AGAPE 09

Spring School

Fixed Parameter and Exact Algorithms

Lozari, Corsica (France)
May 25-29 2009

Table of contents

Mike Fellows

Notes on Parameterized Complexity

AGAPE 2009
Notes on Parameterized Complexity

Mike Fellows

University of Newcastle, Australia
michael.fellows@cs.newcastle.edu.au

Abstract. These notes cover two presentations:
(1) A general overview and introduction to the field.
(2) Parameterized intractability and complexity classes.

1 Introduction to Parameterized Complexity

1.1 Two Forms of Fixed-Parameter Complexity

Many natural computational problems are defined on input consisting of various information, for example, many graph problems are defined as having input consisting of a graph $G = (V, E)$ and a positive integer k. Consider the following well-known problems:

Vertex Cover
Input: A graph $G = (V, E)$ and a positive integer k.
Question: Does G have a vertex cover of size at most k? (A *vertex cover* is a set of vertices $V' \subseteq V$ such that for every edge $uv \in E$, $u \in V'$ or $v \in V'$ (or both).)

Dominating Set
Input: A graph $G = (V, E)$ and a positive integer k.
Question: Does G have a dominating set of size at most k? (A *dominating set* is a set of vertices $V' \subseteq V$ such that $\forall u \in V$: $u \in N[v]$ for some $v \in V'$.)

Although both problems are *NP*-complete, the input *parameter* k contributes to the complexity of these two problems in two qualitatively different ways.

1. There is a simple *bounded search tree* algorithm for Vertex Cover that runs in time $O(2^k n)$
2. The best known algorithm for Dominating Set is basically just the brute force algorithm of trying all k-subsets. For a graph on n vertices this approach has a running time of $O(n^{k+1})$.

(Easy) Exercise: What is the search tree algorithm for Vertex Cover refered to above?

The table below shows the contrast between these two kinds of complexity.

In these two example problems, the parameter is the size of the solution being sought. But a parameter that affects the compexity of a problem can be many things.

	$n = 50$	$n = 100$	$n = 150$
$k = 2$	625	2,500	5,625
$k = 3$	15,625	125,000	421,875
$k = 5$	390,625	6,250,000	31,640,625
$k = 10$	1.9×10^{12}	9.8×10^{14}	3.7×10^{16}
$k = 20$	1.8×10^{26}	9.5×10^{31}	2.1×10^{35}

Table 1. The Ratio $\frac{n^{k+1}}{2^k n}$ for Various Values of n and k.

Example. *The nesting depth of a logical expression.* ML is a logic-based programming language for which relatively efficient compilers exist. One of the problems the compiler must solve is the checking of the compatibility of type declarations. This problem is known to be complete for *EXP* (deterministic exponential time) [HM91], so the situation appears discouraging from the standpoint of classical complexity theory. However, the implementations work well in practice because the ML Type Checking problem is *FPT* with a running time of $O(2^k n)$, where n is the size of the program and k is the maximum nesting depth of the type declarations [LP85]. Since normally $k \leq 5$, the algorithm is clearly practical on the natural input distribution.

The parameter can be size of the solution, or some structural aspect of the natural input distribution — and many other things (to be discussed below).

In the favorable situations (as for Vertex Cover and Type Checking in ML), the exponential cost of solving the problem (that is expected, since the problems are NP-hard) can be entirely confined to an exponential function of the *parameter*, with the overall input size n contributing polynomially.

1.2 Clashes of Function Classes; Multivariate Complexity and Algorithmics

The familiar "P versus NP" framework, that we call the *classical framework*, is fundamentally centered on the notion of *polynomial time*, and this is a one-dimensional framework: there is one measurement (or variable) at work, the overall input size n.

The classical framework revolves around a contrast between two function classes: the *good* class of running times of algorithms of the form: $O(n^c)$, time that is polynomial in the one measurement n. The *bad* class of run times is those of the form 2^{n^c}, and the drama concerns methods for establishing that concrete problems admit good algorithms (and if so, maybe better algorithms?), the *positive toolkit*, or if they only admit bad algorithms (modulo reasonable conjectures), the *negative toolkit* that in the classical case is about NP-hardness, EXP-hardness, etc.

Worth noting at this point is that one of the main motivations to parameterized complexity (and many other approaches) is that while the classical theory is beautiful, and a handful of important problems are in P, the vast majority of problems have turned out to be NP-hard or worse.

Parameterized complexity is basically a two-dimensional sequel, based similarly on a contrast between two function classes in a two-dimensional setting, where in addition to the overall input size n, we have a second measurement (or variable) that captures something else significant that affects computational complexity (and the opportunities for efficient algorithm design), the parameter k (that might be solution size, or something structural about typical inputs, ... or many other things).

How do we formalize this?

Definition 1. *A parameterized language L is a subset $L \subseteq \Sigma^* \times \Sigma^*$. If L is a parameterized language and $(x, k) \in L$ then we will refer to x as the* main part*, and refer to k as the* parameter*.*

A parameter may be non-numerical, and it can also represent an aggregate of various parts or structural properties of the input.

Definition 2. *A parameterized language L is* multiplicatively fixed-parameter tractable *if it can be determined in time $f(k)q(n)$ whether $(x, k) \in L$, where $|x| = n$, $q(n)$ is a polynomial in n, and f is a function (unrestricted).*

Definition 3. *A parameterized language L is* additively fixed-parameter tractable *if it can be determined in time $f(k) + q(n)$ whether $(x, k) \in L$, where $|x| = n$, $q(n)$ is a polynomial in n, and f is a function (unrestricted).*

(Easy) Exercise. Show that a parameterized language is additively fixed-parameter tractable if and only if it is multiplicatively fixed-parameter tractable. This emphasizes how cleanly fixed-parameter tractability isolates the computational difficulty in the complexity contribution of the parameter.

The following definition provides us with a place to put all those problems that are "solvable in polynomial time for fixed k" without making the central distinction about whether this "fixed k" is ending up in the exponent or not (as with the brute force algorithm for Dominating Set).

Definition 4. *A parameterized language L belongs to the class XP if it can be determined in time $f(k)n^{g(k)}$ whether $(x, k) \in L$, where $|x| = n$, with f and g being unrestricted functions.*

Is it possible that $FPT = XP$? This is one of the few structural questions concerning parameterized complexity that currently has an answer [DF98].

Theorem 1. *FPT is a proper subset of XP.*

Summarizing the main point: parameterized complexity is about a natural bivariate generalization of the P versus NP drama. This inevitably leads to two toolkits: the *positive toolkit* of FPT methods (that Daniel Marx will lecture about), and the *negative toolkit* that basically provides a parameterized analog of Cook's Theorem, and methods for showing when fixed-parameter tractable algorithms for parameterized problems are not possible (modulo reasonable assumptions).

There is a larger context captured by the reasonable question that is often asked:

If Parameterized Complexity is the natural two-dimensional sequel to P versus NP, then what is the three-dimensional sequel?

Nobody currently knows the answer. Ideally, one would like to have a *fully multivariate* perspective on complexity analysis and algorithm design that meets the following criteria:
• In dimension 1, you get the (basic) P versus NP drama.
• In dimension 2, you get the (productive) FPT versus XP drama.
• In all dimensions, you have concrete problems where the contrasting outcomes are natural and consequential, and the theory is routinely doable.

Open research problem. Is there such a fully multivariate mathematical perspective?

Introducing (at least one) secondary variable k beyond the overall input size n allows us to ask many new and interesting questions that cannot be asked in any mathematically natural way in the classical framework. Much of this interesting traction is based on the various ways that parameterization can be deployed.

1.3 Parameters Can Be Many Things

There are many ways that parameters arise naturally, for example:

• *The size of a database query.* Normally the size of the database is huge, but frequently queries are small. If n is the size of a relational database, and k is the size of the query, then answering the query (Model Checking) can be solved trivially in time $O(n^k)$. It is known that this problem is unlikely to be *FPT* [DFT96,PY97] because it is hard for $W[1]$, the parameterized analog of NP-hardness. However, if the parameter is the size of the query and the treewidth of the database, then the problem is fixed-parameter tractable [Gr01b].

• *The number of species in an evolutionary tree.* Frequently this parameter is in a range of $k \leq 50$. The PHYLIP computational biology server includes an algorithm which solves the Steiner Problem in Hypercubes in order to compute possible evolutionary trees based on (binary) character information. The exponential heuristic algorithm that is used is in fact an *FPT* algorithm when the parameter is the number of species.

• *The number of variables or clauses in a logical formula, or the number of steps in a deductive procedure.* Determining whether at least k clauses of a CNF formula F are satisfiable is *FPT* with a running time of $O(|F|+1.381^k k^2)$ [BR99]. Since at least half of the m clauses of F can always be satisfied, a more natural parameterization is to ask if at least $m/2 + k$ clauses can be satisfied — this is *FPT* with a running time of $O(|F| + 6.92^k k^2)$ [BR99]. Implementations indicate that these algorithms are quite practical [GN00].

• *The number of moves in a game, or the number of steps in a planning problem.* While most game problems are *PSPACE*-complete classically, it is known that some are *FPT* and others are likely not to be *FPT* (because they are hard for $W[1]$), when parameterized by the number of moves of a winning strategy

[ADF95]. The size n of the input game description usually governs the number of possible moves at any step, so there is a trivial $O(n^k)$ algorithm that just examines the k-step game trees exhaustively.

• *The number of facilities to be located.* Determining whether a planar graph has a dominating set of size at most k is fixed-parameter tractable by an algorithm with a running time of $O(8^k n)$ based on kernelization and search trees. By different methods, an *FPT* running time of $O(3^{36\sqrt{k}})n$ can also be proved.

• *An unrelated parameter.* The input to a problem might come with "extra information" because of the way the input arises. For example, we might be presented with an input graph G together with a k-vertex dominating set in G, and be required to compute an optimal bandwidth layout. Whether this problem is *FPT* is open.

• *The amount of "dirt" in the input or output for a problem.* In the Maximum Agreement Subtree (MAST) problem we are presented with a collection of evolutionary trees trees for a set X of species. These might be obtained by studying different gene families, for example. Because of errors in the data, the trees might not be isomorphic, and the problem is to compute the largest possible subtree on which they do agree. Parameterized by the number of species that need to be deleted to achieve agreement, the MAST problem is *FPT* by an algorithm having a running time of $O(2.27^k + rn^3)$ where r is the number of trees and n is the number of species [NR01].

• *The "robustness" of a solution to a problem, or the distance to a solution.* For example, given a solution of the Minimum Spanning Tree problem in an edge-weighted graph, we can ask if the cost of the solution is robust under all increases in the edge costs, where the parameter is the total amount of cost increases.

• *The distance to an improved solution.* Local search is a mainstay of heuristic algorithm design. The basic idea is that one maintains a *current solution*, and iterates the process of moving to a neighboring "better" solution. A neighboring solution is usually defined as one that is a single step away according to some small edit operation between solutions. The following problem is completely general for these situations, and could potentially provide a valuable subroutine for "speeding up" local search:

k-Speed Up for Local Search
Input: A solution S, k.
Parameter: k
Output: The best solution S' that is within k edit operations of S.

• *The goodness of an approximation.* If we consider the problem of producing solutions whose value is within a factor of $(1+\epsilon)$ of optimal, then we are immediately confronted with a natural parameter $k = 1/\epsilon$. Many of the recent PTAS results for various problems have running times with $1/\epsilon$ in the exponent of the polynomial. Since polynomial exponents larger than 3 are not practical, this is a crucial parameter to consider.

It is obvious that the practical world is full of concrete problems governed by parameters of all kinds that are bounded in small or moderate ranges. If we can design algorithms with running times like $2^k n$ for these problems, then we may have something really useful.

1.4 Kernelization: Another View of FPT

Preprocessing is a practical computing strategy with a lot of power on real world input distributions, as shown by the following example.

Example: Weihe's Train Problem

Weihe describes a problem concerning the train systems of Europe [Wei98]. Consider a bipartite graph $G = (V, E)$ where V is bipartitioned into two sets S (stations) and T (trains), and where an edge represents that a train t stops at a station s. The relevant graphs are huge, on the order of 10,000 vertices. The problem is to compute a minimum number of stations $S' \subseteq S$ such that every train stops at a station in S'. This is a special case of the Hitting Set problem, and is therefore *NP*-complete.

However, the following two reduction rules can be applied to simplify (preprocess) the input to the problem. In describing these rules, let $N(s)$ denote the set of trains that stop at station s, and let $N(t)$ denote the set of stations at which the train t stops.

1. If $N(s) \subseteq N(s')$ then delete s.
2. If $N(t) \subseteq N(t')$ then delete t'.

Applications of these reduction rules cascade, preserving at each step enough information to obtain an optimal solution. Weihe found that, remarkably, these two simple reduction rules were strong enough to "digest" the original, huge input graph into a *problem kernel* consisting of disjoint components of size at most 50 — small enough to allow the problem to be solved optimally by brute force.

The following is an equivalent definition of *FPT* [DFS99].

Definition 5. *A parameterized language L is* kernelizable *if there is there is a parameterized transformation of L to itself, and a function g (unrestricted) that satisfies:*

1. *the running time of the transformation of (x, k) into (x', k'), where $|x| = n$, is bounded by a polynomial $q(n, k)$ (so that in fact this is a polynomial-time transformation of L to itself, considered classically, although with the additional structure of a parameterized reduction),*
2. *$k' \leq k$, and*
3. *$|x'| \leq g(k)$.*

Lemma 1. *A parameterized language L is fixed-parameter tractable if and only if it is kernelizable.*

The proof of this is essentially the solution to the second exercise above.

The kernelization point of view about FPT has become a major enterprise all in itself, that will be covered in the lecture by Saket Saurabh.

There are several points to be noted about kernelization that lead to important research directions:

(1) Kernelization rules are frequently surprising in character, laborious to prove, and nontrivial to discover. Once found, they are small gems of *data reduction* that remain permanently in the heuristic design file for hard problems. No one concerned with any application of HITTING SET on real data should henceforth neglect Weihe's data reduction rules for this problem. The kernelization for VERTEX COVER to graphs of minimum degree 4, for another example, includes the following nontrivial transformation [DFS99]. Suppose G has a vertex x of degree 3 that has three mutually nonadjacent neighbors a, b, c. Then G can be simplified by: (1) deleting x, (2) adding edges from c to all the vertices in $N(a)$, (3) adding edges from a to all the vertices in $N(b)$, (3) adding edges from b to all the vertices in $N(c)$, and (4) adding the edges ab and bc. Note that this transformation is not even symmetric! The resulting (smaller) graph G' has a vertex cover of size k if and only if G has a vertex cover of size k. Moreover, an optimal or good approximate solution for G' lifts constructively to an optimal or good approximate solution for G. The research direction this points to is **to discover these gems of smart preprocessing for all of the hard problems**. There is absolutely nothing to be lost in smart pre-processing, no matter what the subsequent phases of the algorithm (even if the next phase is genetic algorithms or simulated annealing).

(2) Kernelization rules cascade in ways that are surprising, unpredictable in advance, and often quite powerful. Finding a rich set of reduction rules for a hard problem may allow the synergistic cascading of the pre-processing rules to "wrap around" hidden structural aspects of real input distributions. Weihe's train problem provides an excellent example. According to the experience of Alber, Gramm and Niedermeier with implementations of kernelization-based *FPT* algorithms [AGN01], the effort to kernelize is amply rewarded by the subsequently exponentially smaller search tree.

(3) Kernelization is an intrinsically robust algorithmic strategy. Frequently we design algorithms for "pure" combinatorial problems that are not quite like that in practice, because the modeling is only approximate, the inputs are "dirty", etc. For example, what becomes of our VERTEX COVER algorithm if a limited number of edges uv in the graph are *special*, in that it is forbidden to include *both* u and v in the vertex cover? Because they are local in character, the usual kernelization rules are easily adapted to this situation.

(4) Kernelization rules normally preserve all of the information necessary for optimal or approximate solutions. For example, Weihe's kernelization rules for the train problem (HITTING SET) transform the original instance G into a problem kernel G' that can be solved optimally, and the optimal solution for G' "lifts" to an optimal solution for G.

The importance of pre-processing in heuristic design is not a new idea. Cheeseman *et al.* have previously pointed to its importance in the context of artificial intelligence algorithms [CKT91]. What parameterized complexity contributes is a richer theoretical context for this basic element of practical algorithm design. Further research directions include potential methods for mechanizing the discovery and/or verification of reduction rules, and data structures and implementation strategies for efficient kernelization pre-processing.

Lemma 1 of §3 tells us that a parameterized problem is fixed-parameter tractable if and only if there is a polynomial-time kernelization algorithm transforming the input (x, k) into (x', k') where $k' \leq k$ and $|x'| \leq g(k')$ for some function g special to the problem. The basic schema is that reduction rules are applied until an *irreducible* instance (x', k') is obtained. At this point a *Kernel Lemma* is invoked to decide all those reduced instances x' that are larger than $g(k')$ for the kernel-bounding function g. For example, in the cases of VERTEX COVER and PLANAR DOMINATING SET, if a reduced graph G' is larger than $g(k')$ then (G', k') is a no-instance. In the case of MAX LEAF SPANNING TREE large reduced instances are automatically yes-instances. (It is notable that for all three of these problems *linear kernelization*, $g(k) = O(k)$, has been established, in all cases nontrivially [CKJ99,FMcRS01,AFN02].)

How does one proceed to discover an adequate set of reduction rules, or elucidate (and prove) a bounding function $g(k)$ that insures for instances larger than this bound, that the question can be answered directly?

We illustrate a systematic approach with the MAX LEAF SPANNING TREE problem. Our objective is to prove:

The Kernel Lemma. If $(G = (V, E), k)$ is a reduced instance of MAX LEAF SPANNING TREE and G has more than $g(k)$ vertices, then (G, k) is a yes-instance.

We will prove the Kernel Lemma as a corollary to the following.

