & \text{otherwise.} \end{cases}$$

Avoid branching on regular instances in **mis**

else

Select $v \in V$ such that

(1) v has maximum degree, and

(2) among all vertices satisfying (1), v has a neighbor of minimum degree

return $\max(1 + \textbf{mis}(G - N[v]), \textbf{mis}(G - v))$

New measure:

$$\mu'(G) = \mu(G) + \sum_{d=3}^{5} [G \text{ has a } d\text{-regular subgraph}] \cdot C_d$$

where $C_d, 3 \leq d \leq 5,$ are constants.

The Iverson bracket $[F] = \begin{cases} 1 \text{ if } F \text{ true} \\ 0 \text{ otherwise} \end{cases}$

Resulting Branching numbers

For each $d, 3 \leq d \leq 5$ and all $p_i, 2 \leq i \leq d$ such that $\sum_{i=2}^{d} p_i = d$ and $p_d \neq d$,

$$\left(w_d + \sum_{i=2}^{d} p_i \cdot (w_i - w_{i-1}), w_d + \sum_{i=2}^{d} p_i \cdot w_i + h_d\right).$$

All these branching numbers are at most 1 with the optimal set of weights on the next slide

Result

i	w_i	h_i
1	0	0
2	0.207137	0.207137
3	0.322203	0.115066
4	0.343587	0.021384
5	0.347974	0.004387

Thus, the modified Algorithm **mis** has running time $O(2^{0.3480 \cdot n}) = O(1.2728^n)$.

Outline

Further Reading

- Chapter 2, *Branching* in (Fomin and Kratsch, 2010)
- Chapter 6, *Measure & Conquer* in (Fomin and Kratsch, 2010)
- Chapter 2, *Branching Algorithms* in (Gaspers, 2010)

References I

Fedor V. Fomin, Fabrizio Grandoni, and Dieter Kratsch (2009). "A measure & conquer approach for the analysis of exact algorithms". In: *Journal of the ACM* 56.5, 25:1–25:32.

Fedor V. Fomin and Dieter Kratsch (2010). *Exact Exponential Algorithms*. Springer. DOI: 10.1007/978-3-642-16533-7.

Serge Gaspers (2010). *Exponential Time Algorithms: Structures, Measures, and Bounds*. VDM Verlag Dr. Mueller.

Serge Gaspers and Gregory B. Sorkin (2012). "A universally fastest algorithm for Max 2-Sat, Max 2-CSP, and everything in between". In: *Journal of Computer and System Sciences* 78.1, pp. 305–335.

Oliver Kullmann (1999). "New Methods for 3-SAT Decision and Worst-case Analysis". In: *Theoretical Computer Science* 223.1-2, pp. 1–72.

Randomized Algorithms

Serge Gaspers

UNSW

Outline

Outline

Randomized Algorithms

- Turing machines do not inherently have access to randomness.
- Assume algorithm has also access to a stream of random bits drawn uniformly at random.
- With r random bits, the probability space is the set of all 2^r possible strings of random bits (with uniform distribution).

Las Vegas algorithms

Definition 1

A Las Vegas algorithm is a randomized algorithm whose output is always correct.

Randomness is used to upper bound the expected running time of the algorithm.

Example

Quicksort with random choice of pivot.

Monte Carlo algorithms

Definition 2

- A Monte Carlo algorithm is an algorithm whose output is incorrect with probability at most p, $0 < p < 1$.
- A Monte Carlo has one sided error if its output is incorrect only on $\textsc{Yes}$-instances or on $\textsc{No}$-instances, but not both.
- A one-sided error Monte Carlo algorithm with false negatives answers $\textsc{No}$ for every $\textsc{No}$-instance, and answers $\textsc{Yes}$ on $\textsc{Yes}$-instances with probability $p \in (0, 1)$. We say that p is the *success probability* of the algorithm.

Algorithms with increased success probability

Boosting success probability

Suppose A is a one-sided Monte Carlo algorithm with false negatives with success probability p. How can we use A to design a new one-sided Monte Carlo algorithm with success probability $p^* > p$?

Algorithms with increased success probability

Boosting success probability

Suppose A is a one-sided Monte Carlo algorithm with false negatives with success probability p. How can we use A to design a new one-sided Monte Carlo algorithm with success probability $p^* > p$?

Let $t = -\frac{\ln(1-p^*)}{p}$ and run the algorithm t times. Return $\textsc{Yes}$ if at least one run of the algorithm returned $\textsc{Yes}$, and $\textsc{No}$ otherwise.

Algorithms with increased success probability

Boosting success probability

Suppose A is a one-sided Monte Carlo algorithm with false negatives with success probability p. How can we use A to design a new one-sided Monte Carlo algorithm with success probability $p^* > p$?

Let $t = -\frac{\ln(1-p^*)}{p}$ and run the algorithm t times. Return $\textsc{Yes}$ if at least one run of the algorithm returned $\textsc{Yes}$, and $\textsc{No}$ otherwise. Failure probability is

$$(1-p)^t \leq (e^{-p})^t = e^{-p \cdot t} = e^{\ln(1-p^*)} = 1 - p^*$$

via the inequality $1 - x \leq e^{-x}$.

Algorithms with increased success probability

Boosting success probability

Suppose A is a one-sided Monte Carlo algorithm with false negatives with success probability p. How can we use A to design a new one-sided Monte Carlo algorithm with success probability $p^* > p$?

Let $t = -\frac{\ln(1-p^*)}{p}$ and run the algorithm t times. Return YES if at least one run of the algorithm returned YES, and NO otherwise. Failure probability is

$$(1-p)^t \leq (e^{-p})^t = e^{-p \cdot t} = e^{\ln(1-p^*)} = 1 - p^*$$

via the inequality $1 - x \leq e^{-x}$.

Definition 3

A randomized algorithm is a one-sided Monte Carlo algorithm with constant success probability.

Amplification

Theorem 4

If a one-sided error Monte Carlo algorithm has success probability at least p, then repeating it independently $\lceil \frac{1}{p} \rceil$ times gives constant success probability.

In particular if we have a polynomial-time one-sided error Monte Carlo algorithm with success probability $p = \frac{1}{f(k)}$ for some computable function f, then we get a randomized FPT algorithm with running time $O^*(f(k))$.

Outline

Vertex Cover

For a graph $G = (V, E)$ a vertex cover $X \subseteq V$ is a set of vertices such that every edge is adjacent to a vertex in X.

VERTEX COVER	
Input:	Graph G, integer k
Parameter:	k
Question:	Does G have a vertex cover of size k?

Vertex Cover

For a graph $G = (V, E)$ a vertex cover $X \subseteq V$ is a set of vertices such that every edge is adjacent to a vertex in X.

VERTEX COVER	
Input:	Graph G, integer k
Parameter:	k
Question:	Does G have a vertex cover of size k?

Warm-up: design a randomized algorithm with running time $O^*(2^k)$.

Randomized Algorithm for Vertex Cover

Algorithm rvc$(G = (V, E), k)$

$S \leftarrow \emptyset$

while $k > 0$ and $E \neq \emptyset$ **do**

  Select an edge $uv \in E$ uniformly at random

  Select an endpoint $w \in \{u, v\}$ uniformly at random

  $S \leftarrow S \cup \{w\}$

  $G \leftarrow G - w$

  $k \leftarrow k - 1$

if S is a vertex cover of G **then**

  return $\textsc{Yes}$

else

  return $\textsc{No}$

Success probability

- Let C be a minimal (inclusion-wise minimal) vertex cover of G of size $k' \leq k$
- What is the probability that Algorithm rvc returns C?
- When it selects an edge $uv \in E$, we have that $\{u, v\} \cap C \neq \emptyset$
- When it selects a random endpoint $w \in \{u, v\}$, we have that $w \in C$ with probability $\geq 1/2$
- It finds C with probability at least $1/2^{k'}$

Randomized Algorithm for Vertex Cover

Theorem 5

VERTEX COVER *has a randomized algorithm with running time* $O^*(2^k)$.