The Boundary Lemma. If $G = (V, E)$ is a reduced instance of MAX LEAF SPANNING TREE that is a yes-instance for k and a no-instance for $k + 1$, then G has at most $h(k)$ vertices.

Let us first verify that the Kernel Lemma follows from the Boundary Lemma. We will make the mild assumption that our functions $g(k)$ and $h(k)$ are nondecreasing. Take $g(k) = h(k)$. Suppose (G, k) is a counterexample to the Kernel Lemma. Then G is reduced, and has more than $h(k)$ vertices, but is a no-instance, that is, G does not have a spanning tree with at least k leaves. Let $k' < k$ be the maximum number of leaves in a spanning tree of G. Then G is a yes-instance for k' and a no-instance for $k' + 1$. Since $k' < k$ and h is non-decreasing, G has more than $h(k')$ vertices, but this contradicts the Boundary Lemma.

The form of the Boundary Lemma (... which still needs to be proved, and we still need to discover what we mean by "reduced", and we also need to identify the particular bounding function h ...) is conducive to an *extremal theorem* style of argument based on a list of inductive priorities. The proof is sketched as follows.

Sketch Proof of the Boundary Lemma. The proof is by *minimum counterexample*. If there is any counterexample, then we can take G to be one that satisfies:
(1) G is reduced.
(2) G is connected and has more than $h(k)$ vertices.
(3) G is a no-instance for $k+1$.
(4) G is a yes-instance for k, as witnessed by an t-rooted tree subgraph T of G that has k leaves. (We do not assume that T is spanning. Note that if T has k leaves then it can be extended to a spanning tree with at least as many leaves.)
(5) G is a counterexample where T has a minimum possible number of vertices.
(6) Among all of the G, T satisfying (1-5), T has a maximum possible number of internal vertices that are adjacent to a leaf of T.
(7) Among all of the G, T satisfying (1-6), the quantity $\sum_{l \in L} d(t,l)$ is minimized, where L is the set of leaves of T and $d(t,l)$ is the distance in T to the "root" vertex t.
Then we argue for a contradiction.

Comment. The point of all this is to set up a framework for argument that will allow us to see what reduction rules are needed, and what $g(k)$ can be achieved. In essence we are setting up a (possibly elaborate, in the spirit of extremal graph theory) argument by minimum counterexample — and using this as a discovery process for the *FPT* algorithm design. The witness structure T of condition (4) gives us a way of "coordinatizing" the situation — giving us some structure to work with in our inductive argument. How this strucuture is used will become clear as we proceed.

We refer to the vertices of $V - T$ as *outsiders*. The following structural claims are easily established. The first five claims are enforced by condition (3), that is, if any of these conditions did not hold, then we could extend T to a tree T' having one more leaf.

Claim 1: No outsider is adjacent to an internal vertex of T.

Claim 2: No leaf of T can be adjacent to two outsiders.

Claim 3: No outsider has three or more outsider neighbors.

Claim 4: No outsider with 2 outsider neighbors is connected to a leaf of T.

Claim 5: The graph induced by the outsider vertices has no cycles.

It follows from Claims (1-5) that the subgraph induced by the outsiders consists of a collection of paths, where the internal vertices of the paths have degree 2 in G. Since we are ultimately attempting to bound the size of G, this suggests (as a discovery process) the following reduction rule for kernelization.

Kernelization Rule 1: If (G,k) has two adjacent vertices u and v of degree 2, then:
(Rule 1.1) If uv is a bridge, then contract uv to obtain G' and let $k' = k$.
(Rule 1.2) If uv is not a bridge, then delete the edge uv to obtain G' and let $k' = k$.

The soundness of this reduction rule is not completely obvious, although not difficult. Having now partly clarified condition (1), we can continue the argument.

The components of the subgraph induced by the outsiders must consist of paths having either 1,2 or 3 vertices.

Because we are trying to efficiently bound the total number of outsiders (as well as everything else, eventually, in order to obtain the best possible kernelization bound $h(k)$), the situation suggests we should look for further reduction rules to address the remaining possible situations with respect to the outsiders. This discovery process leads us to the following further kernelization rules.

Kernelization Rule 2: If (G, k) is a (connected) instance of MAX LEAF where G has a vertex u of degree 1, with neighbor v, and where $\exists x \notin N(v)$ (that is, not every vertex of G is a neighbor of v), then transform (G, k) into (G', k'), where $k = k'$ and G' is obtained by:
(1) deleting u, and
(2) adding edges to make $N[v]$ into a clique.

The reader can verify that this rule is sound: (G, k) is a yes-instance if and only if (G', k') is a yes-instance.

Kernelization Rule 3: If (G, k) is a (connected) instance of MAX LEAF where G has two vertices u and v such that either:
(1) u and v are adjacent, and $N[u] = N[v]$, or
(2) u and v are not adjacent, and $N(u) = N(v)$,
and also (in either case) there is at least one vertex of G not in $N[u] \cup N[v]$, then transform (G, k) to (G', k') where $k' = k - 1$ and G' is obtained by deleting u.

Returning to our consideration of the outsiders, we are now in the situation that for a reduced graph, the only possibilities are:
(1) A component of the outsider graph is a single vertex having at least 2 leaf neighbors in T.
(2) A component of the outsider graph is a K_2 having at least three leaf neighbors in T.
(3) A component of the outsider is a path of three vertices P_3 having at least four leaf neighbors in T.
The weakest of the ratios is given by case (3). We can conclude that the number of outsiders is bounded by $3k/4$.
The next step is to study the tree T. Since it has k leaves, it has at most $k - 2$ branch vertices. Using conditions (5) and (6), but omitting the details, it is argued that: (1) the paths in T between a leaf and its parental branch vertex has no subdivisions, and (2) any other path in T between branch vertices has at most 3 subdivisions (with respect to T). These statements are proved by various further structural claims (as in the analysis of the outsider population) that must hold, else one of the inductive priorities would fail (constructively) — a tree with $k + 1$ leaves would be possible, or a smaller T, or a T with more internal vertices adjacent to leaves can be devised, or one with a better score on the sum-of-distances priority (7). Consequently T has at most $5k$ vertices, unless there is a contradiction. Together with the bound on the outsiders in a reduced graph, this yields a $g(k)$ of $5.75k$. □

The above sketch illustrates how the project of proving an *FPT* kernelization bound is integrated with the search for efficient kernelization rules. But there is

more to the story. The argument above also leads directly to a constant-factor polynomial-time approximation algorithm in the following way. First, reduce G using the kernelization rules. It is easy to verify that the rules are approximation-preserving. Thus, we might as well suppose that G is reduced to begin with. Now take *any* tree T (not necessarily spanning) in G. If all of the structural claims hold, then (by our arguments above) the tree T must have at least n/c leaves for $c = 5.75$, and therefore we already have (trivially) a c-approximation. (It would require further arguments, but probably the approximation factor is much better than c.) If at least one of the structural claims does not hold, then the tree T can be improved against one of the inductive priorities. Notice that each claim is proved (in the kernelization argument above) by a constructive consequence. For example, if Claim 1 did not hold, then we can find a tree T' (by modifying T) that has one more leaf. Similarly, each claim violation yields a constructive consequence against one of the inductive priorities in the extremal argument for the kernelization bound. These consequences can be applied to our original T (and its successors) only a polynomial number of times (determined by the list of inductive priorities) until we arrive at a tree T' for which all of the various structural claims hold. At that point, we must have a c-approximate solution.

2 Parameterized Intractability and Structural Complexity

Is there a parameterized analog of Cook's Theorem? Yes there is!

2.1 Various Forms of The Halting Problem: A Central Reference Point

The main investigations of computability and efficient computability are tied to three basic forms of the Halting Problem.

1. The Halting Problem
   Input: A Turing machine M.
   Question: If M is started on an empty input tape, will it ever halt?
2. The Polynomial-Time Halting Problem for Nondeterministic Turing Machines
   Input: A nondeterministic Turing machine M.
   Question: Is it possible for M to reach a halting state in n steps, where n is the length of the description of M?
3. The k-Step Halting Problem for Nondeterministic Turing Machines
   Input: A nondeterministic Turing machine M and a positive integer k. (The number of transitions that might be made at any step of the computation is unbounded, and the alphabet size is also unrestricted.)
   Parameter: k
   Question: Is it possible for M to reach a halting state in at most k steps?

The first form of the HALTING PROBLEM is useful for studying the question:

"Is there ANY algorithm for my problem?"

The second form of the HALTING PROBLEM has proved useful for nearly 30 years in addressing the question:

"Is there an algorithm for my problem ... like the ones for Sorting and Matrix Multiplication?"

The second form of the HALTING PROBLEM is trivially *NP*-complete, and essentially defines the complexity class *NP*. For a concrete example of why it is trivially *NP*-complete, consider the 3-COLORING problem for graphs, and notice how easily it reduces to the *P*-TIME NDTM HALTING PROBLEM. Given a graph G for which 3-colorability is to be determined, we just create the following nondeterministic algorithm:

Phase 1. (There are n lines of code here if G has n vertices.)
(1.1) Color vertex 1 one of the three colors nondeterministically.
(1.2) Color vertex 2 one of the three colors nondeterministically.
...
(1.n) Color vertex n one of the three colors nondeterministically.

Phase 2. Check to see if the coloring is proper and if so halt. Otherwise go into an infinite loop.

It is easy to see that the above nondeterministic algorithm has the possibility of halting in m steps (for a suitably padded Turing machine description of size m) if and only if the graph G admits a 3-coloring. Reducing any other problem $\Pi \in NP$ to the *P*-TIME NDTM HALTING PROBLEM is no more difficult than taking an argument that the problem Π belongs to NP and modifying it slightly to be a reduction to this form of the HALTING PROBLEM. It is in this sense that the *P*-TIME NDTM HALTING PROBLEM is essentially the *defining* problem for NP.

The conjecture that $P \neq NP$ is intuitively well-founded. The second form of the HALTING PROBLEM would seem to require exponential time because there is little we can do to analyze unstructured nondeterminism other than to exhaustively explore the possible computation paths.

When the question is:

"Is there an algorithm for my problem ... like the one for Vertex Cover?"

the third form of the HALTING PROBLEM anchors the discussion.

The third natural form of the HALTING PROBLEM is trivially solvable in time $O(n^k)$ by exploring the n-branching, depth-k tree of possible computation paths exhaustively. Our intuition here is essentially the same as for the second form of the Halting Problem — that this cannot be improved. The third form of the Halting Problem defines the parameterized complexity class $W[1]$. Thus $W[1]$ is strongly analogous to NP, and the conjecture that $FPT \neq W[1]$ stands on much the same intuitive grounds as the conjecture that $P \neq NP$. The appropriate notion of problem reduction is as follows.

Definition 6. *A parametric transformation from a parameterized language L to a parameterized language L' is an algorithm that computes from input consisting of a pair (x, k), a pair (x', k') such that:*

1. *$(x, k) \in L$ if and only if $(x', k') \in L'$,*
2. *$k' = g(k)$ is a function only of k, and*
3. *the computation is accomplished in time $f(k)n^{\alpha}$, where $n = |x|$, α is a constant independent of both n and k, and f is an arbitrary function.*

Hardness for $W[1]$ is the working criterion that a parameterized problem is unlikely to be *FPT*. The k-Clique problem is $W[1]$-complete [DF98], and often provides a convenient starting point for $W[1]$-hardness demonstrations. This is the parameterized analog of Cook's Theorem, that the third form of the Halting Problem is FPT if and only if the k-Clique problem is FPT.

The main classes of parameterized problems are organized in the tower

$$P \subseteq lin(k) \subseteq poly(k) \subseteq FPT \subseteq M[1] \subseteq W[1] \subseteq M[2] \subseteq W[2] \subseteq \cdots W[P] \subseteq XP$$

2.2 The W[t] Classes

Loosely speaking, the W-hierarchy captures the complexity of the quest for small solutions for constant depth circuits by stepwise increasing the allowed *weft*. The *weft* of a circuit is the maximum number of large gates (of unbounded fan-in) on any input-output path of the circuit. More precisely, W[t] is characterized by the complete problem asking for satisfying assignments of (Hamming-)*weight* k for constant depth circuits of weft t. Here k is the parameter.

Historically, the $W[t]$-hierarchy was inspired by the observation that the parameterized reduction of Clique to the k-weighted satisfiability problem for circuits produces circuits of weft 1 (and depth 2), while the reduction for Dominating Set produces circuits of weft 2, and yet there seems to be no parameterized reduction from Dominating Set to Clique.

Let Γ be a set of circuits. The *k-weighted satisfiability problem of Γ* is the problem WSat(Γ):

Instance: A circuit $\mathcal{C} \in \Gamma$ and a natural k.
Parameter: k.
Problem: Is there an assignment of weight k satisfying $\mathcal{C}$?

Here the *weight* of an assignment is the number of variables that it maps to 1.

W[t] contains all and only the parameterized problems that are for some d fpt reducible to the weighted circuit satisfiability problem WSat($\Omega_{t,d}$) where $\Omega_{t,d}$ is the set of Boolean circuits of weft t and depth at most d. W[P] is defined similarly by WSat(Circ) where Circ is the set of Boolean circuits.

2.3 The M[t] Classes

There is an important class of parameterized problems seemingly intermediate between FPT and $W[1]$:

$$FPT \subseteq M[1] \subseteq W[1]$$

There are two natural "routes" to $M[1]$.

The renormalization route to $M[1]$.
There are $O^*(2^{O(k)})$ FPT algorithms for many parameterized problems, such as Vertex Cover. In view of this, we can "renormalize" and define the problem:

$k \log n$ Vertex Cover
Input: A graph G on n vertices and an integer k; **Parameter:** k; **Question:** Does G have a vertex cover of size at most $k \log n$?

The FPT algorithm for the original Vertex Cover problem, parameterized by the number of vertices in the vertex cover, allows us to place this new problem in XP. It now makes sense to ask whether the $k \log n$ Vertex Cover problem is also in FPT — or is it parametrically intractable? It turns out that $k \log n$ Vertex Cover is $M[1]$-complete.

The miniaturization route to $M[1]$.
We certainly know an algorithm to solve n-variable 3SAT in time $O(2^n)$. Consider the following parameterized problem.

Mini-3SAT
Input: Positive integers k and n in unary, and a 3SAT expression E having at most $k \log n$ variables; **Parameter:** k; **Question:** Is E satisfiable?

Using our exponential time algorithm for 3SAT, Mini-3SAT is in XP and we can wonder where it belongs — is it in FPT or is it parametrically intractable? This problem also turns out to be complete for $M[1]$.

Dozens of renormalized FPT problems and miniaturized arbitrary problems are now known to be $M[1]$-complete. However, what is known is quite problem-specific. For example, one might expect Mini-Max Leaf to be $M[1]$-complete, but all that is known presently is that it is $M[1]$-hard. It is not known to be $W[1]$-hard, nor is it known to belong to $W[1]$.

The following theorem would be interpreted by most people as indicating that probably $FPT \neq M[1]$. (The theorem is essentially due to Cai and Juedes [CJ01], making use of a result of Impagliazzo, Paturi and Zane [IPZ98].)

Theorem 2. *$FPT = M[1]$ if and only if n-variable 3SAT can be solved in time $2^{o(n)}$.*

$M[1]$ supports convenient although unusual combinatorics. For example, one of the problems that is $M[1]$-complete is the miniature of the Independent Set problem defined as follows.

Mini-Independent Set
Input: Positive integers k and n in unary, a positive integer $r \leq n$, and a graph G having at most $k \log n$ vertices.

Parameter: k
Question: Does G have an independent set of size at least r?

Theorem 3. *There is an FPT reduction from* Mini-Independent Set *to ordinary parameterized* Independent Set *(parameterized by the number of vertices in the independent set).*

Proof. Let $G = (V, E)$ be the miniature, for which we wish to determine whether G has an independent set of size r. Here, of course, $|V| \leq k \log n$ and we may regard the vertices of G as organized in k blocks $V_1, ..., V_k$ of size $\log n$. We now employ a simple but useful *counting trick* that can be used when reducing miniatures to "normal" parameterized problems. Our reduction is a Turing reduction, with one branch for each possible way of writing r as a sum of k terms, $r = r_1 + \cdots + r_k$, where each r_i is bounded by $\log n$. The reader can verify that $(\log n)^k$ is an *FPT* function, and thus that there are an allowed number of branches. A branch represents a commitment to choose r_i vertices from block V_i (for each i) to be in the independent set.

We now produce (for a given branch of the Turing reduction) a graph G' that has an independent set of size k if and only if the miniature G has an independent set of size r, distributed as indicated by the commitment made on that branch. The graph G' consists of k cliques, together with some edges between these cliques. The ith clique consists of vertices in 1:1 correspondence with the subsets of V_i of size r_i. An edge connects a vertex x in the ith clique and a vertex y in the jth clique if and only if there is a vertex u in the subset $S_x \subseteq V_i$ represented by x, and a vertex v in the subset $S_y \subseteq V_j$ represented by y, such that $uv \in E$. Verification is straightforward.

The theorem above shows that $M[1]$ is contained in $W[1]$.

Cai and Juedes [CJ01] proved the following, opening up a broad program of studying the optimality of FPT algorithms.

Theorem 4. *If* $FPT \neq M[1]$ *then there cannot be an FPT algorithm for the general* Vertex Cover *problem with a parameter function of the form* $f(k) = 2^{o(k)}$*, and there cannot be an FPT algorithm for the* Planar Vertex Cover *problem with a parameter function of the form* $f(k) = 2^{o(\sqrt{k})}$.

It has previously been known that Planar Dominating Set, parameterized by the number n of vertices in the graph can be solved optimally in time $O^*(2^{O(\sqrt{n})})$ by using the Lipton-Tarjan Planar Separator Theorem. Combining the lower bound theorem of Cai-Juedes with the linear kernelization result of Alber et al. [AFN02] shows that this cannot be improved to $O^*(2^{o(\sqrt{n})})$ unless $FPT = M[1]$.