Proof.

- If G has vertex cover number at most k, then Algorithm rvc finds one with probability at least $\frac{1}{2^k}$.
- Applying Theorem 4 gives a randomized FPT running time of $O^*(2^k)$.

$\square$

Outline

Feedback Vertex Set

A feedback vertex set of a multigraph $G = (V, E)$ is a set of vertices $S \subset V$ such that $G - S$ is acyclic.

FEEDBACK VERTEX SET	
Input:	Multigraph G, integer k
Parameter:	k
Question:	Does G have a feedback vertex of size k?

Feedback Vertex Set

A feedback vertex set of a multigraph $G = (V, E)$ is a set of vertices $S \subset V$ such that $G - S$ is acyclic.

FEEDBACK VERTEX SET	
Input:	Multigraph G, integer k
Parameter:	k
Question:	Does G have a feedback vertex of size k?

Recall the following simplification rules for FEEDBACK VERTEX SET.

Simplification Rules

1. Loop: If loop at vertex v, remove v and decrease k by 1
2. Multiedge: Reduce the multiplicity of each edge with multiplicity ≥ 3 to 2.
3. Degree-1: If v has degree at most 1 then remove v.
4. Degree-2: If v is incident to exactly two edges uv, vw, then delete these 2 edges uv, vw and add a new edge uw.

The solution is incident to a constant fraction of the edges

Lemma 6

Let G be a multigraph with minimum degree at least 3. Then, for every feedback vertex set X of G, at least $1/3$ of the edges have at least one endpoint in X.

The solution is incident to a constant fraction of the edges

Lemma 6

Let G be a multigraph with minimum degree at least 3. Then, for every feedback vertex set X of G, at least $1/3$ of the edges have at least one endpoint in X.

Proof.

Denote by n and m the number of vertices and edges of G, respectively.
Since $\delta(G) \geq 3$, we have that $m \geq 3n/2$.
Let $F := G - X$.
Since F has at most $n-1$ edges, at least $\frac{1}{3}$ of the edges have an endpoint in X. □

Randomized Algorithm

Theorem 7

FEEDBACK VERTEX SET *has a randomized algorithm with running time* $O^*(6^k)$.

Randomized Algorithm

Theorem 7

FEEDBACK VERTEX SET *has a randomized algorithm with running time* $O^*(6^k)$.

We prove the theorem using the following algorithm.

- $S \leftarrow \emptyset$
- Do k times: Apply simplification rules; add a random endpoint of a random edge to S.
- If S is a feedback vertex set, return YES, otherwise return NO.

Proof

Proof.

- We need to show: each time the algorithm adds a vertex v to S, if $(G - S, k - |S|)$ is a Yes-instance, then with probability at least $1/6$, the instance $(G - (S \cup \{v\}), k - |S| - 1)$ is also a Yes-instance. Then, by induction, we can conclude that with probability $1/(6^k)$, the algorithm finds a feedback vertex set of size at most k if it is given a Yes-instance.

Proof

Proof.

- We need to show: each time the algorithm adds a vertex v to S, if $(G - S, k - |S|)$ is a Yes-instance, then with probability at least $1/6$, the instance $(G - (S \cup \{v\}), k - |S| - 1)$ is also a Yes-instance. Then, by induction, we can conclude that with probability $1/(6^k)$, the algorithm finds a feedback vertex set of size at most k if it is given a Yes-instance.
- Assume $(G - S, k - |S|)$ is a Yes-instance.
- Lemma 6 implies that with probability at least $1/3$, a randomly chosen edge uv has at least one endpoint in some feedback vertex set of size $k - |S|$.
- So, with probability at least $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$, a randomly chosen endpoint of uv belongs some feedback vertex set of size $\leq k - |S|$.

Proof

Proof.

- We need to show: each time the algorithm adds a vertex v to S, if $(G - S, k - |S|)$ is a $\textsc{Yes}$-instance, then with probability at least $1/6$, the instance $(G - (S \cup \{v\}), k - |S| - 1)$ is also a $\textsc{Yes}$-instance. Then, by induction, we can conclude that with probability $1/(6^k)$, the algorithm finds a feedback vertex set of size at most k if it is given a $\textsc{Yes}$-instance.
- Assume $(G - S, k - |S|)$ is a $\textsc{Yes}$-instance.
- Lemma 6 implies that with probability at least $1/3$, a randomly chosen edge uv has at least one endpoint in some feedback vertex set of size $k - |S|$.
- So, with probability at least $\frac{1}{2} \cdot \frac{1}{3} = \frac{1}{6}$, a randomly chosen endpoint of uv belongs some feedback vertex set of size $\leq k - |S|$.
- Applying Theorem 4 gives a randomized FPT running time of $O^*(6^k)$.

□

Improved analysis

Lemma 8

Let G be a multigraph with minimum degree at least 3. For every feedback vertex set X, at least $1/2$ of the edges of G have at least one endpoint in X.

Improved analysis

Lemma 8

Let G be a multigraph with minimum degree at least 3. For every feedback vertex set X, at least $1/2$ of the edges of G have at least one endpoint in X.

Note: For a feedback vertex set X, consider the forest $F := G - X$. The statement is equivalent to:

$$|E(G) \setminus E(F)| \geq |E(F)|$$

Let $J \subseteq E(G)$ denote the edges with one endpoint in X, and the other in $V(F)$. We will show the stronger result:

$$|J| \geq |V(F)|$$

Improved analysis

Proof.

- Let $V_{\leq 1}, V_2, V_{\geq 3}$ be the set of vertices that have degree at most 1, exactly 2, and at least 3, respectively, in F.

Improved analysis

Proof.

- Let $V_{\leq 1}, V_2, V_{\geq 3}$ be the set of vertices that have degree at most 1, exactly 2, and at least 3, respectively, in F.
- Since $\delta(G) \geq 3$, each vertex in $V_{\leq 1}$ contributes at least 2 edges to J, and each vertex in V_2 contributes at least 1 edge to J.

Improved analysis

Proof.

- Let $V_{\leq 1}, V_2, V_{\geq 3}$ be the set of vertices that have degree at most 1, exactly 2, and at least 3, respectively, in F.
- Since $\delta(G) \geq 3$, each vertex in $V_{\leq 1}$ contributes at least 2 edges to J, and each vertex in V_2 contributes at least 1 edge to J.
- We show that $|V_{\geq 3}| \leq |V_{\leq 1}|$ by induction on $|V(F)|$.
  - Trivially true for forests with at most 1 vertex.
  - Assume true for forests with at most $n-1$ vertices.
  - For any forest on n vertices, consider removing a leaf (which must always exist) to obtain F' with the vertex partition $(V'_{\leq 1}, V'_2, V'_{\geq 3})$.
    If $|V_{\geq 3}| = |V'_{\geq 3}|$, then we have that $|V_{\geq 3}| = |V'_{\geq 3}| \leq |V'_{\leq 1}| \leq |V_{\leq 1}|$.
    Otherwise, $|V_{\geq 3}| = |V'_{\geq 3}| + 1 \leq |V'_{\leq 1}| + 1 = |V_{\leq 1}|$.

Improved analysis

Proof.

- Let $V_{\leq 1}, V_2, V_{\geq 3}$ be the set of vertices that have degree at most 1, exactly 2, and at least 3, respectively, in F.
- Since $\delta(G) \geq 3$, each vertex in $V_{\leq 1}$ contributes at least 2 edges to J, and each vertex in V_2 contributes at least 1 edge to J.
- We show that $|V_{\geq 3}| \leq |V_{\leq 1}|$ by induction on $|V(F)|$.
  - Trivially true for forests with at most 1 vertex.
  - Assume true for forests with at most $n-1$ vertices.
  - For any forest on n vertices, consider removing a leaf (which must always exist) to obtain F' with the vertex partition $(V'_{\leq 1}, V'_2, V'_{\geq 3})$.
    If $|V_{\geq 3}| = |V'_{\geq 3}|$, then we have that $|V_{\geq 3}| = |V'_{\geq 3}| \leq |V'_{\leq 1}| \leq |V_{\leq 1}|$.
    Otherwise, $|V_{\geq 3}| = |V'_{\geq 3}| + 1 \leq |V'_{\leq 1}| + 1 = |V_{\leq 1}|$.
- We conclude that:

$$|E(G) \setminus E(F)| \geq |J| \geq 2|V_{\leq 1}| + |V_2| \geq |V_{\leq 1}| + |V_2| + |V_{\geq 3}| = |V(F)|$$

□

Improved Randomized Algorithm

Theorem 9

FEEDBACK VERTEX SET *has a randomized algorithm with running time* $O^*(4^k)$.