2.4 An Example of a $W[1]$-hardness Reduction

We take as our example, how parameterized complexity can be used to study the complexity of approximation. Approximation immediately concerns a fundamental parameter: $k = 1/\epsilon$, *the goodness of the approximation.*

To illustrate the issue, consider the following more-or-less random sample of recent PTAS results:

- The PTAS for the EUCLIDEAN TSP due to Arora [Ar96] has a running time of around $O(n^{3000/\epsilon})$. Thus for a 20% error, we have a "polynomial-time" algorithm that runs in time $O(n^{15000})$.
- The PTAS for the MULTIPLE KNAPSACK problem due to Chekuri and Khanna [CK00] has a running time of $O(n^{12(\log(1/\epsilon)/\epsilon^8)})$. Thus for a 20% error we have a polynomial-time algorithm that runs in time $O(n^{9375000})$.
- The PTAS for the MINIMUM COST ROUTING SPANNING TREE problem due to Wu, Lancia, Banfna, Chao, Ravi and Tang [WLBCRT98] has a running time of $O(n^{2^{\lceil 2/\epsilon \rceil}-2})$. For a 20% error, we thus have a running time of $O(n^{18})$.
- The PTAS for the UNBOUNDED BATCH SCHEDULING problem due to Deng, Feng, Zhang and Zhu [DFZZ01] has a running time of $O(n^{5\log_{1+\epsilon}(1+(1/\epsilon))})$. Thus for a 20% error we have an $O(n^{50})$ polynomial-time algorithm.
- The PTAS for TWO-VEHICLE SCHEDULING ON A PATH due to Karuno and Nagamochi [KN01] has a running time of $O(n^{8(1+(2/\epsilon))})$; thus $O(n^{88})$ for a 20% error.
- The PTAS for the MAXIMUM SUBFOREST PROBLEM due to Shamir and Tsur [ST98] has a running time of $O(n^{2^{2^{1/\epsilon}}-1})$. For a 20% error we thus have a "polynomial" running time of $O(n^{958267391})$.
- The PTAS for the MAXIMUM INDENDENT SET problem on geometric graphs due to Erlebach, Jansen and Seidel [EJS01] has a running time of $O(n^{(4/\pi)(1/\epsilon^2+2)^2(1/\epsilon^2+1)^2})$. Thus for a 20% error we have a running time of $O(n^{532804})$.
- The PTAS for the CLASS-CONSTRAINED PACKING PROBLEM due to Shachnai and Tamir [ST00] has a running time (for 3 colors) of $O(n^{64/\epsilon+(log(1/\epsilon)/\epsilon^8)})$. Thus for a 20% error (for 3 colors) we have a running time of $O(n^{1021570})$.
- The PTAS for the problem of BASE STATION POSITIONING IN UMTS NETWORKS due to Galota, Glasser, Reith and Vollmer [GGRV01] has a running time of $O(n^{25/\epsilon^2})$, and thus $O(n^{627})$ time for a 20% error.
- The PTAS for the GENERAL MULTIPROCESSOR JOB SCHEDULING PROBLEM due to Chen and Miranda [CM99] runs in time $O(n^{(3mm!)^{(m/\epsilon)+1}})$ for m machines. Thus for 4 machines with a 20% error we have an algorithm that runs in time $O(n^{10000000000000000000000000000000000000000000000000000000000000000000000000000})$ or so.

Since polynomial-time algorithms with exponent greater than 3 are generally not very practical, the following question would seem to be important.

Can we get the $k = 1/\epsilon$ out of the exponent?

The following definition captures the essential issue.

Definition 7. *An optimization problem Π has an* efficient P-time approximation scheme *(EPTAS) if it can be approximated to a goodness of $(1+\epsilon)$ of optimal in time $f(k)n^c$ where c is a constant and $k = 1/\epsilon$.*

In 1997, Arora gave an EPTAS for the EUCLIDEAN TSP [Ar97].

The following easy but important connection between parameterized complexity and approximation was first proved by Bazgan [Baz95,CT97].

Theorem 5. *Suppose that Π_{opt} is an optimization problem, and that Π_{param} is the corresponding parameterized problem, where the parameter is the value of an optimal solution. Then Π_{param} is fixed-parameter tractable if Π_{opt} has an efficient PTAS.*

Applying Bazgan's Theorem is not necessarily difficult — we will sketch here a recent example. Khanna and Motwani introduced three planar logic problems in an interesting effort to give a general explanation of PTAS-approximability. Their suggestion is that "hidden planar structure" in the logic of an optimization problem is what allows PTASs to be developed [KM96]. They gave examples of optimization problems known to have PTASs, problems having nothing to do with graphs, that could nevertheless be reduced to these planar logic problems. The PTASs for the planar logic problems thus "explain" the PTASs for these other problems. Here is one of their three general planar logic optimization problems.

PLANAR TMIN
Input: A collection of Boolean formulas in sum-of-products form, with all literals positive, where the associated bipartite graph is planar (this graph has a vertex for each formula and a vertex for each variable, and an edge between two such vertices if the variable occurs in the formula).
Output: A truth assignment of minimum weight (i.e., a minimum number of variables set to *true*) that satisfies all the formulas.

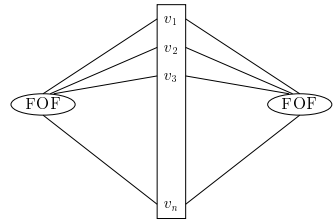
The following theorem is from joint work with Cai, Juedes and Rosamond [CFJR01].

Theorem 6. *Planar TMIN is hard for $W[1]$ and therefore does not have an EPTAS unless $FPT = W[1]$.*

Proof. We show that CLIQUE is parameterized reducible to PLANAR TMIN with the parameter being the weight of a truth assignment. Since CLIQUE is W[1]-complete, it will follow that the parameterized form of PLANAR TMIN is W[1]-hard.

To begin, let $\langle G, k\rangle$ be an instance of CLIQUE. Assume that G has n vertices. From G and k, we will construct a collection C of FOFs (sum-of-products formulas) over $f(k)$ blocks of n variables. C will contain at most $2f(k)$ FOFs and the incidence graph of C will be planar. Moreover, each minterm in each FOF will contain at most 4 variables. The collection C is constructed so that G has a clique of size k if and only if C has a weight $f(k)$ satisfying assignment with exactly one variable set to true in each block of n variables. Here we have that $f(k) = O(k^4)$.

To maintain planarity in the incidence graph for C, we ensure that each block of n variables appears in at most 2 FOFs. If this condition is maintained, then we can draw each block of n variables as follows.

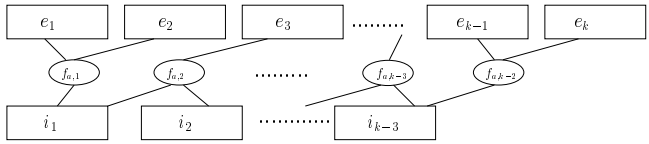


We describe the construction in two stages. In the first stage, we use k blocks of n variables and a collection C' of $k(k-1)/2+k$ FOFs. In a weight k satisfying assignment for C', exactly one variable v_i, j in each block of variables $b_i = [v_{i,1}, \ldots, v_{i,n}]$ will be set to true. We interpret this event as "vertex j is the ith vertex in the clique of size k." The $k(k-1)/2+k$ FOFs are described as follows. For each $1 \leq i \leq k$, let f_i be the FOF $\bigvee_{j=1}^{n} v_{i,j}$. This FOF ensures that at least one variable in b_i is set to true. For each pair $1 \leq i < j \leq k$, let $f_{i,j}$ be the FOF $\bigvee_{(u,v)\in E} v_{i,u}v_{j,v}$. Each FOF $f_{i,j}$ ensures that there is an edge in G between the ith vertex the clique and the jth vertex in the clique.

It is somewhat straightforward to show that $C' = \{f_1, \ldots, f_k, f_{1,2}, \ldots, f_{k-1,k}\}$ has a weight k satisfying assignment if and only if G has a clique of size k. To see this, notice that any weight k satisfying assignment for C' must satisfy exactly 1 variable in each block b_i. Each first order formula $f_{i,j}$ ensures that there is an edge between the ith vertex in the potential clique and the jth vertex in the potential clique. Notice also that, since we assume that G does not contain edges of the form (u, u), the FOF $f_{i,j}$ also ensures that the ith vertex in the potential clique is not the jth vertex in the potential clique. This completes the first stage.

The incidence graph for the collection C' in the first stage is almost certainly not planar. In the second stage, we achieve planarity by removing crossovers in incidence graph for C'. Here we use two types of widgets to remove crossovers while keeping the number of variables per minterm bounded by 4. The first widget A_k consists of $k + k - 3$ blocks of n variables and $k - 2$ FOFs. This widget consists of $k-3$ internal and k external blocks of variables. Each external block $e_i = [e_{i,1}, \ldots, e_{i,n}]$ of variables is connected to exactly one FOF inside the widget. Each internal block $i_j = [i_{j,1}, \ldots, e_{j,n}]$ is connected to exactly two FOFs inside the widget. The $k - 2$ FOFs are given as follows. The FOF $f_{a,1}$ is $\bigvee_{j=1}^{n} e_{1,j}e_{2,j}i_{1,j}$. For each $2 \leq l \leq k-3$, the FOF $f_{a,l} = \bigvee_{j=1}^{n} i_{l-1,j}e_{l+1,j}i_{l,j}$. Finally, $f_{a,k-2} = \bigvee_{j=1}^{n} i_{k-3,j}e_{k-1,j}e_{k,j}$. These $k-2$ FOFs ensure that the settings of variables in each block is the same if there is a weight $2k-3$ satisfying assignment to the $2k-3$ blocks of n variables.

The widget A_k can be drawn as follows.

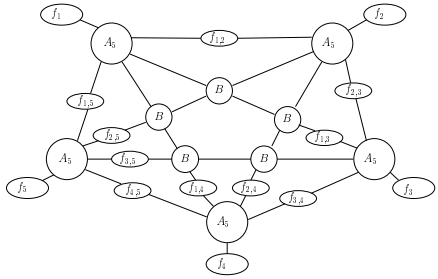


Since each internal block is connected to exactly two FOFs, the incidence graph for this widget can be drawn on the plane without crossing any edges.

The second widget removes crossover edges from the first stage of the construction. In the first stage, crossovers can occur in the incidence graphs because two FOFs may cross from one block to another. To eliminate this, consider each edge i, j in K_k with $i < j$ as a directed edge from i to j. In the construction, we send a copy of block i to block j. At each crossover point from the direction of block $u = [u_1, \ldots, u_n]$ and $v = [v_1, \ldots, v_n]$, insert a widget B that introduces 2 new blocks of n variables $u_1 = [u_{1_1} \ldots u_{1_n}]$ and $v_1 = [v_{1_1} \ldots v_{1_n}]$ and a FOF $f_B = \bigvee_{j=1}^{n} \bigvee_{l=1}^{n} u_j u_{1_j} v_l v_{1_l}$. The FOF f_B ensures that u_1 and v_1 are copies of u and v. Moreover, notice that the incidence graph for the widget B is also planar.

To complete the construction, we replace each of the original k blocks of n variables from the first stage with a copy of the widget A_{k-1}. At each crossover point in the graph, we introduce a copy of widget B. Finally, for each directed edge between blocks (i, j), we insert the original FOF $f_{i,j}$ between the last widget B and the destination widget A_{k-1}. Since one of the new blocks of variables created by the widget B is a copy of block i, the effect of the FOF $f_{i,j}$ in this new collections is the same as before.

The following diagram shows the full construction when $k = 5$.



Since each the incidence graph of each widget in this drawing is planar, the entire collection C of first order formulas has a planar incidence graph.

Now, if we assume that there are $c(k) = O(k^4)$ crossover points in standard drawing of K_k, then our collection has $c(k)$ B widgets. Since each B widget introduces 2 new blocks of n variables, this gives $2c(k)$ new blocks. Since we have k A_{k-1} widgets, each of which has $2(k-1) - 3 = 2k - 5$ blocks of n variables, this gives an additional $k(2k-5)$ blocks. So, in total, our construction

has $f(k) = 2c(k) + 2k^2 - 5k = O(k^4)$ blocks of n variables. Note also that there are $g(k) = k(k-1)/2 + k(k-2) + c(k) = O(k^4)$ FOFs in the collection C.

As shown in our construction C has a weight $f(k)$ satisfying assignment (i.e., each block has exactly one variable set to true) if and only if the original graph G has a clique of size k. Since the incidence graph of C is planar and each minterm in each FOF contains at most four variables, it follows that this construction is a parameterized reduction as claimed. □

In a similar manner the other two planar logic problems defined by Khanna and Motwani can also be shown to be $W[1]$-hard.

3 Recommended Books and Articles

Parameterized Complexity – R. Downey and M. Fellows, Springer, 1999.

Parameterized Complexity Theory – J. Flum and M. Grohe, Springer, 2006.

Invitation to Fixed Parameter Algorithms – R. Niedermeier, Oxford Univ. Press, 2006.

The Computer Journal, 2008, Numbers 1 and 3 – a double special issue of surveys of various aspects and application areas of parameterized complexity.

References

[ADF95] K. Abrahamson, R. Downey and M. Fellows, "Fixed Parameter Tractability and Completeness IV: On Completeness for $W[P]$ and *PSPACE* Analogs," *Annals of Pure and Applied Logic* 73 (1995), 235–276.

[AFN01] J. Alber, H. Fernau and R. Niedermeier, "Parameterized Complexity: Exponential Speed-Up for Planar Graph Problems," in: Proceedings of ICALP 2001, Crete, Greece, *Lecture Notes in Computer Science* vol. 2076 (2001), 261-272.

[AFN02] J. Alber, M. Fellows and R. Niedermeier, "Efficient Data Reduction for Dominating Set: A Linear Problem Kernel for the Planar Case," to appear in the *Proceedings of Scandinavian Workshop on Algorithms and Theory* (SWAT 2002), Springer-Verlag, *Lecture Notes in Computer Science*, 2002.

[AGN01] J. Alber, J. Gramm and R. Niedermeier, "Faster Exact Algorithms for Hard Problems: A Parameterized Point of View," *Discrete Mathematics* 229 (2001), 3–27.

[Ar96] S. Arora, "Polynomial Time Approximation Schemes for Euclidean TSP and Other Geometric Problems," In: *Proceedings of the 37th IEEE Symposium on Foundations of Computer Science*, 1996, pp. 2–12.

[Ar97] S. Arora, "Nearly Linear Time Approximation Schemes for Euclidean TSP and Other Geometric Problems," *Proc. 38th Annual IEEE Symposium on the Foundations of Computing* (FOCS'97), IEEE Press (1997), 554-563.

[Baz95] C. Bazgan, "Schémas d'approximation et complexité paramétrée," Rapport de stage de DEA d'Informatique à Orsay, 1995.

[BR99] N. Bansal and V. Raman, "Upper Bounds for MaxSat: Further Improved," *Proc. 10th International Symposium on Algorithms and Computation (ISAAC '99)*, Springer-Verlag, *Lecture Notes in Computer Science* 1741 (1999), 247–258.

[CFJR01] Liming Cai, M. Fellows, D. Juedes and F. Rosamond, "Efficient Polynomial-Time Approximation Schemes for Problems on Planar Structures: Upper and Lower Bounds," manuscript, 2001.

[CJ01] L. Cai and D. Juedes. "On the Existence of Subexponential Parameterized Algorithms," manuscript, 2001. Revised version of the paper, "Subexponential Parameterized Algorithms Collapse the W-Hierarchy," in: *Proceedings 28th ICALP*, Springer-Verlag LNCS 2076 (2001), 273–284.

[CK00] C. Chekuri and S. Khanna, "A PTAS for the Multiple Knapsack Problem," *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms* (SODA 2000), pp. 213-222.

[CKJ99] J. Chen, I.A. Kanj and W. Jia, "Vertex Cover: Further Observations and Further Improvements," *Proceedings of the 25th International Workshop on Graph-Theoretic Concepts in Computer Science* (WG'99), *Lecture Notes in Computer Science 1665* (1999), 313–324.

[CKT91] P. Cheeseman, B. Kanefsky and W. Taylor, "Where the Really Hard Problems Are," *Proc. 12th International Joint Conference on Artificial Intelligence* (1991), 331-337.

[CM99] J. Chen and A. Miranda, "A Polynomial-Time Approximation Scheme for General Multiprocessor Scheduling," *Proc. ACM Symposium on Theory of Computing* (STOC '99), ACM Press (1999), 418–427.

[CS97] Leizhen Cai and B. Schieber, "A Linear Time Algorithm for Computing the Intersection of All Odd Cycles in a Graph," *Discrete Applied Math.* 73 (1997), 27-34.

[CT97] M. Cesati and L. Trevisan, "On the Efficiency of Polynomial Time Approximation Schemes," *Information Processing Letters* 64 (1997), 165–171.

[CW95] M. Cesati and H. T. Wareham, "Parameterized Complexity Analysis in Robot Motion Planning," *Proceedings 25th IEEE Intl. Conf. on Systems, Man and Cybernetics: Volume 1*, IEEE Press, Los Alamitos, CA (1995), 880-885.

[DF98] R. G. Downey and M. R. Fellows, *Parameterized Complexity*, Springer-Verlag, 1998.

[DFS99] R. G. Downey, M. R. Fellows and U. Stege, "Parameterized Complexity: A Framework for Systematically Confronting Computational Intractability." In: *Contemporary Trends in Discrete Mathematics*, (R. Graham, J. Kratochvíl, J. Nesetril and F. Roberts, eds.), Proceedings of the DIMACS-DIMATIA Workshop, Prague, 1997, *AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, vol. 49 (1999), 49–99.

[DFT96] R. G. Downey, M. Fellows and U. Taylor, "The Parameterized Complexity of Relational Database Queries and an Improved Characterization of $W[1]$," in: *Combinatorics, Complexity and Logic: Proceedings of DMTCS'96*, Springer-Verlag (1997), 194–213.