Note

This algorithmic method is applicable whenever the vertex set we seek is incident to a constant fraction of the edges.

Outline

Longest Path

LONGEST PATH	
Input:	Graph G, integer k
Parameter:	k
Question:	Does G have a path on k vertices as a subgraph?

Longest Path

LONGEST PATH

Input:	Graph G, integer k
Parameter:	k
Question:	Does G have a path on k vertices as a subgraph?

NP-complete

To show that LONGEST PATH is NP-hard, reduce from HAMILTONIAN PATH by setting $k = n$ and leaving the graph unchanged.

Color Coding

Notation: $[k] = \{1, 2, \ldots, k\}$

Lemma 10

Let U be a set of size n, and let $X \subseteq U$ be a subset of size k. Let $\chi : U \to [k]$ be a coloring of the elements of U, chosen uniformly at random. Then the probability that the elements of X are colored with pairwise distinct colors is at least e^{-k}.

Color Coding

Notation: $[k] = \{1, 2, \dots, k\}$

Lemma 10

Let U be a set of size n, and let $X \subseteq U$ be a subset of size k. Let $\chi : U \to [k]$ be a coloring of the elements of U, chosen uniformly at random. Then the probability that the elements of X are colored with pairwise distinct colors is at least e^{-k}.

Proof.

There are k^n possible colorings χ and $k!k^{n-k}$ of them are injective on X. Using the inequality

$$k! > (k/e)^k,$$

the lemma follows since

$$\frac{k! \cdot k^{n-k}}{k^n} > \frac{k^k \cdot k^{n-k}}{e^k \cdot k^n} = e^{-k}.$$

□

Colorful Path

A path is colorful if all vertices of the path are colored with pairwise distinct colors.

Lemma 11

Let G be an undirected graph, and let $\chi : V(G) \to [k]$ be a coloring of its vertices with k colors. There is an algorithm that checks in time $O^*(2^k)$ whether G contains a colorful path on k vertices.

Colorful Path II

Proof.

Partition $V(G)$ into $V_1, ..., V_k$ subsets such that vertices in V_i are colored i.

Colorful Path II

Proof.

Partition $V(G)$ into $V_1, ..., V_k$ subsets such that vertices in V_i are colored i.
Apply dynamic programming on nonempty $S \subseteq \{1, ..., k\}$. For $u \in \bigcup_{i \in S} V_i$ let $P(S, u) = 1$ if there is a colorful path with colors from S and u as an endpoint.

Colorful Path II

Proof.

Partition $V(G)$ into $V_1, ..., V_k$ subsets such that vertices in V_i are colored i. Apply dynamic programming on nonempty $S \subseteq \{1, ..., k\}$. For $u \in \bigcup_{i \in S} V_i$ let $P(S, u) = 1$ if there is a colorful path with colors from S and u as an endpoint. We have the following:

- For $|S| = 1$, $P(S, u) = 1$ for $u \in V(G)$ iff $S = \{\chi(u)\}$.
- For $|S| > 1$

$$P(S,u) = \begin{cases} \bigvee_{uv \in E(G)} P(S \setminus \{\chi(u)\}, v) & \text{if } \chi(u) \in S \\ 0 & \text{otherwise} \end{cases}$$

Colorful Path II

Proof.

Partition $V(G)$ into $V_1, ..., V_k$ subsets such that vertices in V_i are colored i. Apply dynamic programming on nonempty $S \subseteq \{1, ..., k\}$. For $u \in \bigcup_{i \in S} V_i$ let $P(S, u) = 1$ if there is a colorful path with colors from S and u as an endpoint. We have the following:

- For $|S| = 1$, $P(S, u) = 1$ for $u \in V(G)$ iff $S = \{\chi(u)\}$.
- For $|S| > 1$

$$P(S, u) = \begin{cases} \bigvee_{uv \in E(G)} P(S \backslash \{\chi(u)\}, v) & \text{if } \chi(u) \in S \\ 0 & \text{otherwise} \end{cases}$$

All values of P can be computed in $O^*(2^k)$ time and there exists a colorful k-path iff $P([k], v) = 1$ for some vertex $v \in V(G)$. □

Longest Path

Theorem 12

Longest Path *has a randomized algorithm with running time* $O^*((2 \cdot e)^k)$.

Note

This algorithmic method is applicable whenever we seek a subgraph of size $f(k)$ with constant treewidth.

Outline

Exponential-time algorithms and parameterized algorithms

Exponential-time algorithms

- Algorithms for NP-hard problems
- Beat brute-force & improve
- Running time measured in the size of the universe n
- $O(2^n \cdot n)$, $O(1.5086^n)$, $O(1.0892^n)$

Parameterized algorithms

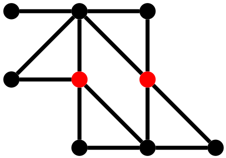
- Algorithms for NP-hard problems
- Use a parameter k (often k is the solution size)
- Algorithms with running time $f(k) \cdot n^c$
- $k^k n^{O(1)}$, $5^k n^{O(1)}$, $O(1.2738^k + kn)$

Can we use Parameterized algorithms to design fast Exponential-time algorithms?

Example: Feedback Vertex Set

$S \subseteq V$ is a feedback vertex set in a graph $G = (V, E)$ if $G - S$ is acyclic.

FEEDBACK VERTEX SET
- Input: Graph $G = (V, E)$, integer k
- Parameter: k
- Question: Does G have a f.v.s. of size at most k?

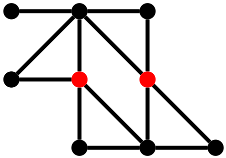


Example: Feedback Vertex Set

$S \subseteq V$ is a feedback vertex set in a graph $G = (V, E)$ if $G - S$ is acyclic.

FEEDBACK VERTEX SET

Input:	Graph $G = (V, E)$, integer k
Parameter:	k
Question:	Does G have a f.v.s. of size at most k?



Exponential-time algorithms

- $O^*(2^n)$ trivial
- $O(1.7548^n)$ (Fomin, Gaspers, Pyatkin, et al., 2008)
- $O(1.7347^n)$ (Fomin and Villanger, 2010)
- $O(1.7266^n)$ (Xiao and Nagamochi, 2015)

Parameterized algorithms

- $O^*((17k^4)!)$ (Bodlaender, 1994)
- $O^*((2k+1)^k)$ (Downey and Fellows, 1999)

$\vdots$

- $O^*(3.460^k)$ deterministic (Iwata and Kobayashi, 2019)
- $O^*(2.7^k)$ randomized (Li and

Exponential-time algorithms via parameterized algorithms

Binomial coefficients

$$\arg\max_{0\le k\le n} \binom{n}{k} = n/2 \qquad \text{and} \qquad \binom{n}{n/2} = \Theta(2^n/\sqrt{n})$$

Exponential-time algorithms via parameterized algorithms

Binomial coefficients

$$\arg\max_{0 \le k \le n} \binom{n}{k} = n/2 \qquad \text{and} \qquad \binom{n}{n/2} = \Theta(2^n/\sqrt{n})$$

Algorithm for Feedback Vertex Set

- Set $t = 0.6511 \cdot n$
- If $k \le t$, run $O^*(2.7^k)$ algorithm
- Else check all $\binom{n}{k}$ vertex subsets of size k

Running time: $O^*\left(\max\left(2.7^t, \binom{n}{t}\right)\right) = O^*(1.9093^n)$

Exponential-time algorithms via parameterized algorithms

Binomial coefficients

$$\arg\max_{0 \le k \le n} \binom{n}{k} = n/2 \quad \text{and} \quad \binom{n}{n/2} = \Theta(2^n/\sqrt{n})$$

Algorithm for Feedback Vertex Set

- Set $t = 0.6511 \cdot n$
- If $k \le t$, run $O^*(2.7^k)$ algorithm
- Else check all $\binom{n}{k}$ vertex subsets of size k

Running time: $O^*\left(\max\left(2.7^t, \binom{n}{t}\right)\right) = O^*(1.9093^n)$

This approach gives algorithms faster than $O^*(2^n)$ for subset problems with a parameterized algorithm faster than $O^*(4^k)$.