[DFZZ01] X. Deng, H. Feng, P. Zhang and H. Zhu, "A Polynomial Time Approximation Scheme for Minimizing Total Completion Time of Unbounded Batch Scheduling," *Proc. 12th International Symposium on Algorithms and Computation* (ISAAC '01), Springer-Verlag, *Lecture Notes in Computer Science* 2223 (2001), 26–35.

[EJS01] T. Erlebach, K. Jansen and E. Seidel, "Polynomial Time Approximation Schemes for Geometric Graphs," *Proc. ACM Symposium on Discrete Algorithms* (SODA'01), 2001, pp. 671–679.

[FG01] J. Flum and M. Grohe, "Describing Parameterized Complexity Classes," manuscript, 2001.

[FMcRS01] M. Fellows, C. McCartin, F. Rosamond and U. Stege, "Trees with Few and Many Leaves," manuscript, full version of the paper: "Coordinatized kernels and catalytic reductions: An improved FPT algorithm for max leaf spanning tree and other problems," *Proceedings of the 20th FST TCS Conference*, New Delhi, India, Lecture Notes in Computer Science vol. 1974, Springer Verlag (2000), 240-251.

[GGRV01] M. Galota, C. Glasser, S. Reith and H. Vollmer, "A Polynomial Time Approximation Scheme for Base Station Positioning in UMTS Networks," *Proc. Discrete Algorithms and Methods for Mobile Computing and Communication*, 2001.

[GN00] J. Gramm and R. Niedermeier, "Faster Exact Algorithms for Max2Sat," *Proc. 4th Italian Conference on Algorithms and Complexity*, Springer-Verlag, *Lecture Notes in Computer Science* 1767 (2000), 174–186.

[Gr01a] M. Grohe, "Generalized Model-Checking Problems for First-Order Logic," *Proc. STACS 2001*, Springer-Verlag LNCS vol. 2001 (2001), 12–26.

[Gr01b] M. Grohe, "The Parameterized Complexity of Database Queries," *Proc. PODS 2001*, ACM Press (2001), 82–92.

[GSS01] G. Gottlob, F. Scarcello and M. Sideri, "Fixed Parameter Complexity in AI and Nonmonotonic Reasoning," to appear in *The Artificial Intelligence Journal*. Conference version in: *Proc. of the 5th International Conference on Logic Programming and Nonmonotonic Reasoning* (LPNMR'99), vol. 1730 of *Lecture Notes in Artificial Intelligence* (1999), 1–18.

[HM91] F. Henglein and H. G. Mairson, "The Complexity of Type Inference for Higher-Order Typed Lambda Calculi." In *Proc. Symp. on Principles of Programming Languages (POPL)* (1991), 119-130.

[IPZ98] R. Impagliazzo, R. Paturi and F. Zane. "Which Problems Have Strongly Exponential Complexity?" *Proceedings of the 39th Annual Symposium on Foundations of Computer Science (FOCS'1998)*, 653–663.

[KM96] S. Khanna and R. Motwani, "Towards a Syntactic Characterization of PTAS," in: *Proc. STOC 1996*, ACM Press (1996), 329–337.

[KN01] Y. Karuno and H. Nagamochi, "A Polynomial Time Approximation Scheme for the Multi-vehicle Scheduling Problem on a Path with Release and Handling Times," *Proc. 12th International Symposium on Algorithms and Computation* (ISAAC '01), Springer-Verlag, *Lecture Notes in Computer Science* 2223 (2001), 36–47.

[KR00] S. Khot and V. Raman, 'Parameterized Complexity of Finding Subgraphs with Hereditary properties', *Proceedings of the Sixth Annual International Computing and Combinatorics Conference (COCOON 2000)* July 2000, Sydney, Australia, Lecture Notes in Computer Science, Springer Verlag **1858** (2000) 137-147.

[LP85] O. Lichtenstein and A. Pneuli. "Checking That Finite-State Concurrents Programs Satisfy Their Linear Specification." In: *Proceedings of the 12th ACM Symposium on Principles of Programming Languages* (1985), 97–107.

[MR99] M. Mahajan and V. Raman, "Parameterizing Above Guaranteed Values: MaxSat and MaxCut," *J. Algorithms* 31 (1999), 335-354.

[NR01] R. Niedermeier and P. Rossmanith, "An Efficient Fixed Parameter Algorithm for 3-Hitting Set," *Journal of Discrete Algorithms* 2(1), 2001.

[NSS98] A. Natanzon, R. Shamir and R. Sharan, "A Polynomial-Time Approximation Algorithm for Minimum Fill-In," *Proc. ACM Symposium on the Theory of Computing* (STOC'98), ACM Press (1998), 41–47.

[PY97] C. Papadimitriou and M. Yannakakis, "On the Complexity of Database Queries," *Proc. ACM Symp. on Principles of Database Systems* (1997), 12–19.

[ST98] R. Shamir and D. Tzur, "The Maximum Subforest Problem: Approximation and Exact Algorithms," *Proc. ACM Symposium on Discrete Algorithms* (SODA'98), ACM Press (1998), 394–399.

[ST00] H. Shachnai and T. Tamir, "Polynomial Time Approximation Schemes for Class-Constrained Packing Problems," *Proc. APPROX 2000*.

[St00] U. Stege, "Resolving Conflicts in Problems in Computational Biochemistry," Ph.D. dissertation, ETH, 2000.

[Wei98] K. Weihe, "Covering Trains by Stations, or the Power of Data Reduction," *Proc. ALEX'98* (1998), 1–8.

[Wei00] K. Weihe, "On the Differences Between 'Practical' and 'Applied' (invited paper)," *Proc. WAE 2000*, Springer-Verlag, *Lecture Notes in Computer Science* 1982 (2001), 1–10.

[WLBCRT98] B. Wu, G. Lancia, V. Bafna, K-M. Chao, R. Ravi and C. Tang, "A Polynomial Time Approximation Scheme for Minimum Routing Cost Spanning Trees," *Proc. SODA '98*, 1998, pp. 21–32.

Fedor Fomin

Gridology

Gridology

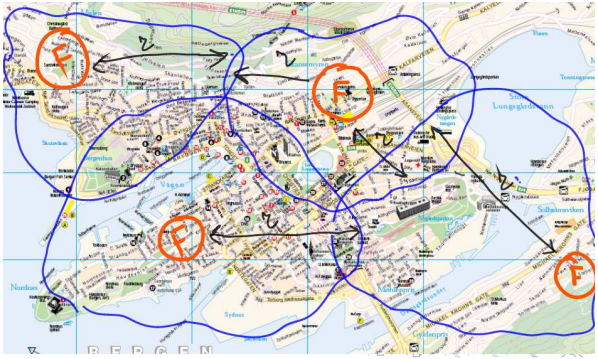
Fedor V. Fomin

Department of Informatics, University of Bergen, Norway

AGAPE 2009

Example: The city of Bergen

How to place k fire stations such that every building is within r city blocks from the nearest fire station?



How to place k fire stations?

- Some simplifications: Bergen is a planar graph and $r = 1$.

How to place k fire stations?

- Some simplifications: Bergen is a planar graph and $r = 1$.
- There is a linear kernel $O(k)$ for dominating set on planar graph, so $2^{O(k)}n^{O(1)}$ algorithm is possible

How to place k fire stations?

- Some simplifications: Bergen is a planar graph and $r = 1$.
- There is a linear kernel $O(k)$ for dominating set on planar graph, so $2^{O(k)}n^{O(1)}$ algorithm is possible
- We show how to get *subexponential* $2^{\sqrt{k}}n^{O(1)}$ algorithms.

How to place k fire stations?

- Some simplifications: Bergen is a planar graph and $r = 1$.
- There is a linear kernel $O(k)$ for dominating set on planar graph, so $2^{O(k)}n^{O(1)}$ algorithm is possible
- We show how to get *subexponential* $2^{\sqrt{k}}n^{O(1)}$ algorithms.
- The idea works even when Bergen has more complicated structure, like embedded on a surface of bounded genus, or excluding some fixed graph as a minor; it works for every fixed $r \geq 1$, and for many other problems

Outline of the tutorial

- Framework for parameterized algorithms: combinatorial bounds + dynamic programming
- Combinatorial bounds via Graph Minor theorems
  - Bidimensionality
- Dynamic programming which uses graph structure
  - Catalan structures

Graph Minors

The framework exploits the structure of graph classes that exclude some graph as a minor

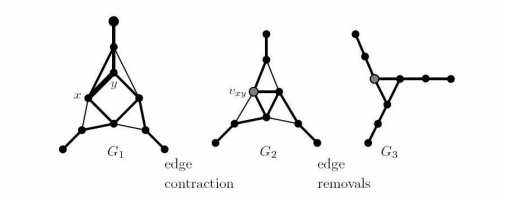
Minors and contractions

H is a contraction of G ($H \leq_c G$) if H occurs from G after applying a series of edge contractions.

H is a minor of G ($H \leq_m G$) if H is the contraction of some subgraph of G.

Notice: $\leq_m$ and $\leq_c$ are partial relations on graphs

Minors and contractions



$G_3 \preceq G_2 \preceq G_1$, $G_2 \preceq_c G_1$ but also $G_3 \not\preceq_c G_2$ and $G_3 \not\preceq_c G_1$

Minors and contractions

A graph class $\mathcal{G}$ is *minor (contraction) closed* if any minor (contraction) of a graph in $\mathcal{G}$ is again in $\mathcal{G}$.

A graph G is *H-minor-free* when it does not contain H as a minor.

A graph class $\mathcal{G}$ is *H-minor-free* (or, excludes H as a minor) when all its members are H-minor-free.

Examples of H-minor-free classes

- Forests: K_3
- Outerplanar Graphs: $K_{2,3}$, K_4
- Planar Graphs: $K_{3,3}$, K_5
- Link-free Graphs: 7 graphs (X-Y transformations of K_6)
- Graphs of the projective plane: 103 graphs

Graph Minor theorem

Robertson & Seymour (1986–2004):

Theorem (Graphs Minor Theorem)

Graphs are well-quasi-ordered by the minor relation $\leq_m$.

- Consequence: every minor closed graph class $\mathcal{G}$ has a finite set of minimal excluded minors.

Graph Minor theorem

Graphs Minor Theorem is not used in our tutorial. However,we need tools created by Roberston-Seymour in order to prof this theorem.

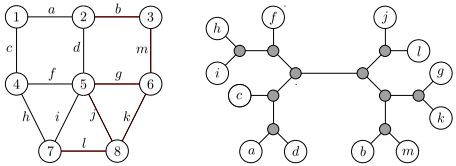
Main tool: Branch Decompositions

Definition

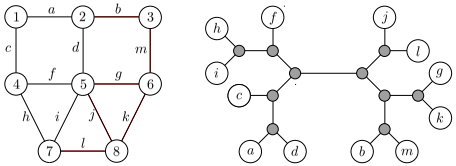
A branch decomposition of a graph $G = (V, E)$ is a tuple (T, μ) where

- T is a tree with degree 3 for all internal nodes.
- μ is a bijection between the leaves of T and $E(G)$.

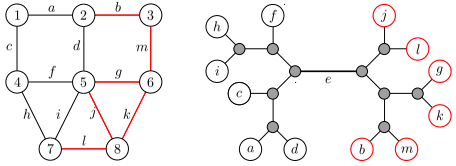
Example of Branch Decomposition



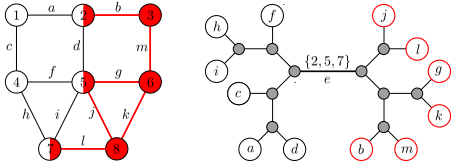
Edge $e \in T$ partitions the edge set of G in A_e and B_e



Edge $e \in T$ partitions the edge set of G in A_e and B_e



Middle set $\mathrm{mid}(e) = V(A_e) \cap V(B_e)$



Branchwidth

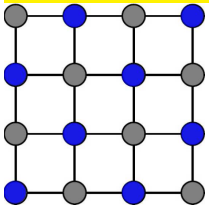
- The *width* of a branch decomposition is $\max_{e \in T} |\mathrm{mid}(e)|$.
- The *branchwidth* of a graph G is the minimum width over all branch decompositions of G.

Exercises

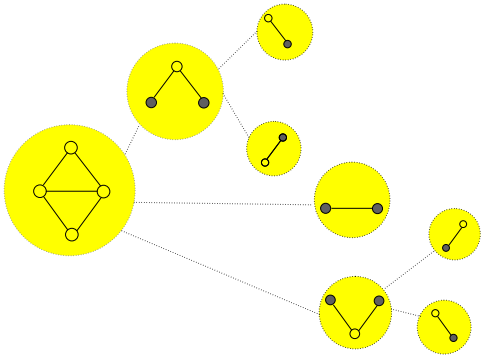
- What is the branchwidth of a tree?
- Complete graph on n vertices?
- $(\ell \times \ell)$-grid?

Vertex Cover

A *vertex cover* C of a graph G, $vc(G)$, is a set of vertices such that every edge of G has at least one endpoint in C.



Dynamic programming: Vertex Cover



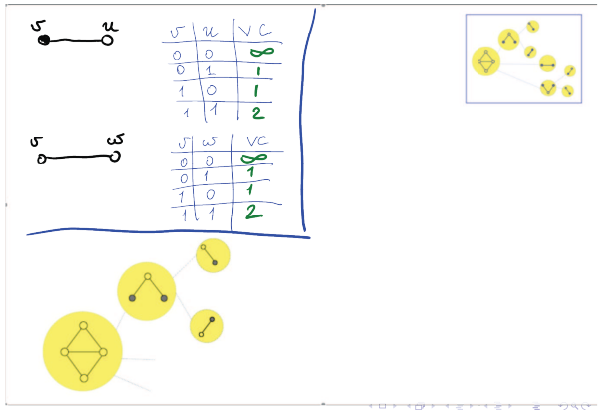
Dynamic programming: Vertex Cover

Main idea—dynamic programming.

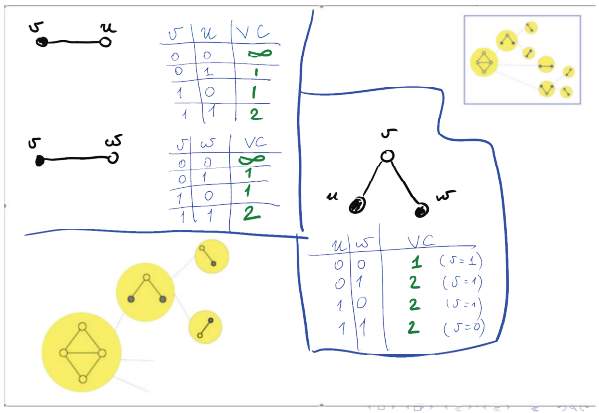
- Start from leaves, compute all possible vertex covers of each edge
- We have two branches Left and Right, and middle set M of vertices separating Left and Right. For every possible assignment A of VC for vertices M, compute

$$VALUE(\text{Left}, A) + VALUE(\text{Right}, A) - VALUE(A)$$

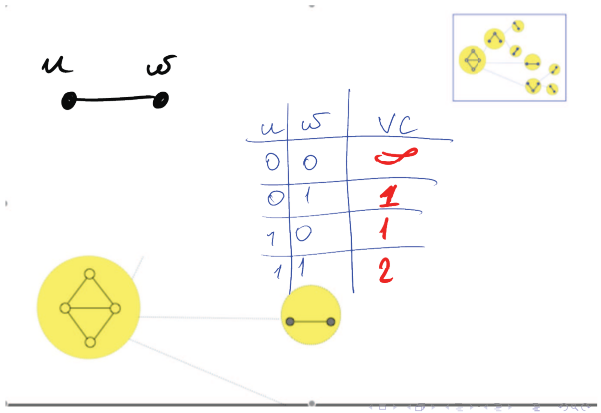
Dynamic programming: Vertex Cover



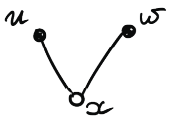
Dynamic programming: Vertex Cover



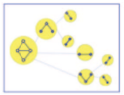
Dynamic programming: Vertex Cover

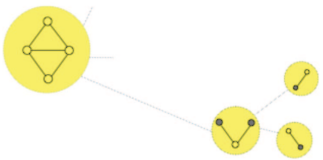


Dynamic programming: Vertex Cover

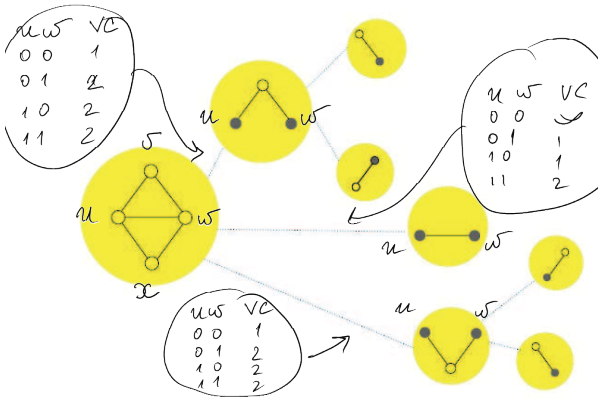


u w	VC
0 0	1
0 1	2
1 0	2
1 1	2

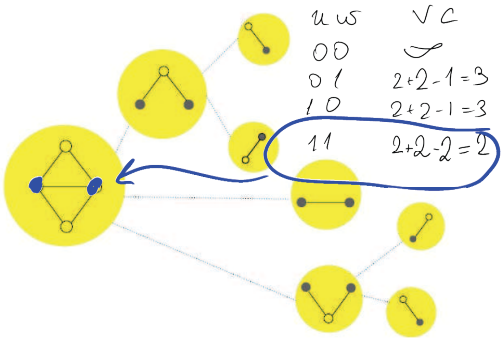




Dynamic programming: Vertex Cover



Dynamic programming: Vertex Cover



Dynamic programming: Vertex Cover

Let $\ell = \mathbf{bw}(G)$ and $m = |E(G)|$.

- Running time: size of every table for middle set is $O(2^\ell)$.
- To compute a new table: $O(2^{2\ell})$
- Number of steps $O(m)$
- Total running time: $O(2^{2\ell}m)$.

Dynamic programming: Vertex Cover

Exercise

Try to improve the running time, say to $O(2^{1.5\ell}m)$.

Grid Theorem

Theorem (Robertson, Seymour & Thomas, 1994)

Let $\ell \geq 1$ be an integer. Every planar graph of branchwidth $\geq 4\ell$ contains  *as a minor.*

Grid Theorem: Sketch of the proof

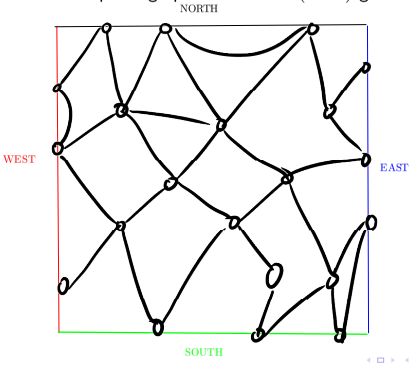
The proof is based on Menger's Theorem

Theorem (Menger 1927)

Let G be a finite undirected graph and x and y two nonadjacent vertices. The size of the minimum vertex cut for x and y (the minimum number of vertices whose removal disconnects x and y) is equal to the maximum number of pairwise vertex-disjoint paths from x to y.

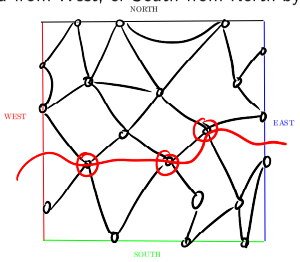
Grid Theorem: Sketch of the proof

Let G be a plane graph that has no $(\ell \times \ell)$-grid as a minor.



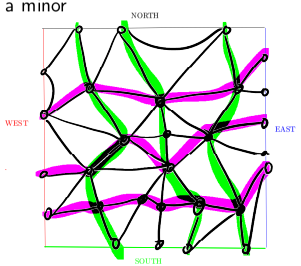
Grid Theorem: Sketch of the proof

Either East can be separated from West, or South from North by removing at most ℓ vertices



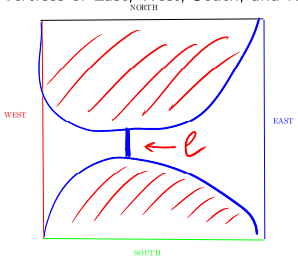
Grid Theorem: Sketch of the proof

Otherwise by making use of Menger we can construct $\ell \times \ell$ grid as a minor



Grid Theorem: Sketch of the proof

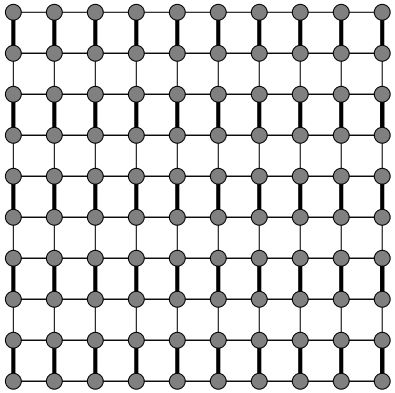
Partition the edges. Every time the middle set contains only vertices of East, West, South, and North, at most 4ℓ in total.



How to compute branchwidth

- NP-hard in general (Seymour-Thomas, Combinatorica 1994)
- On planar graphs can be computed in time $O(n^3)$ (Seymour-Thomas, Combinatorica 1994 and Gu-Tamaki, ICALP 2005)
- RST grid theorem provides 4-approximation.

We know enough to solve Vertex Cover!



$\mathbf{vc}(H_{r,r}) \geq \frac{r^2}{2}$

We know enough to solve Vertex Cover!

Let G be a planar graph of

branchwidth $\geq \ell$

We know enough to solve Vertex Cover!

Let G be a planar graph of
branchwidth $\geq \ell$

$\Longrightarrow$

G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

We know enough to solve Vertex Cover!

Let G be a planar graph of
branchwidth $\geq \ell$

$\Longrightarrow$

G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

The size of any vertex cover of H is at least $\ell^2/32$. Since H is a minor of G, the size of any vertex cover of G is at least $\ell^2/32$.

We know enough to solve Vertex Cover!

Let G be a planar graph of branchwidth $\geq \ell$ $\Longrightarrow$ G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

The size of any vertex cover of H is at least $\ell^2/32$. Since H is a minor of G, the size of any vertex cover of G is at least $\ell^2/32$.

WIN/WIN

If $k < \ell^2/32$, we say "NO"

If $k \geq \ell^2/32$, then we do DP in time $O(2^{2\ell}m) = O(2^{O(\sqrt{k})}m)$.

Challenges to discuss

- How to generalize the idea to work for other parameters?
- Does not work for Dominating Set. Why?
- Is planarity essential?
- Dynamic programming. Does MSOL helps here?

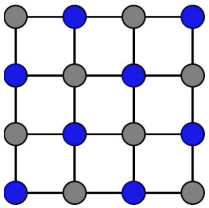
Parameters

A *parameter* P is any function mapping graphs to nonnegative integers. The *parameterized problem associated with* P asks, for some fixed k, whether for a given graph G, $P(G) \leq k$ (for minimization) and $P(G) \geq k$ (for maximization problem). We say that a parameter P is *closed under taking of minors/contractions* (or, briefly, *minor/contraction closed*) if for every graph H, $H \preceq G$ / $H \preceq_c G$ implies that $P(H) \leq P(G)$.

Examples of parameters: k-Vertex Cover

A *vertex cover* C of a graph G, $vc(G)$, is a set of vertices such that every edge of G has at least one endpoint in C. The k-Vertex Cover problem is to decide, given a graph G and a positive integer k, whether G has a vertex cover of size k.

k-Vertex Cover

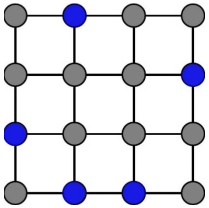


k-Vertex Cover is closed under taking minors.

Examples of parameters: k-Dominating set

A *dominating* set D of a graph G is a set of vertices such that every vertex outside D is adjacent to a vertex of D. The k-Dominating Set problem is to decide, given a graph G and a positive integer k, whether G has a dominating set of size k.

k-Dominating set



k-Dominating set is not closed under taking minors. However, it is closed under contraction of edges.

(Not exactly related to this tutorial but worth to be mentioned)

By Robertson-Seymour theory, every minor closed parameter problem is FPT.

Subexponential algorithms on planar graphs: What is the main idea?

Dynamic programming and Grid Theorem

Meta conditions

(A) For every graph $G \in \mathcal{G}$, $\mathbf{bw}(G) \leq \alpha \cdot \sqrt{P(G)} + O(1)$

(B) For every graph $G \in \mathcal{G}$ and given a branch decomposition (T, μ) of G, the value of $P(G)$ can be computed in $f(\mathbf{bw}(T, \mu)) \cdot n^{O(1)}$ steps.

Algorithm

(A) For every graph $G \in \mathcal{G}$, $\mathbf{bw}(G) \leq \alpha \cdot \sqrt{P(G)} + O(1)$

(B) For every graph $G \in \mathcal{G}$ and given a branch decomposition (T, μ) of G, the value of $P(G)$ can be computed in $f(\mathbf{bw}(T, \mu)) \cdot n^{O(1)}$ steps.

If $\mathbf{bw}(T, \mu) > \alpha \cdot \sqrt{k}$, then by (A) the answer is clear

Else, by (B), $P(G)$ can be computed in $f(\alpha \cdot \sqrt{k}) \cdot n^{O(1)}$ steps.

When $f(k) = 2^{O(k)}$, the running time is $2^{O(\sqrt{k})} \cdot n^{O(1)}$

This tutorial:

(A) For every graph $G \in \mathcal{G}$, $\mathbf{bw}(G) \leq \alpha \cdot \sqrt{P(G)} + O(1)$

(B) For every graph $G \in \mathcal{G}$ and given a branch decomposition (T, μ) of G, the value of $P(G)$ can be computed in $f(\mathbf{bw}(T, \mu)) \cdot n^{O(1)}$ steps

- How to prove (A)
- How to do (B)

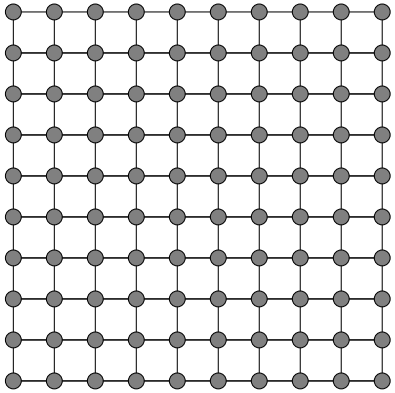
Combinatorial bounds: Bidimensionality and excluding a grid as a minor

Reminder

Theorem (Robertson, Seymour & Thomas, 1994)

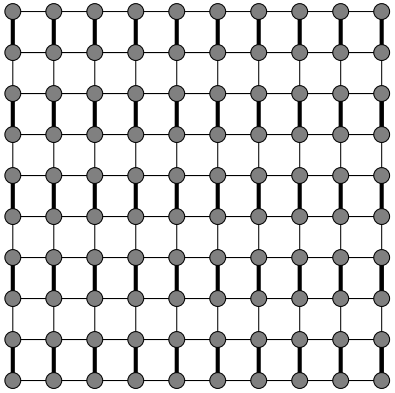
Let $\ell \geq 1$ be an integer. Every planar graph of branchwidth $\geq \ell$ contains an $(\ell/4 \times \ell/4)$-grid as a minor.

Planar k-Vertex Cover



$H_{r,r}$ for $r = 10$

Planar k-Vertex Cover



$\mathbf{vc}(H_{r,r}) \geq \frac{r^2}{2}$

Planar k-Vertex Cover

Let G be a planar graph of branchwidth $\geq \ell$

Planar k-Vertex Cover

Let G be a planar graph of branchwidth $\geq \ell$ $\Longrightarrow$ G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

Planar k-Vertex Cover

Let G be a planar graph of branchwidth $\geq \ell$ $\Longrightarrow$ G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

The size of any vertex cover of H is at least $\ell^2/32$. Since H is a minor of G, the size of any vertex cover of G is at least $\ell^2/32$.

Planar k-Vertex Cover

Let G be a planar graph of branchwidth $\geq \ell$ $\Longrightarrow$ G contains an $(\ell/4 \times \ell/4)$-grid H as a minor

The size of any vertex cover of H is at least $\ell^2/32$. Since H is a minor of G, the size of any vertex cover of G is at least $\ell^2/32$.

Conclusion: Property (A) holds for $\alpha = 4\sqrt{2}$, i.e. $\mathbf{bw}(G) \leq 4\sqrt{2}\sqrt{\mathbf{vc}(G)}$.

Planar k-Vertex Cover

Dorn, 2006: given a branch decomposition of G of width ℓ, the minimum vertex cover of G can be computed in time $f(\ell)n = 2^{\frac{\omega}{2}\ell}n$, where ω is the fast matrix multiplication constant.

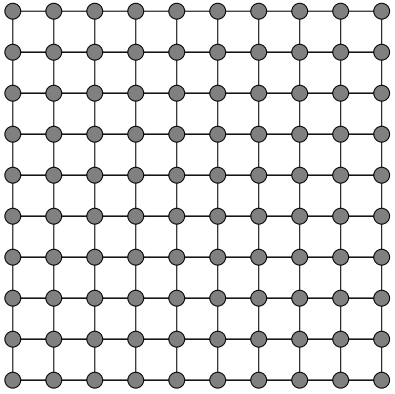
Planar k-Vertex Cover: Putting things together

- Use Seymour-Thomas algorithm to compute a branchwidth of a planar graph G in time $O(n^3)$
- If $\mathbf{bw}(G) \geq \frac{4\sqrt{k}}{\sqrt{2}}$, then G has no vertex cover of size k
- Otherwise, compute vertex cover in time $O(2^{\frac{2\omega\sqrt{k}}{\sqrt{2}}}n) = O(2^{3.56\sqrt{k}}n)$
- Total running time $O(n^3 + 2^{3.56\sqrt{k}}n)$

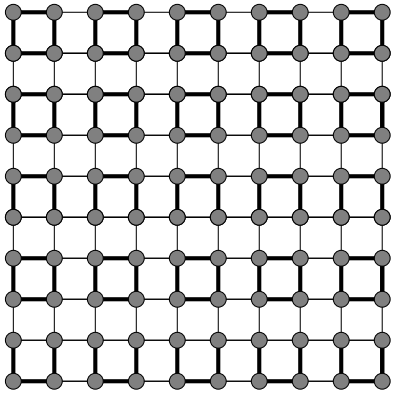
Planar k-Vertex Cover: Kernelization never hurts

- Find a kernel of size $O(k)$ in time $n^{3/2}$ (use Fellows et al. crown decomposition method)
- Use Seymour-Thomas algorithm to compute a branchwidth of the reduced planar graph G in time $O(k^3)$
- If $\mathbf{bw}(G) \geq \frac{4\sqrt{k}}{\sqrt{2}}$, then G has no vertex cover of size k
- Otherwise, compute vertex cover in time $O(2^{\frac{2\omega\sqrt{k}}{\sqrt{2}}}k) = O(2^{3.56\sqrt{k}}k)$
- Total running time $O(n^{3/2} + 2^{3.56\sqrt{k}}k)$

k-Feedback Vertex Set



k-Feedback Vertex Set



$\mathbf{fvc}(H_{r,r}) \geq \frac{r^2}{4}$

k-Feedback Vertex Set

- If $\mathbf{bw}(G) \geq r$, then $G \geq_m H_{\frac{r}{4},\frac{r}{4}}$
- $\mathbf{fvs}$ is minor-closed, therefore $\mathbf{fvs}(G) \geq \mathbf{fvs}(H_{\frac{r}{4},\frac{r}{4}}) \geq \frac{r^2}{64}$

we have that $\mathbf{bw}(G) \leq 8 \cdot \sqrt{\mathbf{fvs}(G)}$

k-Feedback Vertex Set

- If $\mathbf{bw}(G) \geq r$, then $G \geq_m H_{\frac{r}{4},\frac{r}{4}}$
- $\mathbf{fvs}$ is minor-closed, therefore $\mathbf{fvs}(G) \geq \mathbf{fvs}(H_{\frac{r}{4},\frac{r}{4}}) \geq \frac{r^2}{64}$

we have that $\mathbf{bw}(G) \leq 8 \cdot \sqrt{\mathbf{fvs}(G)}$

therefore, for p-Vertex Feedback Set, $f(k) = O(\sqrt{k})$

k-Feedback Vertex Set

- If $\mathbf{bw}(G) \geq r$, then $G \geq_m H_{\frac{r}{4},\frac{r}{4}}$
- $\mathbf{fvs}$ is minor-closed, therefore $\mathbf{fvs}(G) \geq \mathbf{fvs}(H_{\frac{r}{4},\frac{r}{4}}) \geq \frac{r^2}{64}$

we have that $\mathbf{bw}(G) \leq 8 \cdot \sqrt{\mathbf{fvs}(G)}$

therefore, for p-Vertex Feedback Set, $f(k) = O(\sqrt{k})$

Conclusion:

p-Vertex Feedback Set has a $2^{O(\log k \cdot \sqrt{k})} \cdot O(n)$ step algorithm.

Planar k-Dominating Set

Can we proceed by the same arguments with Planar k-Dominating Set?

Planar k-Dominating Set

Can we proceed by the same arguments with Planar k-Dominating Set?

Oops! Here is a problem! Dominating set is not minor closed!

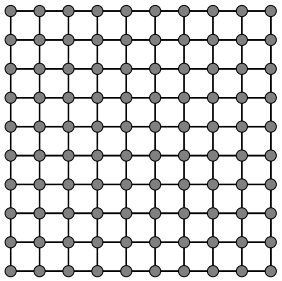
Planar k-Dominating Set

Can we proceed by the same arguments with Planar k-Dominating Set?

Oops! Here is a problem! Dominating set is not minor closed!

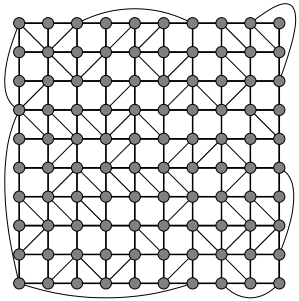
However, dominating set is closed under contraction

Planar k-Dominating Set



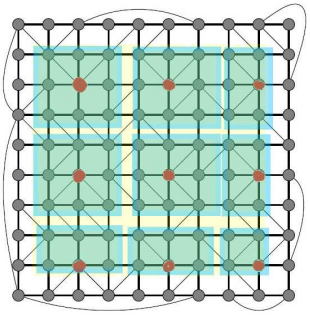
$H_{r,r}$ for $r = 10$

Planar k-Dominating Set



a partial triangulation of $H_{10,10}$

Planar k-Dominating Set



Every inner vertex of p.t. grid $\tilde{H}_{r,r}$ dominates at most 9 vertices. Thus $ds(\tilde{H}_{r,r}) \geq \frac{(r-2)^2}{9}$.

Planar k-Dominating Set

- By RST-Theorem, a planar graph G of branchwidth $\geq \ell$ can be contracted to a partially triangulated $(\ell/4 \times \ell/4)$-grid
- Since dominating set is closed under contraction, we can make the following

Conclusion: Property (A) holds for $\alpha = 12$, i.e. $\mathbf{bw}(G) \leq 12\sqrt{\mathbf{ds}(G)}$.

Planar k-Dominating Set

- By RST-Theorem, a planar graph G of branchwidth $\geq \ell$ can be contracted to a partially triangulated $(\ell/4 \times \ell/4)$-grid
- Since dominating set is closed under contraction, we conclude that Planar k-Dominating Set also satisfies property (A) with $\alpha = 12$.
- Dorn, 2006, show that for k-Dominating Set in (B), one can choose $f(\ell) = 3^{\frac{\omega}{2}\ell}$, where ω is the fast matrix multiplication constant.