Subset Problems

An *implicit set system* is a function Φ with:

- Input: instance $I \in \{0,1\}^*$, $|I| = N$
- Output: set system $(U_I, \mathcal{F}_I)$:
  - universe U_I, $|U_I| = n$
  - family $\mathcal{F}_I$ of subsets of U_I

Subset Problems

An *implicit set system* is a function Φ with:

- Input: instance $I \in \{0,1\}^*$, $|I| = N$
- Output: set system $(U_I, \mathcal{F}_I)$:
  - universe U_I, $|U_I| = n$
  - family $\mathcal{F}_I$ of subsets of U_I

Φ-SUBSET

Input:	Instance I
Question:	Is $|\mathcal{F}_I| > 0$?

Subset Problems

An *implicit set system* is a function Φ with:

- Input: instance $I \in \{0,1\}^*$, $|I| = N$
- Output: set system $(U_I, \mathcal{F}_I)$:
  - universe U_I, $|U_I| = n$
  - family $\mathcal{F}_I$ of subsets of U_I

Φ-Subset

Input:	Instance I
Question:	Is $\|\mathcal{F}_I\| > 0$?

Φ-Extension

Input:	Instance I, a set $X \subseteq U_I$, and an integer k
Question:	Does there exist a subset $S \subseteq (U_I \setminus X)$ such that $S \cup X \in \mathcal{F}_I$ and $\|S\| \leq k$?

Algorithm

Suppose Φ-EXTENSION has a $O^*(c^k)$ time algorithm B.

Algorithm for checking whether $\mathcal{F}_I$ contains a set of size k

- Set $t = \max\left(0, \frac{ck-n}{c-1}\right)$
- Uniformly at random select a subset $X \subseteq U_I$ of size t
- Run $B(I, X, k-t)$

Algorithm

Suppose Φ-EXTENSION has a $O^*(c^k)$ time algorithm B.

Algorithm for checking whether $\mathcal{F}_I$ contains a set of size k

- Set $t = \max\left(0, \frac{ck-n}{c-1}\right)$
- Uniformly at random select a subset $X \subseteq U_I$ of size t
- Run $B(I, X, k-t)$

Running time: (Fomin, Gaspers, Lokshtanov, et al., 2019)

$$O^*\left(\frac{\binom{n}{t}}{\binom{k}{t}} \cdot c^{k-t}\right) = O^*\left(2 - \frac{1}{c}\right)^n$$

Intuition

Brute-force randomized algorithm

- Pick k elements of the universe one-by-one.
- Suppose $\mathcal{F}_I$ contains a set of size k.

Success probability:

$$\frac{k}{n} \cdot \frac{k-1}{n-1} \cdot \ldots \cdot \underbrace{\frac{k-t}{n-t}}_{\substack{\parallel \\ \frac{1}{c}}} \cdot \ldots \cdot \frac{2}{n-(k-2)} \frac{1}{n-(k-1)} = \frac{1}{\binom{n}{k}}$$

Randomized Monotone Local Search

Theorem 13 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

If there exists a (randomized) algorithm for Φ-EXTENSION *with running time* $O^*(c^k)$ *then there exists a randomized algorithm for* Φ-SUBSET *with running time* $(2 - \frac{1}{c})^n \cdot N^{O(1)}$.

Randomized Monotone Local Search

Theorem 13 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

If there exists a (randomized) algorithm for Φ-EXTENSION *with running time* $O^*(c^k)$ *then there exists a randomized algorithm for* Φ-SUBSET *with running time* $(2 - \frac{1}{c})^n \cdot N^{O(1)}$.

Theorem 14 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

FEEDBACK VERTEX SET *has a randomized algorithm with running time* $O^*\left(\left(2 - \frac{1}{2.7}\right)^n\right) \subseteq O(1.6297^n)$.

Derandomization

Derandomization at the expense of a subexponential factor in the running time.

Theorem 15 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

If there exists an algorithm for Φ-EXTENSION *with running time* $O^*(c^k)$ *then there exists an algorithm for* Φ-SUBSET *with running time* $(2-\frac{1}{c})^{n+o(n)} \cdot N^{O(1)}$.

Derandomization

Derandomization at the expense of a subexponential factor in the running time.

Theorem 15 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

If there exists an algorithm for Φ-EXTENSION *with running time* $O^*(c^k)$ *then there exists an algorithm for* Φ-SUBSET *with running time* $(2 - \frac{1}{c})^{n+o(n)} \cdot N^{O(1)}$.

Theorem 16 ((Fomin, Gaspers, Lokshtanov, et al., 2019))

FEEDBACK VERTEX SET *has an algorithm with running time* $O^*\left(\left(2 - \frac{1}{3.460}\right)^n\right) \subseteq O(1.7110^n)$.

Further Reading

- Chapter 5, *Randomized methods in parameterized algorithms* by (Cygan et al., 2015)
- *Exact Algorithms via Monotone Local Search* (Fomin, Gaspers, Lokshtanov, et al., 2019)

References I

Hans L. Bodlaender (1994). "On Disjoint Cycles". In: *International Journal of Foundations of Computer Science* 5.1, pp. 59–68.

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (1999). *Parameterized Complexity*. Monographs in Computer Science. New York: Springer.

Fedor V. Fomin, Serge Gaspers, Daniel Lokshtanov, and Saket Saurabh (2019). "Exact Algorithms via Monotone Local Search". In: *Journal of the ACM* 66.2, 8:1–8:23.

Fedor V. Fomin, Serge Gaspers, Artem V. Pyatkin, and Igor Razgon (2008). "On the minimum feedback vertex set problem: exact and enumeration algorithms". In: *Algorithmica* 52.2, pp. 293–307.

Fedor V. Fomin and Yngve Villanger (2010). "Finding Induced Subgraphs via Minimal Triangulations". In: *Proceedings of the 27th International Symposium on Theoretical Aspects of Computer Science (STACS 2010)*. Vol. 5. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, pp. 383–394.

References II

Yoichi Iwata and Yusuke Kobayashi (2019). *Improved Analysis of Highest-Degree Branching for Feedback Vertex Set*. Tech. rep. abs/1905.12233. arXiv CoRR. URL: http://arxiv.org/abs/1905.12233.

Jason Li and Jesper Nederlof (2019). *Detecting Feedback Vertex Sets of Size k in $O^*(2.7^k)$ Time*. Tech. rep. abs/1906.12298. arXiv CoRR. URL: http://arxiv.org/abs/1906.12298.

Mingyu Xiao and Hiroshi Nagamochi (2015). "An improved exact algorithm for undirected feedback vertex set". In: *Journal of Combinatorial Optimization* 30.2, pp. 214–241.

Parameter Treewidth

Serge Gaspers

UNSW

Outline

1. Algorithms for trees
2. Tree decompositions
3. Monadic Second Order Logic
4. Dynamic Programming over Tree Decompositions
   - SAT
   - CSP
5. Further Reading

Outline

Exercise

Recall: An independent set of a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that $G[S]$ has no edge.