Planar k-Dominating Set

- By RST-Theorem, a planar graph G of branchwidth $\geq \ell$ can be contracted to a partially triangulated $(\ell/4 \times \ell/4)$-grid
- Since dominating set is closed under contraction, we conclude that Planar k-Dominating Set also satisfies property (A) with $\alpha = 12$.
- Dorn, 2006, show that for k-Dominating Set in (B), one can choose $f(\ell) = 3^{\frac{\omega}{2}\ell}$, where ω is the fast matrix multiplication constant.
- Conclusion: Planar k-Dominating Set can be solved in time $O(n^3 + 2^{22.6\sqrt{k}}n)$

Bidimensionality: The main idea

If the graph parameter is closed under taking minors or contractions, the only thing needed for the proof branchwidth/parameter bound is to understand how this parameter behaves on a (partially triangulated) grid.

Bidimensionality: Demaine, FF, Hajiaghayi, Thilikos, 2005

Definition

A parameter P is *minor bidimensional with density* δ if

1. P is closed under taking of minors, and
2. for the $(r \times r)$-grid R, $P(R) = (\delta r)^2 + o((\delta r)^2)$.

Bidimensionality: Demaine, FF, Hajiaghayi, Thilikos, 2005

Definition

A parameter P is called *contraction bidimensional with density* δ if

1. P is closed under contractions,
2. for any partially triangulated $(r \times r)$-grid R, $P(R) = (\delta_R r)^2 + o((\delta_R r)^2)$, and
3. δ is the smallest δ_R among all paritally triangulated $(r \times r)$-grids.

Bidimensionality

Lemma

If P is a bidimensional parameter with density δ then P satisfies property (A) for $\alpha = 4/\delta$, on planar graphs.

Proof.

Let R be an $(r \times r)$-grid.

$P(R) \geq (\delta_R r)^2$.

If G contains R as a minor, then $\mathbf{bw}(G) \leq 4r \leq 4/\delta\sqrt{P(G)}$. □

Examples of bidimensional problems

Vertex cover

Dominating Set

Independent Set

(k, r)-center

Feedback Vertex Set

Minimum Maximal Matching

Planar Graph TSP

Longest Path ...

How to extend bidimensionality to more general graph classes?

- We need excluding grid theorems (sufficient for minor closed parameters)
- For contraction closed parameters we have to be more careful

Bounded genus graphs: Demaine, FF, Hajiaghayi, Thilikos, 2005

Theorem

If G is a graph of genus at most γ with branchwidth more than r, then G contains a $(r/4(\gamma+1) \times r/4(\gamma+1))$-grid as a minor.

Can we go further?

What about more general graph classes?

- How to define bidimensionality for non-planar graphs?

The grid-minor-excluding theorem gives linear bounds for H-minor free graphs:

Theorem (Demaine & Hajiaghayi, 2008)

There is a function $\phi : \mathbb{N} \to \mathbb{N}$ such that for every graph G excluding a fixed h-vertex graph H as a minor the following holds:

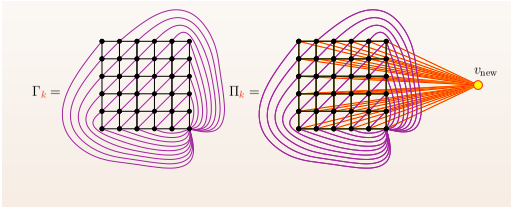
- *if* $\mathbf{bw}(G) \geq \phi(h) \cdot k$ *then* $\boxplus_k \leq_m G$.

For every minor-closed graph class a minor-closed parameter $\mathbf{p}$ is bidimensional if

$$\mathbf{p}(\boxplus_k) = \Omega(k^2)$$

What about contraction-closed parameters?

We define the following two pattern graphs Γ_k and Π_k:



$\Pi_k = \Gamma_k +$ a new vertex v_{new}, connected to all the vertices in $V(\Gamma_k)$.

The grid-minor-excluding theorem gives linear bounds for H-minor free graphs:

Theorem (Fomin, Golovach, & Thilikos, 2009)

There is a function $\phi : \mathbb{N} \to \mathbb{N}$ such that for every graph G excluding a fixed h-vertex graph H as contraction the following holds:

- *if* $\mathbf{bw}(G) \geq \phi(h) \cdot k$ *then either* $\Gamma_k \leq_c G$, *or* $\Pi_k \leq_c G$.

For contraction-closed graph class a contraction-closed parameter $\mathbf{p}$ is bidimensional if

$$\mathbf{p}(\Gamma_k) = \Omega(k^2) \text{ and } \mathbf{p}(\Pi_k) = \Omega(k^2).$$

Therefore for every apex-minor free graph class

a contraction-closed parameter $\mathbf{p}$ is bidimensional if

$\mathbf{p}($  $) = \Omega(k^2)$

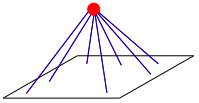
Limits of the bounded branchwidth WIN/WIN technique

As for each contraction-closed parameter $\mathbf{p}$ that we know, it holds that $\mathbf{p}(\Pi_k) = O(1)$ for all k,

Bidimensionality can be defined for apex-minor free graphs

(*apex graphs* are exactly the minors of Π_k)

H^* is an *apex graph* if

$\exists v \in V(H^*)$: $H^* - v$ is planar 

Conclusion

Minor bidimensional: minor- closed and $\mathbf{p}($  $) = \Omega(k^2)$

Contraction-bidimensional: contraction-closed and

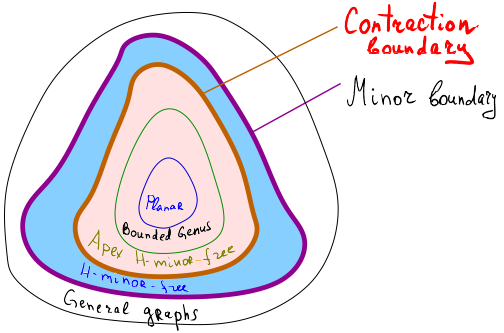
$\mathbf{p}($  $) = \Omega(k^2)$

Theorem (Bidimensionality meta-algorithm)

Let $\mathbf{p}$ be a minor (resp. contraction)-bidimensional parameter that is computable in time $2^{O(\mathbf{bw}(G))} \cdot n^{O(1)}$.

Then, deciding $\mathbf{p}(G) \leq k$ for general (resp. apex) minor-free graphs can be done (optimally) in time $2^{O(\sqrt{k})} \cdot n^{O(1)}$.

Limits of the bidimensionality



Remark

Bidimensionality cannot be used to obtain subexponential algorithms for contraction closed parameterized problems on H-minor free graphs.

For some problems, like k-Dominating Set, it is still possible to design subexponential algorithms on H-minor free graphs.

The main idea here is to use decomposition theorem of Robertson-Seymour about decomposing an H-minor free graph into pieces of apex-minor-free graphs, apply bidimensionality for each piece, and do dynamic programming over the whole decomposition.

More grids
Grids for other problems

EXAMPLE I: t-spanners (ICALP 2008, Dragan, FF, Golovach)

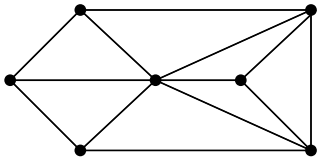
t-spanners

Definition (t-spanner)

Let t be a positive integer. A subgraph S of G, such that $V(S) = V(G)$, is called a t-spanner, if $\mathrm{dist}_S(u,v) \leq t \cdot \mathrm{dist}_G(u,v)$ for every pair of vertices u and v. The parameter t is called the *stretch factor* of S.

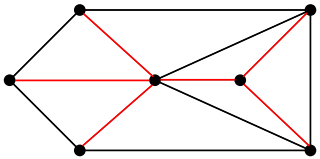
Examples of spanners

3 and 2-spanners



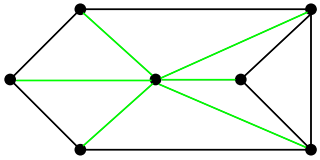
Examples of spanners

3 and 2-spanners



Examples of spanners

3 and 2-spanners



Spanners of bounded branchwidth

Problem (k-Branchwidth t-spanner)

Instance: A connected graph G and positive integers k and t.

Question: Is there a t-spanner of G of branchwidth at most k?

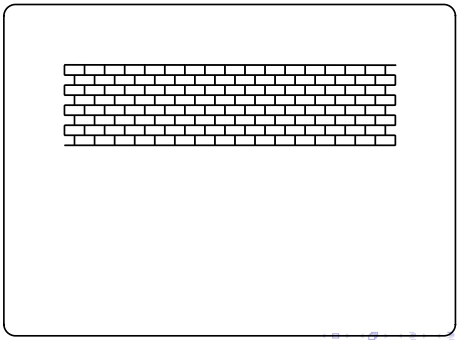
Planar graphs

Theorem (Bounds for planar graphs)

Let G be a planar graph of branchwidth k and let S be a t-spanner of G. Then the branchwidth of S is $\Omega(k/t)$.

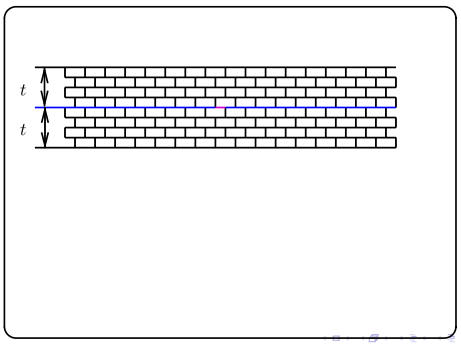
Sketch of the proof

Walls and grids



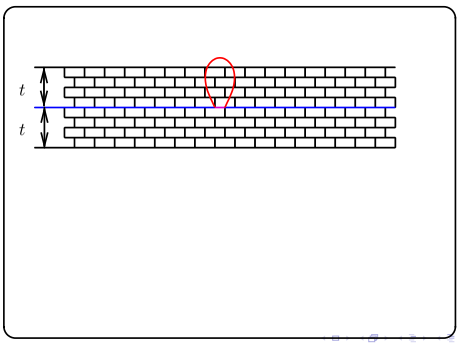
Sketch of the proof

Walls and grids



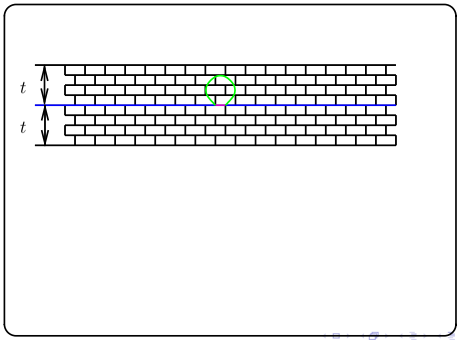
Sketch of the proof

Walls and grids



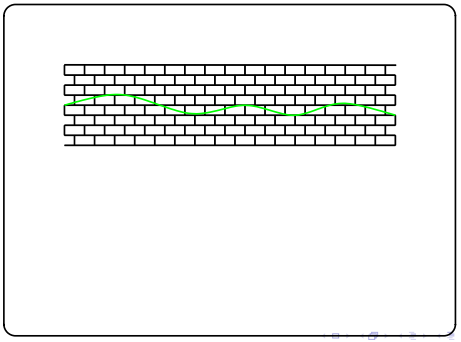
Sketch of the proof

Walls and grids



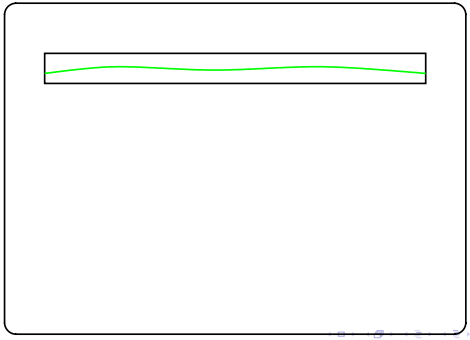
Sketch of the proof

Walls and grids



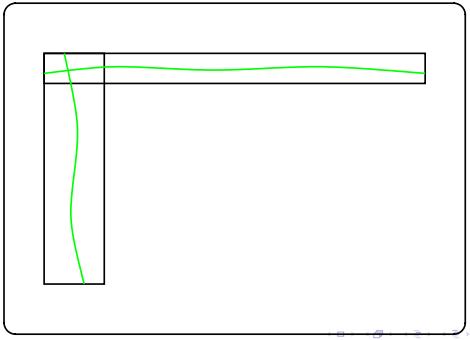
Sketch of the proof

Walls and grids



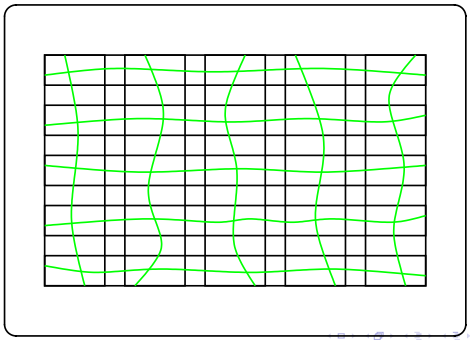
Sketch of the proof

Walls and grids



Sketch of the proof

Walls and grids



Algorithmic consequences

Theorem (Dragan, FF, Golovach, 2008)

Deciding if a planar graph G has a t-spanner of treewidth at most k is solvable in time $O(f(k,t) \cdot n)$.

Theorem (Dragan, FF, Golovach, 2008)

Let H be a fixed apex graph. For every fixed k and t, the existence of a t-spanner of treewidth at most k in an H-minor-free graph G can be decided in linear time.

Another example

Induced cycle spanning a specified set of vertices

(SODA 2009, KOBAYASHI and KAWARABAYASHI)

Induced cycle

Problem (Induced Cycle Problem)

Instance: Planar graph G and and a subset $S \subseteq V(G)$ of terminal vertices of size k.

Question: Is there an induced cycle in G containing all terminal vertices S?

Parameter k.

Algorithm sketch

- If there is a vertex which is far from each of the terminals—just remove it, it does not change the solution. (Far here means that there are $22k + 2$ nested disjoint cycles around v.)
- If every vertex is "close" to each of the terminals, then the branchwidth of the graph $O(k^{3/2})$. To prove this, one has to look at the grid!

Bidimensional theory: Conclusion

If $\mathbf{P}$ is a parameter that

(A) is minor (contraction) bidimensional

(B) can be computed in $f(\mathbf{bw}(G)) \cdot n^{O(1)}$ steps.

then there is a $f(O(\sqrt{k})) \cdot n^{O(1)}$ step algorithm for checking whether $\mathbf{P}(G) \leq k$ for H (apex) -minor free graphs.

We now fix our attention to property (B) and function f.

Dynamic programming and Catalan structures

Dynamic programming for branch decompositions

- We root the tree T of the branch decomposition (T, τ),
- We define a partial solution for each cut-set of an edge e of T
- We compute all partial solutions bottom-up (using the partial solutions corresponding to the children edges).

This can be done in $O(f(\ell) \cdot n)$ if we have a branch decomposition of width at most ℓ.

Dynamic programming for branch decompositions

- We root the tree T of the branch decomposition (T, τ),
- We define a partial solution for each cut-set of an edge e of T
- We compute all partial solutions bottom-up (using the partial solutions corresponding to the children edges).

This can be done in $O(f(\ell) \cdot n)$ if we have a branch decomposition of width at most ℓ.

$f(\ell)$ depends on the number of partial solutions we have to compute for each edge of T.

- To find a good bound for $f(\ell)$ is important!

For many problems, $2^{O(\mathbf{bw}(G))} \cdot n^{O(1)}$ step algorithms exist.

Dynamic programming on graphs with small branchwidth gives such algorithms for problems like

Vertex Cover,

Dominating Set, or

Edge Dominating Set, (and others...)

However: There are (many) problems where no general $2^{O(\mathbf{bw}(G))} \cdot n^{O(1)}$ step algorithm is known.

However: There are (many) problems where no general $2^{O(\mathbf{bw}(G))} \cdot n^{O(1)}$ step algorithm is known.

Such problems are

Longest Path, Longest Cycle, Connected Dominating Set, Feedback Vertex Set, Hamiltonian Cycle, Max Leaf Tree and Graph Metric TSP

However: There are (many) problems where no general $2^{O(\mathbf{bw}(G))} \cdot n^{O(1)}$ step algorithm is known.

Such problems are

Longest Path, Longest Cycle, Connected Dominating Set, Feedback Vertex Set, Hamiltonian Cycle, Max Leaf Tree and Graph Metric TSP

For the natural parameterizations of these problems, no $2^{O(\sqrt{k})} \cdot n^{O(1)}$ step FPT-algorithm follows by just using bidimensionality theory and dynamic programming.

Example: k-Longest Path

The k-Longest Path problem is to decide, given a graph G and a positive integer k, whether G contains a path of length k.

This problem is closed under the operation of taking minor.

Example: k-Longest Path

k-Longest Path has a $2^{O(\sqrt{k}\cdot\log k)} \cdot n^{O(1)}$ step algorithm.

Because

Example: k-Longest Path

k-Longest Path has a $2^{O(\sqrt{k}\cdot\log k)} \cdot n^{O(1)}$ step algorithm.

Because

(A) The parameter is minor bidimensional

Example: k-Longest Path

k-Longest Path has a

$2^{O(\sqrt{k}\cdot\log k)} \cdot n^{O(1)}$ step algorithm.

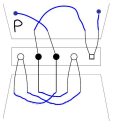
Because

(A) The parameter is minor bidimensional

(B) to find a longest path in a graph G takes

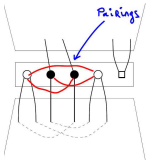
$2^{O(\mathbf{bw}(G)\cdot\log\ \mathbf{bw}(G))} \cdot n$ steps

Why log $\mathbf{bw}(G)$?



Let P be a path in G. An edge e of a branch decomposition T splits G into G_e and $G\setminus G_e$.

Why log $\mathbf{bw}(G)$?



Let P be a path in G. An edge e of a branch decomposition T splits G into G_e and $G\setminus G_e$.

- The restriction of a P to G_e is a collection $\mathcal{P}$ of internally disjoint paths in G_e with ends in $\mathrm{mid}(e)$.
- Each $\mathcal{P}$ corresponds to some pairing (a disjoint set of paths in the clique formed from $\mathrm{mid}(e)$)
- For a set S, let $\mathtt{pairs}(S)$ be the set of all pairings of S

Therefore, the complexity of dynamic programming depends on $|\mathtt{pairs}(\mathrm{mid}(e))|$, which is $\Omega(\mathbf{bw}!)$.

This obstacle does not allow to break $2^{O(\mathbf{bw}(G)\cdot\log\ \mathbf{bw}(G))} \cdot n$ barrier.

Therefore, the complexity of dynamic programming depends on $|\texttt{pairs}(\text{mid}(e))|$, which is $\Omega(\mathbf{bw}!)$.

This obstacle does not allow to break $2^{O(\mathbf{bw}(G)\cdot\log\ \mathbf{bw}(G))}\cdot n$ barrier.

- Problem: The local info in dynamic programming is too big!

Therefore, the complexity of dynamic programming depends on $|\texttt{pairs}(\text{mid}(e))|$, which is $\Omega(\mathbf{bw}!)$.

This obstacle does not allow to break $2^{O(\mathbf{bw}(G)\cdot\log\ \mathbf{bw}(G))}\cdot n$ barrier.

- Problem: The local info in dynamic programming is too big!
- Issue: The same problem appears in many dynamic programming algorithms!

Therefore, the complexity of dynamic programming depends on $|\texttt{pairs}(\text{mid}(e))|$, which is $\Omega(\mathbf{bw}!)$.

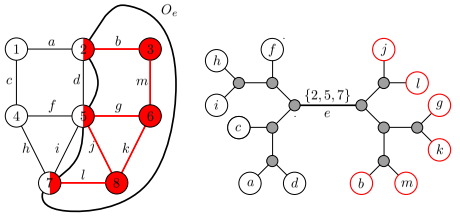
This obstacle does not allow to break $2^{O(\mathbf{bw}(G)\cdot\log\ \mathbf{bw}(G))}\cdot n$ barrier.

- Problem: The local info in dynamic programming is too big!
- Issue: The same problem appears in many dynamic programming algorithms!
- Idea: as long as we care about sparse graph classes, we can take their structure into consideration!

Sphere-cut decomposition

Let G be a planar graph embedded on the sphere $\mathcal{S}_0$

A *sphere-cut decomposition* of G is a branch decomposition (T,τ) where for every $e\in E(T)$, the vertices in $\text{mid}(e)$ are the vertices in a Jordan curve of $\mathcal{S}_0$ that meets no edges of G.



Seymour-Thomas 1994, Dorn-Penninkx-Bodlaender-FF 2005

Theorem

Every planar graph G of branchwidth ℓ has a sphere-cut decomposition of width ℓ. This decomposition can be constructed in $O(n^3)$ steps.

For doing dynamic programming on a sphere cut decomposition (T, τ) of width ℓ we define, for every $e \in E(T)$ the set $\texttt{pairs}(\mathrm{mid}(e))$ be the set of all pairings of $\mathrm{mid}(e)$

For doing dynamic programming on a sphere cut decomposition (T, τ) of width ℓ we define, for every $e \in E(T)$ the set $\texttt{pairs}(\mathrm{mid}(e))$ be the set of all pairings of $\mathrm{mid}(e)$

The "usual" bound on the size of $\texttt{pairs}(\mathrm{mid}(e))$ is $2^{O(\ell \cdot \log \ell)}$

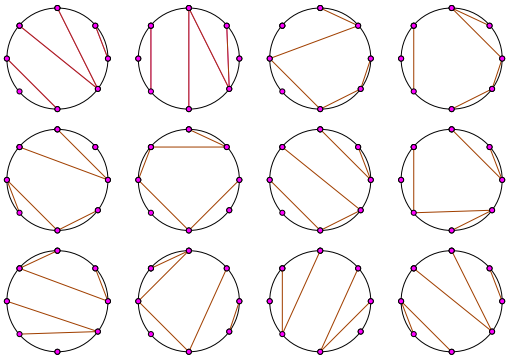
For doing dynamic programming on a sphere cut decomposition (T, τ) of width ℓ we define, for every $e \in E(T)$ the set $\texttt{pairs}(\mathrm{mid}(e))$ be the set of all pairings of $\mathrm{mid}(e)$

The "usual" bound on the size of $\texttt{pairs}(\mathrm{mid}(e))$ is $2^{O(\ell \cdot \log \ell)}$

However, we now have that

1: the vertices of $\mathrm{mid}(e)$ lay on the boundary of a disk and

2: the pairings cannot be crossing because of planarity.



It follows that $\texttt{pairs}(\text{mid}(e)) = O(C(|\text{mid}(e)|)) = O(C(\ell))$

Where $C(\ell)$ is the ℓ-th *Catalan Number*.

It is known that $C(\ell) \sim \frac{4^\ell}{\ell^{3/2}\sqrt{\pi}} = 2^{O(\ell)}$

It follows that $\texttt{pairs}(\text{mid}(e)) = O(C(|\text{mid}(e)|)) = O(C(\ell))$

Where $C(\ell)$ is the ℓ-th *Catalan Number*.

It is known that $C(\ell) \sim \frac{4^\ell}{\ell^{3/2}\sqrt{\pi}} = 2^{O(\ell)}$

Therefore: dynamic programming for finding the longest path of a planar graph G on a sphere cut decompositions of G with width $\leq \ell$ takes $O(2^{O(\ell)} \cdot n)$ steps.

It follows that $\texttt{pairs}(\text{mid}(e)) = O(C(|\text{mid}(e)|)) = O(C(\ell))$

Where $C(\ell)$ is the ℓ-th *Catalan Number*.

It is known that $C(\ell) \sim \frac{4^\ell}{\ell^{3/2}\sqrt{\pi}} = 2^{O(\ell)}$

Therefore: dynamic programming for finding the longest path of a planar graph G on a sphere cut decompositions of G with width $\leq \ell$ takes $O(2^{O(\ell)} \cdot n)$ steps.

Conclusion: [by bidimensionality]

Planar k-Longest Path can be solved in $O(2^{O(\sqrt{k})} \cdot n + n^3)$ steps

▶ The same holds for several other problems where an analogue of $\texttt{pairs}(\text{mid}(e))$ can be defined for controlling the size of the tables in dynamic programming.

▶ The same holds for several other problems where an analogue of $\texttt{pairs}(\text{mid}(e))$ can be defined for controlling the size of the tables in dynamic programming.

▶ Like that one can design $2^{O(\sqrt{k})} \cdot n^{O(1)}$ step algorithms for parameterized planar versions of Cycle Cover, Path Cover, Longest Cycle, Connected Dominating Set, Feedback Vertex Set, Hamiltonian Cycle, Graph Metric TSP, Max Leaf Tree, etc.

How to use Catalan structure in non-planar graphs?

We say that branch decomposition (T, τ) of width ℓ has the *Catalan Structure* for k-Longest Path if

$$\forall_{e \in E(T)} \ \texttt{pairs}(\text{mid}(e)) = 2^{O(\ell)}$$

How to use Catalan structure in non-planar graphs?

We say that branch decomposition (T, τ) of width ℓ has the *Catalan Structure* for k-Longest Path if

$$\forall_{e \in E(T)} \ \texttt{pairs}(\text{mid}(e)) = 2^{O(\ell)}$$

▶ We have seen that, for planar graphs, one can construct a branch decomposition with the Catalan structure for the k-Longest Path problem.

Dorn-FF-Thilikos 2008

Theorem

For any H-minor free graph class $\mathcal{G}$ there is a constant c_H (depending only on H) such that the following holds: For every graph $G \in \mathcal{G}$ and any positive integer w, it is possible to construct a $c_H \cdot n^{O(1)}$-step algorithm that outputs either

1. *a correct report that $\mathbf{bw}(G) > w$ or*
2. *a branch decomposition (T, τ) with the Catalan structure and of width $c_H \cdot w$.*

Consequences:

- For H-minor free graphs, one can construct an algorithm that solves the k-Longest Path problem in $2^{O(\sqrt{k})} \cdot n^{O(1)}$ steps.

Consequences:

- For H-minor free graphs, one can construct an algorithm that solves the k-Longest Path problem in $2^{O(\sqrt{k})} \cdot n^{O(1)}$ steps.
- Using the same result one can also solve, for H-minor free graphs, in $2^{O(\sqrt{k})} \cdot n^{O(1)}$ steps, the the standard parameterization of Longest Cycle, and Cycle/Path Cover, parameterized either by the total length of the cycles/paths or the number of the cycles/paths.

By applying modifications it is possible to define an analogue of Catalan Structure property for other problems like Feedback Vertex Set, Connected Dominating Set, and Max Leaf Tree

Proof idea: again Graph Minors

[Robertson and Seymour – GM 16]: *any H-minor free graph can roughly be obtained by identifying in a tree-like way small cliques of a collection of components that are almost embeddable on bounded genus surfaces.*

Proof idea: again Graph Minors

[Robertson and Seymour – GM 16]: *any H-minor free graph can roughly be obtained by identifying in a tree-like way small cliques of a collection of components that are almost embeddable on bounded genus surfaces.*

- Proof idea: We construct an "almost"-planarizing with certain topological properties, able to reduce the high genus "almost"-embeddings to planar ones where the planarizing vertices are "almost"-cyclically arranged in the plain.

In the plane, we use sphere cut decompositions, that permit to encode collections of paths that may pass through a separator as non crossing pairings of the vertices of a cycle.

In the plane, we use sphere cut decompositions, that permit to encode collections of paths that may pass through a separator as non crossing pairings of the vertices of a cycle.

- This provides the so-called Catalan structure of the decomposition and permits us to suitably bound the ways a path may cross its separators.

In the plane, we use sphere cut decompositions, that permit to encode collections of paths that may pass through a separator as non crossing pairings of the vertices of a cycle.

- This provides the so-called Catalan structure of the decomposition and permits us to suitably bound the ways a path may cross its separators.
- This decomposition is used to build a decomposition on the initial almost embeddible graph (following the tree-like way these components are linked together).

Open problems I

Lower bounds on dynamic programming over branchwidth. Is it possible to prove (up to some conjecture in complexity theory) that Longest Path on graphs of branchwidth ℓ cannot be solved in $2^{o(\ell \log \ell)}m$?

Open problems I

Lower bounds on dynamic programming over branchwidth. Is it possible to prove (up to some conjecture in complexity theory) that Longest Path on graphs of branchwidth ℓ cannot be solved in $2^{o(\ell \log \ell)}m$?

Can Vertex Cover be solved faster than $2^{\ell}n^{O(1)}$?

Open problems II

When applying our technique on different problems we define, for each one of them, an appropriate analogue of `pairs` and prove that it also satisfies the Catalan structure property (i.e. is bounded by $2^{O(|\mathrm{mid}(e)|)}$).

Open problems II

When applying our technique on different problems we define, for each one of them, an appropriate analogue of `pairs` and prove that it also satisfies the Catalan structure property (i.e. is bounded by $2^{O(|\operatorname{mid}(e)|)}$).

- It is challenging to find a classification criterion (logical or combinatorial) for the problems that are amenable to this approach.

Open problems III

Sufficient condition: Bidimensionality (plus fast dynamic programming) yields subexponential parameterized algorithm.

Open problems III

Sufficient condition: Bidimensionality (plus fast dynamic programming) yields subexponential parameterized algorithm.

What are the necessary conditions?

Remark: Every problem on planar graphs for which we know subexponential parameterized algorithm is either bidimensional, or can be reduced to a bidimensional problem.

Open problems IV

Branchwidth: Polynomial time algorithm for graphs of bounded genus? H-minor free graphs?

Further reading. Subexponential algorithms and bidimensionality

- J. Alber, H. L. Bodlaender, H. Fernau, T. Kloks, and R. Niedermeier, *Fixed parameter algorithms for dominating set and related problems on planar graphs*, Algorithmica, 33 (2002), pp. 461–493.
- E. D. Demaine, F. V. Fomin, M. Hajiaghayi, and D. M. Thilikos, *Subexponential parameterized algorithms on graphs of bounded genus and H-minor-free graphs*, Journal of the ACM, 52 (2005), pp. 866–893.

Further reading. Surveys

- E. Demaine and M. Hajiaghayi, *The bidimensionality theory and its algorithmic applications*, The Computer Journal, (2007), pp. 332–337.
- F. Dorn, F. V. Fomin, and D. M. Thilikos, *Subexponential parameterized algorithms*, Computer Science Review, 2 (2008), pp. 29–39.

Further reading. Catalan structures and dynamic programming

- F. Dorn, E. Penninkx, H. Bodlaender, and F. V. Fomin, *Efficient exact algorithms on planar graphs: Exploiting sphere cut branch decompositions*, Proceedings of the 13th Annual European Symposium on Algorithms (ESA 2005), vol. 3669 of LNCS, Springer, 2005, pp. 95–106.
- F. Dorn, F. V. Fomin, and D. M. Thilikos, *Catalan structures and dynamic programming on H-minor-free graphs*, in Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2008), ACM-SIAM, pp. 631–640.

Fabrizio Grandoni

A Measure & Conquer Approach for the Analysis of Exact Algorithms

A Measure & Conquer Approach for the Analysis of Exact Algorithms

Fabrizio Grandoni*

May 19, 2009

1 Introduction

The aim of *exact algorithms* is to exactly solve NP-hard problems in the smallest possible (exponential) worst-case running time. This field dates back to the sixties and seventies, and it has started to attract a growing interest in the last two decades. There are several explanations to the increasing interest in exact algorithms:

- There are certain applications that require exact solutions of NP-hard problems, although this might only be possible for moderate input sizes. This holds in particular for NP-complete decision problems.
- Approximation algorithms are not always satisfactory. Various problems are hard to approximate. For example, maximum independent set is hard to approximate within $O(n^{1-\varepsilon})$, for any constant $\varepsilon > 0$, unless $P = NP$ [25].
- A reduction of the base of the exponential running time, say from $O(2^n)$ to $O(1.9^n)$, increases the size of the instances solvable within a given amount of time by a constant *multiplicative* factor; running a given exponential algorithm on a faster computer can enlarge the mentioned size only by a (small) *additive* factor.
- The design and analysis of exact algorithms leads to a better understanding of NP-hard problems and initiates interesting new combinatorial and algorithmic challenges.

1.1 Branch & Reduce algorithms

One of the major techniques in the design of exact algorithms is *Branch & Reduce*, which traces back to the paper of Davis and Putnam [5] (see also [4]). A typical Branch & Reduce algorithm for a given problem $\mathcal{P}$ works as follows. If $\mathcal{P}$ is a *base instance*, the problems is solved directly in polynomial time. Otherwise the algorithm transforms the problem by applying a set of polynomial-time *reduction rules*. Then it branches, in polynomial-time, on two or more subproblems $\mathcal{P}_1, \ldots, \mathcal{P}_h$, according to a proper set of *branching rules*. Such subproblems are solved recursively, and the partial solutions obtained are eventually combined, in polynomial time, to get a solution for $\mathcal{P}$.

*Dipartimento di Informatica, Sistemi e Produzione, Università di Roma Tor Vergata, via del Politecnico 1, 00133 Roma, Italy, grandoni@disp.uniroma2.it.

Branch & Reduce algorithms are usually analyzed with the *bounded search tree* technique. Suppose we wish to find a time bound for a problem of size n. Assume that the depth of the search tree is polynomially bounded (which is trivially true in most cases). It is sufficient to bound the maximum number $P(n)$ of base instances generated by the algorithm: the running time will be $O^*(P(n))$[1]. If $\mathcal{P}$ is a base instance, trivially $P(n) = 1$. Otherwise, consider a possible branching step b, generating subproblems $\mathcal{P}^b_1, \ldots, \mathcal{P}^b_{h(b)}$, and let $n - \delta^b_j < n$ be the size of subproblem $\mathcal{P}^b_j$. The vector $\delta^b = (\delta^b_1, \ldots, \delta^b_{h(b)})$ is sometimes called *branching vector*. It follows that

$$P(n) \leq \sum_{j=1}^{h(b)} P(n - \delta^b_j).$$

Consider function

$$f^b(x) = 1 - \sum_{j=1}^{h(b)} x^{-\delta^b_j}.$$

This function has a unique positive root $\lambda^b = bf(\delta^b)$ (*branching factor* of δ^b). Branching factors can be easily computed numerically (see Appendix A). It turns out that $P(n) \leq \lambda^n$, where $\lambda = \max_b\{\lambda^b\}$.

We say that a branching vector δ *dominates* a branching vector δ' if $\delta \leq \delta'$, i.e. δ is component-wise not larger than δ'. It is not hard to see that, when $\delta \leq \delta'$, $bf(\delta) \geq bf(\delta')$. Hence, with respect to the running time analysis, it is sufficient to consider a dominating set of branching vectors. In other words, each time we replace the branching vector of a feasible branching with a branching vector dominating it, we obtain a pessimistic estimate of the running time. These properties will be extensively used in these notes.

1.2 Measure & Conquer

Branch & Reduce algorithms have been used for more than 40 years to solve NP-hard problems. The fastest known such algorithms are often very complicated. Typically, they consist of a long list of non-trivial branching and reduction rules, and are designed by means of a long and tedious case distinction. Despite that, the analytical tools available are still far from producing tight worst-case running time bounds for this kind of algorithms.

In these notes we present an improved analytical tool, that we called *Measure & Conquer*. In the standard analysis, n is both the measure used in the analysis and the quantity in terms of which the final time bound is expressed. However, one is free to use any, possibly sophisticated, measure m in the analysis, provided that $m \leq f(n)$ for some known function f. This way, one achieves a time bound of the kind $O^*(\lambda^m) = O^*(\lambda^{f(n)})$, which is in the desired form. The idea behind Measure & Conquer is focusing on the choice of the measure. In fact, a more sophisticated measure may capture phenomena which standard measures are not able to exploit, and hence lead to a tighter analysis of a *given* algorithm.