> #INDEPENDENT SETS ON TREES
> Input: A tree $T = (V, E)$
> Output: The number of independent sets of T.

- Design a polynomial time algorithm for #INDEPENDENT SETS ON TREES

Solution

- Select an arbitrary root r of T
- Bottom-up dynamic programming (starting at the leaves) to compute, for each subtree T_x rooted at x the values
  - $\#\text{in}(x)$: the number of independent sets of T_x containing x, and
  - $\#\text{out}(x)$: the number of independent sets of T_x not containing x.
- If x is a leaf, then $\#\text{in}(x) = \#\text{out}(x) = 1$
- Otherwise,

$$\#\text{in}(x) = \Pi_{y \in \text{children}(x)} \, \#\text{out}(y) \text{ and}$$
$$\#\text{out}(x) = \Pi_{y \in \text{children}(x)} \, (\#\text{in}(y) + \#\text{out}(y))$$

- The final result is $\#\text{in}(r) + \#\text{out}(r)$

Exercise

Recall: A dominating set of a graph $G = (V, E)$ is a set of vertices $S \subseteq V$ such that $N_G[S] = V$.

#Dominating Sets on Trees

Input:	A tree $T = (V, E)$
Output:	The number of dominating sets of T.

- Design a polynomial time algorithm for #Dominating Sets on Trees

Solution

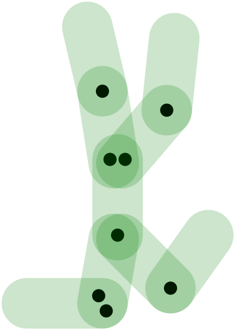
- Select an arbitrary root r of T
- Bottom-up dynamic programming (starting at the leaves) to compute, for each subtree T_x rooted at x the values
  - $\#\mathsf{in}(x)$: the number of dominating sets of T_x containing x,
  - $\#\mathsf{outD}(x)$: the number of dominating sets of T_x not containing x, and
  - $\#\mathsf{outND}(x)$: the number of vertex subsets of T_x dominating $V(T_x) \setminus \{x\}$.
- If x is a leaf, then $\#\mathsf{in}(x) = \#\mathsf{outND}(x) = 1$ and $\#\mathsf{outD}(x) = 0$.
- Otherwise,

$$\begin{aligned}
\#\mathsf{in}(x) &= \Pi_{y \in \mathsf{children}(x)} \left(\#\mathsf{in}(y) + \#\mathsf{outD}(y) + \#\mathsf{outND}(y)\right), \\
\#\mathsf{outD}(x) &= \Pi_{y \in \mathsf{children}(x)} \left(\#\mathsf{in}(y) + \#\mathsf{outD}(y)\right) \\
&\qquad - \Pi_{y \in \mathsf{children}(x)} \#\mathsf{outD}(y) \\
\#\mathsf{outND}(x) &= \Pi_{y \in \mathsf{children}(x)} \#\mathsf{outD}(y)
\end{aligned}$$

- The final result is $\#\mathsf{in}(r) + \#\mathsf{outD}(r)$

Outline

Algorithms using graph decompositions

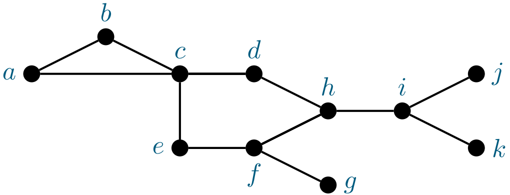


Idea: decompose the problem into subproblems and combine solutions to subproblems to a global solution.

Parameter: overlap between subproblems.

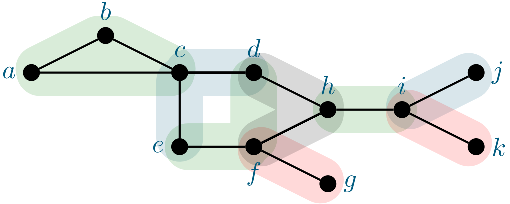
Tree decompositions (by example)

- A graph G

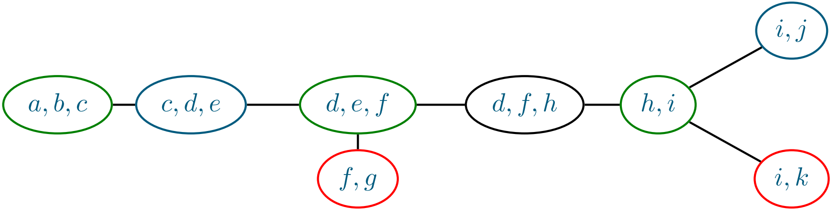


Tree decompositions (by example)

- A graph G

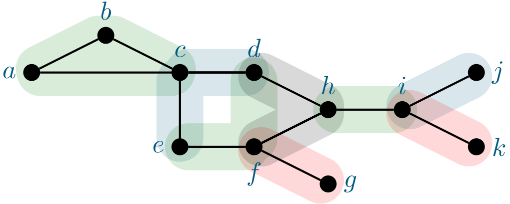


- A *tree decomposition* of G

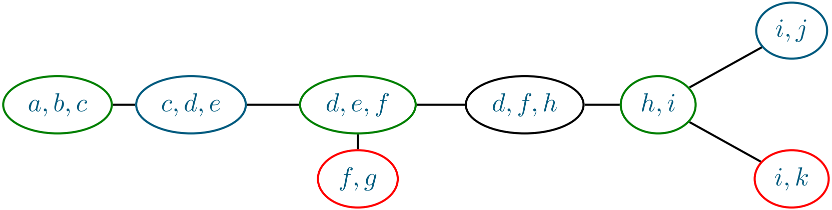


Tree decompositions (by example)

- A graph G



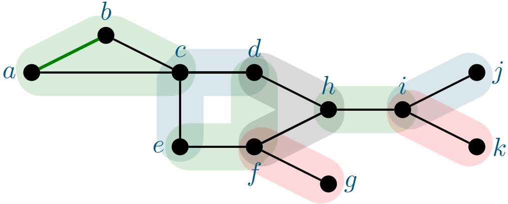
- A *tree decomposition* of G



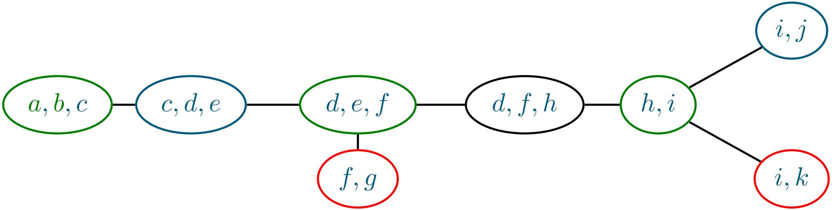
Conditions:

Tree decompositions (by example)

- A graph G



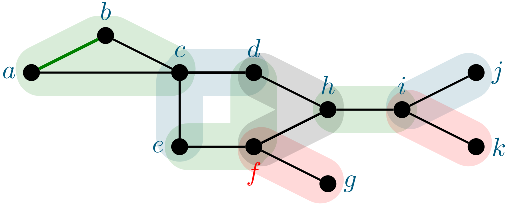
- A *tree decomposition* of G



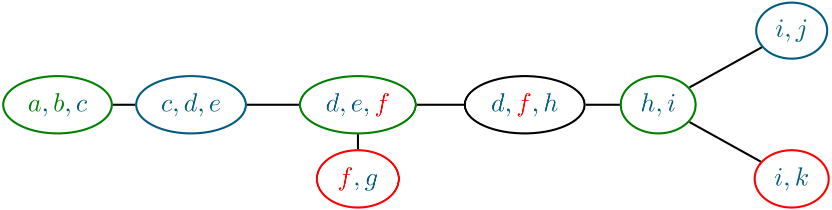
Conditions: covering

Tree decompositions (by example)

- A graph G



- A *tree decomposition* of G



Conditions: covering and connectedness.

Tree decomposition (more formally)

- Let G be a graph, T a tree, and γ a labeling of the vertices of T by sets of vertices of G.
- We refer to the vertices of T as "nodes", and we call the sets $\gamma(t)$ "bags".
- The pair (T, γ) is a *tree decomposition* of G if the following three conditions hold:
  1. For every vertex v of G there exists a node t of T such that $v \in \gamma(t)$.
  2. For every edge vw of G there exists a node t of T such that $v, w \in \gamma(t)$ ("covering").
  3. For any three nodes t_1, t_2, t_3 of T, if t_2 lies on the unique path from t_1 to t_3, then $\gamma(t_1) \cap \gamma(t_3) \subseteq \gamma(t_2)$ ("connectedness").

Treewidth

- The *width* of a tree decomposition (T, γ) is defined as the maximum $|\gamma(t)| - 1$ taken over all nodes t of T.
- The *treewidth* $\mathrm{tw}(G)$ of a graph G is the minimum width taken over all its tree decompositions.

Basic Facts

- Trees have treewidth 1.
- Cycles have treewidth 2.
- Consider a tree decomposition (T, γ) of a graph G and two adjacent nodes i, j in T. Let T_i and T_j denote the two trees obtained from T by deleting the edge ij, such that T_i contains i and T_j contains j. Then, every vertex contained in both $\bigcup_{a \in V(T_i)} \gamma(a)$ and $\bigcup_{b \in V(T_j)} \gamma(b)$ is also contained in $\gamma(i) \cap \gamma(j)$.
- The complete graph on n vertices has treewidth $n - 1$.
- If a graph G contains a clique K_r, then every tree decomposition of G contains a node t such that $K_r \subseteq \gamma(t)$.

Complexity of Treewidth

TREEWIDTH

Input:	Graph $G = (V, E)$, integer k
Parameter:	k
Question:	Does G have treewidth at most k?

- TREEWIDTH is NP-complete.
- TREEWIDTH is FPT: there is a $k^{O(k^3)} \cdot |V|$ time algorithm (Bodlaender, 1996)

Easy problems for bounded treewidth

- Many graph problems that are polynomial time solvable on trees are FPT with parameter treewidth.
- Two general methods:
  - *Dynamic programming*: compute local information in a bottom-up fashion along a tree decomposition
  - *Monadic Second Order Logic*: express graph problem in some logic formalism and use a meta-algorithm

Outline

Monadic Second Order Logic

- **Monadic Second Order (MSO)** Logic is a powerful formalism for expressing graph properties. One can quantify over vertices, edges, vertex sets, and edge sets.
- **Courcelle's theorem** (Courcelle, 1990). Checking whether a graph G satisfies an MSO property is FPT parameterized by the treewidth of G plus the length of the MSO expression.
- **Arnborg et al.'s generalizations** (Arnborg, Lagergren, and Seese, 1991).
  - FPT algorithm for parameter $\mathsf{tw}(G) + |\phi(X)|$ that takes as input a graph G and an MSO sentence $\phi(X)$ where X is a free (non-quantified) vertex set variable, that computes a minimum-sized set of vertices X such that $\phi(X)$ is true in G.
  - Also, the input vertices and edges may be colored and their color can be tested.

Elements of MSO

An MSO formula has

- variables representing vertices $(u, v, \dots)$, edges $(a, b, \dots)$, vertex subsets $(X, Y, \dots)$, or edge subsets $(A, B, \dots)$ in the graph
- atomic operations
  - $u \in X$: testing set membership
  - $X = Y$: testing equality of objects
  - $\mathrm{inc}(u, a)$: incidence test "is vertex u an endpoint of the edge a?"
- propositional logic on subformulas: $\phi_1 \wedge \phi_2$, $\phi_1 \vee \phi_2$, $\neg\phi_1$, $\phi_1 \Rightarrow \phi_2$
- Quantifiers: $\forall X \subseteq V$, $\exists A \subseteq E$, $\forall u \in V$, $\exists a \in E$, etc.

Shortcuts in MSO

We can define some shortcuts

- $u \neq v$ is $\neg(u = v)$
- $X \subseteq Y$ is $\forall v \in V.\ (v \in X) \Rightarrow (v \in Y)$
- $\forall v \in X\ \varphi$ is $\forall v \in V.\ (v \in X) \Rightarrow \varphi$
- $\exists v \in X\ \varphi$ is $\exists v \in V.\ (v \in X) \land \varphi$
- $\mathsf{adj}(u, v)$ is $(u \neq v) \land \exists a \in E.\ (\mathsf{inc}(u, a) \land \mathsf{inc}(v, a))$

MSO Logic Example

Example: 3-COLORING,

- "there are three independent sets in $G = (V, E)$ which form a partition of V"
- $$\begin{aligned} \text{3COL} := \exists R \subseteq V.\ \exists G \subseteq V.\ \exists B \subseteq V.\ & \\ & \text{partition}(R, G, B) \\ & \wedge \text{independent}(R) \wedge \text{independent}(G) \wedge \text{independent}(B), \end{aligned}$$

  where

  $$\begin{aligned} \text{partition}(R, G, B) := \forall v \in V.\ ((& v \in R \wedge v \notin G \wedge v \notin B) \\ & \vee (v \notin R \wedge v \in G \wedge v \notin B) \vee (v \notin R \wedge v \notin G \wedge v \in B)) \end{aligned}$$

  and

  $$\text{independent}(X) := \neg(\exists u \in X.\ \exists v \in X.\ \text{adj}(u, v))$$

MSO Logic Example II

By Courcelle's theorem and our 3COL MSO formula, we have:

Theorem 1

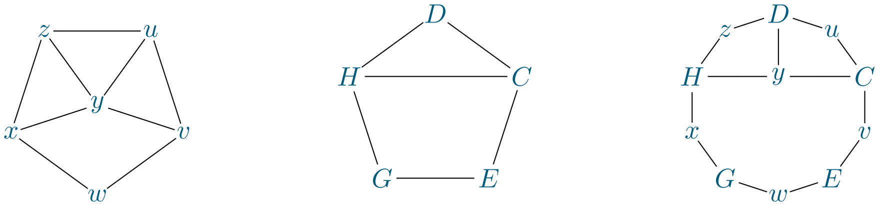
3-Coloring *is* FPT *with parameter treewidth.*

Treewidth only for graph problems?

Let us use treewidth to solve a Logic Problem

- associate a graph with the instance
- take the tree decomposition of the graph
- most widely used: primal graphs, incidence graphs, and dual graphs of formulas.

CNF Formula $F = C \wedge D \wedge E \wedge G \wedge H$ where $C = (u \vee v \vee \neg y)$, $D = (\neg u \vee z \vee y)$, $E = (\neg v \vee w)$, $G = (\neg w \vee x)$, $H = (x \vee y \vee \neg z)$.



primal graph dual graph incidence graph

This gives rise to parameters primal treewidth, dual treewidth, and incidence treewidth.

Formally

Definition 2

Let F be a CNF formula with variables $\mathsf{var}(F)$ and clauses $\mathsf{cla}(F)$.
The primal graph of F is the graph with vertex set $\mathsf{var}(F)$ where two variables are adjacent if they appear together in a clause of F.
The dual graph of F is the graph with vertex set $\mathsf{cla}(F)$ where two clauses are adjacent if they have a variable in common.
The incidence graph of F is the bipartite graph with vertex set $\mathsf{var}(F) \cup \mathsf{cla}(F)$ where a variable and a clause are adjacent if the variable appears in the clause.
The primal treewidth, dual treewidth, and incidence treewidth of F is the treewidth of the primal graph, the dual graph, and the incidence graph of F, respectively.

Incidence treewidth is most general

Lemma 3

The incidence treewidth of F is at most the primal treewidth of F plus 1.

Proof.

Start from a tree decomposition (T, γ) of the primal graph with minimum width. For each clause C:

- There is a node t of T with $\mathsf{var}(C) \subseteq \gamma(t)$, since $\mathsf{var}(C)$ is a clique in the primal graph.
- Add to t a new neighbor t' with $\gamma(t') = \gamma(t) \cup \{C\}$.

□

Incidence treewidth is most general II

Lemma 4

The incidence treewidth of F is at most the dual treewidth of F plus 1.

Incidence treewidth is most general II

Lemma 4

The incidence treewidth of F is at most the dual treewidth of F plus 1.

Primal and dual treewidth are incomparable.

- One big clause alone gives large primal treewidth.
- $\{\{x, y_1\}, \{x, y_2\}, \ldots, \{x, y_n\}\}$ gives large dual treewidth.

SAT parameterized by treewidth

SAT

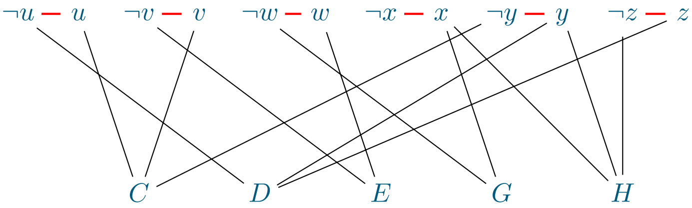
Input: A CNF formula F

Question: Is there an assignment of truth values to var(F) such that F evaluates to true?

Note: If SAT is FPT parameterized by incidence treewidth, then SAT is FPT parameterized by primal treewidth and by dual treewidth.

CNF Formula $F = C \wedge D \wedge E \wedge G \wedge H$ where $C = (u \vee v \vee \neg y)$, $D = (\neg u \vee z \vee y)$, $E = (\neg v \vee w)$, $G = (\neg w \vee x)$, $H = (x \vee y \vee \neg z)$

Auxiliary graph:



- MSO Formula: *"There exists an independent set of literal vertices that dominates all the clause vertices."*
- The treewidth of the auxilary graph is at most twice the treewidth of the incidence graph plus one.

FPT via MSO

Theorem 5

SAT *is* FPT *for each of the following parameters: primal treewidth, dual treewidth, and incidence treewidth.*

Outline

Coucelle's theorem: discussion

Advantages of Courcelle's theorem:

- general, applies to many problems
- easy to obtain FPT results

Drawback of Courcelle's theorem

- the resulting running time depends non-elementarily on the treewidth t and the length ℓ of the MSO-sentence, i.e., a tower of 2's whose height is $\omega(1)$

$$2^{2^{2^{\cdot^{\cdot^{\cdot^{t+\ell}}}}}}$$

Dynamic progamming over tree decompositions

Idea: extend the algorithmic methods that work for trees to tree decompositions.

Step 1 Compute a minimum width tree decomposition using Bodlaender's algorithm

Step 2 Transform it into a standard form making computations easier

Step 3 Bottom-up Dynamic Programming (from the leaves of the tree decomposition to the root)

Nice tree decomposition

A *nice* tree decomposition (T, γ) is rooted and has only 4 kinds of nodes:

- *leaf node*: leaf t in T and $|\gamma(t)| = 1$
- *introduce node*: node t with one child t' in T and $\gamma(t) = \gamma(t') \cup \{x\}$
- *forget node*: node t with one child t' in T and $\gamma(t) = \gamma(t') \setminus \{x\}$
- *join node*: node t with two children t_1, t_2 in T and $\gamma(t) = \gamma(t_1) = \gamma(t_2)$

Every tree decomposition of width w of a graph G on n vertices can be transformed into a nice tree decomposition of width w and $O(w \cdot n)$ nodes in polynomial time (Kloks, 1994).

Outline

1. Algorithms for trees
2. Tree decompositions
3. Monadic Second Order Logic
4. Dynamic Programming over Tree Decompositions
   - SAT
   - CSP
5. Further Reading

Dynamic programming: primal treewidth

- Compute a nice tree decomposition (T, γ) of F's primal graph with minimum width rooted at some node r (Bodlaender, 1996; Kloks, 1994)

Dynamic programming: primal treewidth

- Compute a nice tree decomposition (T, γ) of F's primal graph with minimum width rooted at some node r (Bodlaender, 1996; Kloks, 1994)
- Notation
  - T_t is the subtree of T rooted at node t
  - $\gamma_\downarrow(t) = \{x \in \gamma(t') : t' \in V(T_t)\}$ is the set of vertices/variables in T_t's bags
  - $F_\downarrow(t) = \{C \in \text{cla}(F) : \text{var}(C) \subseteq \gamma_\downarrow(t)\}$ is the set of clauses containing only variables from $\gamma_\downarrow$
  - For a clause $C \in \text{cla}(F)$ and an assignment $\tau : S \to \{0, 1\}$ to a subset of variables $S \subseteq \text{var}(F)$, we can efficiently compute

$$\text{falsifies}(\tau, C) = \begin{cases} 1 & \text{if } \tau \text{ sets each literal of } C \text{ to } 0 \\ 0 & \text{otherwise.} \end{cases}$$

Dynamic programming: primal treewidth

- Compute a nice tree decomposition (T, γ) of F's primal graph with minimum width rooted at some node r (Bodlaender, 1996; Kloks, 1994)
- Notation
  - T_t is the subtree of T rooted at node t
  - $\gamma_\downarrow(t) = \{x \in \gamma(t') : t' \in V(T_t)\}$ is the set of vertices/variables in T_t's bags
  - $F_\downarrow(t) = \{C \in \mathsf{cla}(F) : \mathsf{var}(C) \subseteq \gamma_\downarrow(t)\}$ is the set of clauses containing only variables from $\gamma_\downarrow$
  - For a clause $C \in \mathsf{cla}(F)$ and an assignment $\tau : S \to \{0,1\}$ to a subset of variables $S \subseteq \mathsf{var}(F)$, we can efficiently compute

$$\mathsf{falsifies}(\tau, C) = \begin{cases} 1 & \text{if } \tau \text{ sets each literal of } C \text{ to } 0 \\ 0 & \text{otherwise.} \end{cases}$$

- For each node t and each assignment $\tau : \gamma(t) \to \{0,1\}$, our DP algorithm will compute

$$\mathsf{sat}(t, \tau) = \begin{cases} 1 & \text{if } \tau \text{ can be extended to a} \\ & \text{satisfying assignment of } F_\downarrow(t) \\ 0 & \text{otherwise.} \end{cases}$$

DP: primal treewidth II

$$\mathsf{sat}(t,\tau) = \begin{cases} 1 & \text{if } \tau \text{ can be extended to a} \\ & \text{satisfying assignment of } F_{\downarrow}(t) \\ 0 & \text{otherwise.} \end{cases}$$

- *leaf node*: $|\gamma(t)| = 1$

DP: primal treewidth II

$$\mathsf{sat}(t,\tau) = \begin{cases} 1 & \text{if } \tau \text{ can be extended to a} \\ & \text{satisfying assignment of } F_\downarrow(t) \\ 0 & \text{otherwise.} \end{cases}$$

- *leaf node*: $|\gamma(t)| = 1$

$$\mathsf{sat}(t,\tau) = \begin{cases} 0 & \text{if } \exists C \in \mathsf{cla}(F) \text{ s.t. } \mathsf{falsifies}(\tau, C) \\ 1 & \text{otherwise} \end{cases}$$

DP: primal treewidth II

$$\text{sat}(t, \tau) = \begin{cases} 1 & \text{if } \tau \text{ can be extended to a} \\ & \text{satisfying assignment of } F_{\downarrow}(t) \\ 0 & \text{otherwise.} \end{cases}$$

- *leaf node*: $|\gamma(t)| = 1$

$$\text{sat}(t, \tau) = \begin{cases} 0 & \text{if } \exists C \in \text{cla}(F) \text{ s.t. falsifies}(\tau, C) \\ 1 & \text{otherwise} \end{cases}$$

- *introduce node*: $\gamma(t) = \gamma(t') \cup \{x\}$.

DP: primal treewidth II

$$\text{sat}(t,\tau) = \begin{cases} 1 & \text{if } \tau \text{ can be extended to a} \\ & \text{satisfying assignment of } F_{\downarrow}(t) \\ 0 & \text{otherwise.} \end{cases}$$

- *leaf node*: $|\gamma(t)| = 1$

$$\text{sat}(t,\tau) = \begin{cases} 0 & \text{if } \exists C \in \text{cla}(F) \text{ s.t. } \text{falsifies}(\tau, C) \\ 1 & \text{otherwise} \end{cases}$$

- *introduce node*: $\gamma(t) = \gamma(t') \cup \{x\}$.

$$\text{sat}(t,\tau) = \text{sat}(t', \tau|_{\gamma(t')}) \ \wedge \ (\nexists C \in F : \text{falsifies}(\tau, C)).$$

DP: primal treewidth III

- *forget node*: $\gamma(t) = \gamma(t') \setminus \{x\}$.

DP: primal treewidth III

- *forget node*: $\gamma(t) = \gamma(t') \setminus \{x\}$.

$$\mathsf{sat}(t,\tau) = \mathsf{sat}(t',\tau_{x=0}) \vee \mathsf{sat}(t',\tau_{x=1}),$$
$$\text{where } \tau_{x=a}(y) = \begin{cases} a & \text{if } y = x \\ \tau(y) & \text{otherwise} \end{cases}$$

DP: primal treewidth III

- *forget node*: $\gamma(t) = \gamma(t') \setminus \{x\}$.

$$\mathsf{sat}(t,\tau) = \mathsf{sat}(t',\tau_{x=0}) \vee \mathsf{sat}(t',\tau_{x=1}),$$
$$\text{where } \tau_{x=a}(y) = \begin{cases} a & \text{if } y = x \\ \tau(y) & \text{otherwise} \end{cases}$$

- *join node*: $\gamma(t) = \gamma(t_1) = \gamma(t_2)$

DP: primal treewidth III

- *forget node*: $\gamma(t) = \gamma(t') \setminus \{x\}$.

$$\mathsf{sat}(t,\tau) = \mathsf{sat}(t',\tau_{x=0}) \vee \mathsf{sat}(t',\tau_{x=1}),$$

$$\text{where } \tau_{x=a}(y) = \begin{cases} a & \text{if } y = x \\ \tau(y) & \text{otherwise} \end{cases}$$

- *join node*: $\gamma(t) = \gamma(t_1) = \gamma(t_2)$

$$\mathsf{sat}(t,\tau) = \mathsf{sat}(t_1,\tau) \wedge \mathsf{sat}(t_2,\tau).$$

DP: primal treewidth III

- *forget node*: $\gamma(t) = \gamma(t') \setminus \{x\}$.

$$\mathsf{sat}(t, \tau) = \mathsf{sat}(t', \tau_{x=0}) \vee \mathsf{sat}(t', \tau_{x=1}),$$

$$\text{where } \tau_{x=a}(y) = \begin{cases} a & \text{if } y = x \\ \tau(y) & \text{otherwise} \end{cases}$$

- *join node*: $\gamma(t) = \gamma(t_1) = \gamma(t_2)$

$$\mathsf{sat}(t, \tau) = \mathsf{sat}(t_1, \tau) \wedge \mathsf{sat}(t_2, \tau).$$

- Finally: F is satisfiable iff $\exists \tau : \gamma(r) \to \{0, 1\}$ such that $\mathsf{sat}(r, \tau) = 1$
- Running time: $O^*(2^k)$, where k is the primal treewidth of F
- Also extends to computing the number of satisfying assignments

Direct Algorithms

Known treewidth based algorithms for $\textsc{Sat}$:

$k =$ primal tw	$k =$ dual tw	$k =$ incidence tw
$O^*(2^k)$	$O^*(2^k)$	$O^*(2^k)$

- These algorithms all count the number of satisfying assignments
- The algorithm for incidence treewidth (Slivovsky and Szeider, 2020) uses Fast Subset Convolution

Outline

Constraint Satisfaction Problem

CSP

Input:	A set of variables X, a domain D, and a set of constraints C
Question:	Is there an assignment $\tau : X \to D$ satisfying all the constraints in C?

A constraint has a scope $S = (s_1, \ldots, s_r)$ with $s_i \in X, i \in \{1, \ldots, r\}$, and a constraint relation R consisting of r-tuples of values in D.
An assignment $\tau : X \to D$ satisfies a constraint $c = (S, R)$ if there exists a tuple $(d_1, \ldots, d_r)$ in R such that $\tau(s_i) = d_i$ for each $i \in \{1, \ldots, r\}$.

Bounded Treewidth for Constraint Satisfaction

- Primal, dual, and incidence graphs are defined similarly as for SAT.

Theorem 6 ((Gottlob, Scarcello, and Sideri, 2002))

CSP is FPT for parameter primal treewidth if $|D| = O(1)$.

- What if domains are unbounded?

Unbounded domains

Theorem 7

CSP is W[1]*-hard for parameter primal treewidth.*

Unbounded domains

Theorem 7

CSP is W[1]-hard for parameter primal treewidth.

Proof Sketch.

Parameterized reduction from Clique.
Let $(G = (V, E), k)$ be an instance of Clique.
Take k variables $x_1, \dots, x_k$, each with domain V.
Add $\binom{k}{2}$ binary constraints $E_{i,j}$, $1 \le i < j \le k$.
A constraint $E_{i,j}$ has scope (x_i, x_j) and its constraint relation contains the tuple (u, v) if $uv \in E$.
The primal treewidth of this CSP instance is $k - 1$. $\square$

Outline

Further Reading

- Chapter 7, *Treewidth* in (Cygan et al., 2015)
- Chapter 5, *Treewidth* in (Fomin and Kratsch, 2010)
- Chapter 10, *Tree Decompositions of Graphs* in (Niedermeier, 2006)
- Chapter 10, *Treewidth and Dynamic Programming* in (Downey and Fellows, 2013)
- Chapter 13, *Courcelle's Theorem* in (Downey and Fellows, 2013)

References I

Stefan Arnborg, Jens Lagergren, and Detlef Seese (1991). "Easy problems for tree-decomposable graphs". In: *Journal of Algorithms* 12.2, pp. 308–340.

Hans L. Bodlaender (1996). "A linear-time algorithm for finding tree-decompositions of small treewidth". In: *SIAM Journal on Computing* 25.6, pp. 1305–1317.

Bruno Courcelle (1990). "The monadic second-order logic of graphs. I. Recognizable sets of finite graphs". In: *Information and Computation* 85.1, pp. 12–75.

Marek Cygan, Fedor V. Fomin, Łukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michał Pilipczuk, and Saket Saurabh (2015). *Parameterized Algorithms*. Springer. DOI: 10.1007/978-3-319-21275-3.

Rodney G. Downey and Michael R. Fellows (2013). *Fundamentals of Parameterized Complexity*. Springer. DOI: 10.1007/978-1-4471-5559-1.

Fedor V. Fomin and Dieter Kratsch (2010). *Exact Exponential Algorithms*. Springer. DOI: 10.1007/978-3-642-16533-7.

References II

Georg Gottlob, Francesco Scarcello, and Martha Sideri (2002). "Fixed-parameter complexity in AI and nonmonotonic reasoning". In: *Journal of Artificial Intelligence* 138.1-2, pp. 55–86.

Ton Kloks (1994). *Treewidth: Computations and Approximations*. Berlin: Springer.

Rolf Niedermeier (2006). *Invitation to Fixed Parameter Algorithms*. Oxford University Press. DOI: 10.1093/ACPROF:OSO/9780198566076.001.0001.

Friedrich Slivovsky and Stefan Szeider (2020). "A Faster Algorithm for Propositional Model Counting Parameterized by Incidence Treewidth". In: *Proceedings of the 23rd International Conference on Theory and Applications of Satisfiability Testing (SAT 2020)*. Vol. 12178. Lecture Notes in Computer Science. Springer, pp. 267–276.