We apply Measure & Conquer to a toy algorithm `mis` for MIS. According to a standard analysis, the running time of this algorithm is $O^*(1.33^n)$. Thanks to a better measure, we prove that the *same* algorithm has indeed running time $O^*(1.26^n)$. This result shows that a good choice of the measure can have a tremendous impact on the time bounds achievable,

[1]Throughout this paper we use a modified big-Oh notation that suppresses all polynomially bounded factors. For functions f and g we write $f(n) = O^*(g(n))$ if $f(n) = O(g(n)poly(n))$, where $poly(n)$ is a polynomial. Also while speaking about graph problems, we use n to denote the number of nodes in the graph.

comparable to the impact of improved branching and reduction rules. Hence, finding a good measure should be at first concern when designing Branch & Reduce algorithms.

2 The Maximum Independent Set Problem

Let $G = (V, E)$ be an n-node undirected, simple graph without loops. Sometimes, we also use $V(G)$ for V and $E(G)$ for E. The (open) *neighborhood* of a node v is denoted by $N(v) = \{u \in V : uv \in E\}$, and its closed neighborhood by $N[v] = N(v) \cup \{v\}$. We let $d(v) = |N(v)|$ be the *degree* of v. By $N^x(v)$ we denote the set of nodes at distance x from v. In particular, $N^1(v) = N(v)$. Given a subset V' of nodes, $G[V']$ is the graph induced by V', and $G - V' = G[V \setminus V']$. We use $G - v$ for $G - \{v\}$.

A set $S \subseteq V$ is called an *independent set* for G if the nodes of S are pairwise non adjacent. The *independence number* $\alpha(G)$ of a graph G is the maximum cardinality of an independent set of G. The *maximum independent set* problem (`MIS`) asks to determine $\alpha(G)$.

Suppose that the considered algorithm, at a given branching or reduction step, decides that a node v belongs or does not belong to the optimum solution. In the first case we say that v is *selected*, and otherwise *discarded*.

Let us describe some simple properties of maximum independent sets.

Lemma 1 *Let G be a graph with a connected component $C \subset G$. Then*

$$\alpha(G) = \alpha(C) + \alpha(G - C).$$

Lemma 2 *Let G be a graph and v and w two nodes of G with $N[w] \subseteq N[v]$ (w* dominates *v), then*

$$\alpha(G) = \alpha(G - v).$$

Lemma 3 *Let G be a graph and v any node of G. Then there exists a maximum independent set either containing v or at least two of its neighbors $N(v)$.*

Exercise 1 *Prove Lemmas 1, 2, and 3.*

We will use the following folding operation, which is a special case of the *struction* operation defined in [6], and which was introduced in the context of exact algorithm for `MIS` in [1, 3]. A node v is *foldable* if $N(v) = \{u_1, u_2, \ldots, u_{d(v)}\}$ contains no anti-triangle[2]. *Folding* a given foldable node v of G is the process of transforming G into a new graph G_v by:

(1) adding a new node u_{ij} for each anti-edge u_iu_j in $N(v)$;

(2) adding edges between each u_{ij} and the nodes in $N(u_i) \cup N(u_j) \setminus N[v]$;

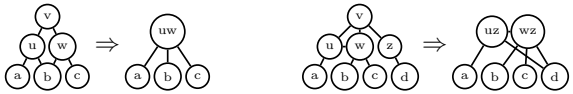
(3) adding one edge between each pair of new nodes;

(4) removing $N[v]$.

Note that nodes of degree at most two are always foldable. Examples of folding are given in Figure 1. The following simple property holds.

[2]An anti-triangle is a triple of nodes which are pairwise not adjacent. Similarly, an anti-edge is a pair of non-adjacent nodes.

Figure 1 Examples of folding.



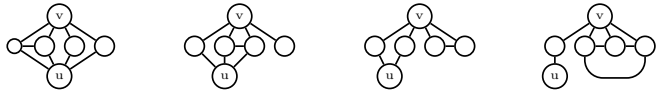
Lemma 4 *Consider a graph G, and let G_v be the graph obtained by folding a foldable node v. Then*

$$\alpha(G) = 1 + \alpha(G_v).$$

Exercise 2 *Prove Lemma 4. Hint: use Lemma 3*

We eventually introduce the following useful notion of mirror defined in [11, 13]. Given a node v, a *mirror* of v is a node $u \in N^2(v)$ such that $N(v) \setminus N(u)$ is a (possibly empty) clique. We denote by $M(v)$ the set of mirrors of v. Examples of mirrors are given in Figure 2. Intuitively, when we discard a node v, we can discard its mirrors as well without modifying

Figure 2 Example of mirrors: u is a mirror of v.



the maximum independent set size. This intuition is formalized in the following lemma.

Lemma 5 *For any graph G and for any node v of G,*

$$\alpha(G) = \max\{\alpha(G - v - M(v)), 1 + \alpha(G - N[v])\}.$$

Exercise 3 *Prove Lemma 5. Hint: use Lemma 3*

3 A Simple MIS Algorithm

In this section we describe a toy algorithm `mis` for MIS, and show that its running time is $O^*(1.33^n)$ via a standard analysis. Algorithm `mis` is described in Figure 3. When the graph is empty (base case) the algorithm simply returns 0. Otherwise it applies, when possible, the Folding Lemma 4 to a node v of degree at most 2, considering nodes of smaller degree first in case of ties:

$$\texttt{mis}(G) = 1 + \texttt{mis}(G_v).$$

As a last choice, the algorithm *greedily* takes a node v of maximum degree, and branches by either discarding v or selecting v (and discarding its neighbors)

$$\texttt{mis}(G) = max\{\texttt{mis}(G - v), 1 + \texttt{mis}(G - N[v])\}.$$

Theorem 1 *Algorithm* `mis` *solves MIS in $O^*(1.33^n)$ time.*

Figure 3 Algorithm `mis` for the maximum independent set problem.

```
int mis(G) {
    if (G = ∅) return 0; //Base case
    //Folding
    Take v of minimum degree
    if (d(v) ≤ 2) return 1 + mis(G_v);
    //"Greedy" branching
    Take v of maximum degree;
    return max{ mis(G − v), 1 + mis(G − N[v]) };
}
```

Proof. Let $P(n)$ be the number of base instances generated by the algorithm to solve an instance of size n. The depth of the recursion is $O(n)$ since in each step the number of nodes decreases at least by one. Moreover, the algorithm takes polynomial time, excluding the time needed for the recursive calls. It follows that the running time of the algorithm is $O(P(n)n^{O(1)}) = O^*(P(n))$.

We next show by induction that $P(n) \leq \lambda^n$ for $\lambda < 1.33$. In the base case $n = 0$ and $P(0) = 1 \leq \lambda^0$ for every $\lambda > 0$. When the algorithm folds a node, the number of nodes decreases at least by one. Hence, for every $\lambda \geq 1$,

$$P(n) \leq P(n-1) \leq \lambda^{n-1} \leq \lambda^n.$$

When the algorithm branches at a node v with $d(v) \geq 4$, in one subproblem it removes 1 node (i.e. v), and in the other it removes $1 + d(v) \geq 5$ nodes (i.e. $N[v]$). Let $\lambda_1 = bf(1,5) = 1.32\ldots < 1.33$ be the positive root of $1 - x^{-1} - x^{-5}$. For $\lambda \geq \lambda_1$ we obtain

$$P(n) \leq P(n-1) + P(n-5) \leq \lambda^{n-1} + \lambda^{n-5} \leq \lambda^n.$$

Otherwise, the algorithm branches at a node v of degree exactly 3, hence removing either 1 or 4 nodes. However, in the first case a node of degree 2 is folded afterwards, with the removal of at least 2 more nodes. Let $\lambda_2 = bf(3,4) = 1.22\ldots < 1.23$ be the positive root of $1 - x^{-3} - x^{-4}$. For $\lambda \geq \lambda_2$,

$$P(n) \leq P(n-3) + P(n-4) \leq \lambda^{n-3} + \lambda^{n-4} \leq \lambda^n.$$

The claim follows. □

The time bound above is the best one can achieve via a standard analysis. We will see how a non-standard analysis can provide much better time bounds.

4 A Refined Analysis via Measure & Conquer

The classical approach to *improve* on `mis` would be designing refined branching and reduction rules. In particular, one tries to improve on the *tight* recurrences. We next show how to get a much better time bound thanks to a better measure of subproblems size (without changing

Figure 4 Case analysis of folding for $m = n_{\geq 3}$.

$d(u)$	$d(w)$	$d(uw)$	m'
2	2	2	m
2	≥ 3	≥ 3	$m-1+1$
≥ 3	≥ 3	≥ 4	$m-2+1$

the algorithm!). We will start by introducing in Section 4.1 an alternative, simple, measure. This measure does not immediately give a better time bound, but it will be a good starting point to define a really better measure.

4.1 An Alternative Measure

Nodes of degree at most 2 can be removed without branching. Hence they do not really contribute to the *size* of the problem. For example, if the maximum degree is 2, then `mis` solves the problem in polynomial time! In view of that, let us define the size of the problem to be number of nodes of degree at least 3.

More formally, let n_i denote the number of nodes of degree i, and $n_{\geq i} = \sum_{j \geq i} n_j$. We define the size of the problem to be $m = n_{\geq 3}$ (rather than $m = n$). We remark that, since $m = n_{\geq 3} \leq n$, if we prove a running time bound of type $O^*(\lambda^m)$, we immediately get a $O^*(\lambda^n)$ time bound.

Let us give an alternative proof of Theorem 1.

Proof. (Theorem 1) Let us define G a base instance if the maximum degree in G is 2 (which implies $m = n_{\geq 3} = 0$). Let moreover $P(m)$ be the number of base instances generated by the algorithm to solve an instance of size m. By the usual argument the running time is $O^*(P(m))$. We prove by induction that $P(m) \leq \lambda^m$ for $\lambda < 1.33$, which implies the claim being $m \leq n$. In the base case $m = 0$. Thus

$$P(0) = 1 \leq \lambda^0.$$

Let m' be the size of the problem after folding a node v. It is sufficient to show that $m' \leq m$, from which

$$P(m) \leq P(m') \leq \lambda^{m'} \leq \lambda^m$$

for $\lambda \geq 1$. This condition trivially holds when folding only removes nodes. In the remaining case, $N(v) = \{u, w\}$ with $uw \notin E$. In this case we remove $\{v, u, w\}$ and add a node uw with $d(uw) \leq d(u) + d(w) - 2$. By case analysis (see Figure 4) $m' \leq m$ also in this case.

Suppose now that we branch at a node v with $d(v) \geq 4$. Note that all the nodes of the graph have degree ≥ 3 (since we do not fold). Hence by the standard argument

$$P(m) \leq P(m-1) + P(m-5) \leq \lambda^{m-1} + \lambda^{m-5} \leq \lambda^m.$$

Recall that the inequality above is satisfied for $\lambda \geq 1.33$.

Eventually, consider branching at v, $d(v) = 3$. In this case we remove either 1 or 4 nodes of degree 3. However, in the first case the degree of the 3 neighbors of v drops from 3 to 2, with a consequent further reduction of the size by 3:

$$P(m) \leq P(m-4) + P(m-4) \leq \lambda^{m-4} + \lambda^{m-4} \leq \lambda^m.$$

The inequality above is satisfied for $\lambda \geq bf(4,4) = 1.18\ldots$. The claim follows. □

4.2 A Better Measure

When we branch at a node of large degree, we decrement by 1 the degree of many other nodes. This is beneficial on long term, since we can remove nodes of degree at most 2 without branching. We are not exploiting this fact to its full extent in the current analysis.

An idea is then to attribute a larger *weight* $\omega_i \leq 1$ to nodes v of larger degree i, and let the size of the problem be the sum of node sizes. This way, when the degree of a node decreases, the size of the problem decreases as well. More formally, for a constant $\omega \in (0,1]$ to be fixed later, we let

$$\omega_i = \begin{cases} 0 & \text{if } i \leq 2; \\ \omega & \text{if } i = 3; \\ 1 & \text{otherwise.} \end{cases}$$

We also use $\omega(v)$ for $\omega_{d(v)}$. The size $m = m(G)$ of graph $G = (V, E)$ is

$$m = \sum_{v \in V} w(v) = \omega \cdot n_3 + n_{\geq 4}.$$

Thanks to this new measure of subproblems size, we are able to refine the analysis of `mis`.

Theorem 2 *Algorithm* `mis` *solves MIS in* $O^*(1.29^n)$ *time.*

Proof. With the usual notation, let us show that $P(m) \leq \lambda^m$ for $\lambda < 1.29$. In the base case $m = 0$, and thus

$$P(0) = 1 \leq \lambda^0.$$

In case of folding of node v, let $m' = m(G_v)$ be the size of the corresponding subproblem. Also in this case it is sufficient to show that $m' \leq m$. This condition is satisfied when nodes are only removed (being the weight increasing with the degree). The unique remaining case is $N(v) = \{u, w\}$, with u and w not adjacent. In this case we remove $\{v, u, w\}$, and add a node uw with $d(uw) \leq d(u) + d(w) - 2$. Hence it is sufficient to show that

$$\omega(v) + \omega(u) + \omega(w) - \omega(uw) = \omega(u) + \omega(w) - \omega(uw) \geq 0.$$

By a simple case analysis (see Figure 5), it follows that this condition holds for $\omega \geq 0.5$.

Consider now the case of branching at a node v, $d(v) \geq 5$. Let d_i be the degree of the ith neighbor of v (which thus has weight ω_{d_i}). Then

$$\begin{aligned} P(m) &\leq P(m - \omega_{d(v)} - \sum_{i=1}^{d(v)} (\omega_{d_i} - \omega_{d_i - 1})) + P(m - \omega_{d(v)} - \sum_{i=1}^{d(v)} \omega_{d_i}) \\ &\leq P(m - 1 - \sum_{i=1}^{5} (\omega_{d_i} - \omega_{d_i - 1})) + P(s - 1 - \sum_{i=1}^{5} \omega_{d_i}). \end{aligned}$$

Observe that we can replace $d_i \geq 6$ with $d_i = 5$. In fact in both cases $\omega_{d_i} = 1$ and $\omega_{d_i} - \omega_{d_i - 1} = 0$. Hence we can assume $d_i \in \{3, 4, 5\}$. This is crucial to obtain a finite number of recurrences! We obtain the following set of recurrences

$$P(m) \leq P(m - 1 - t_3(\omega - 0) - t_4(1 - \omega) - t_5(1 - 1)) + P(m - 1 - t_3\,\omega - t_4 - t_5),$$

Figure 5 Case analysis of folding for $m = \omega n_3 + n_{\geq 4}$.

$d(u)$	$d(w)$	$d(uw)$	$\omega(u) + \omega(w) - \omega(uw) \geq 0$
2	2	2	$0 + 0 - 0 \geq 0$
2	3	3	$0 + \omega - \omega \geq 0$
2	≥ 4	≥ 4	$0 + 1 - 1 \geq 0$
3	3	4	$\omega + \omega - 1 \geq 0$
3	≥ 4	≥ 4	$\omega + 1 - 1 \geq 0$
≥ 4	≥ 4	≥ 4	$1 + 1 - 1 \geq 0$

where t_3, t_4, and t_5 are non-negative integers satisfying $t_3 + t_4 + t_5 = 5$. (Intuitively, t_i is the number of neighbors of v of degree i).

Consider now branching at a node v, $d(v) = 4$. By a similar argument (with $d_i \in \{3, 4\}$), we obtain

$$P(m) \leq \begin{cases} P(m-1-4\cdot\omega-0\cdot(1-\omega)) + P(m-1-4\cdot\omega-0\cdot 1) \\ P(m-1-3\cdot\omega-1\cdot(1-\omega)) + P(m-1-3\cdot\omega-1\cdot 1) \\ P(m-1-2\cdot\omega-2\cdot(1-\omega)) + P(m-1-2\cdot\omega-2\cdot 1) \\ P(m-1-1\cdot\omega-3\cdot(1-\omega)) + P(m-1-1\cdot\omega-3\cdot 1) \\ P(m-1-0\cdot\omega-4\cdot(1-\omega)) + P(m-1-0\cdot\omega-4\cdot 1) \end{cases}$$

Consider eventually branching at a node v, $d(v) = 3$. By an analogous argument (with $\omega(v) = \omega_3 = \omega$ and $d_i = 3$)

$$P(m) \leq P(m - \omega - 3\,\omega) + P(m - \omega - 3\,\omega).$$

For every $\omega \in [1/2, 1]$, the set of recurrences above provides an upper bound $\lambda(\omega)$ on λ. Our goal is minimizing $\lambda(\omega)$ (hence getting a better time bound). Via exhaustive search over a grid of values for ω we obtained $\lambda(0.7) < 1.29$ (see Appendix B for a C++ program computing it). The claim follows. □

4.3 An Even Better Measure

We can extend the analysis from previous section to larger degrees. For example, we might let the weight ω_i associated to degree-i nodes be:

$$\omega_i = \begin{cases} 0 & \text{if } i \leq 2; \\ \omega & \text{if } i = 3; \\ \omega' & \text{if } i = 4; \\ 1 & \text{otherwise.} \end{cases}$$

Here ω and ω' are two proper constants, with $0 < \omega \leq \omega' \leq 1$. Using this measure, and an analysis similar to the one from previous section, it is not hard to prove the following result (see Appendix C for a C++ program optimizing the weights).

Theorem 3 *Algorithm* `mis` *solves MIS in* $O^*(1.26^n)$ *time.*

Exercise 4 *Prove the theorem above. Hint:* $\omega = 0.750$ *and* $\omega' = 0.951$.

Exercise 5 *What happens if we set* $\omega_5 = \omega''$ *for a further weight* $\omega' \leq \omega'' \leq 1$*? Do you see any pattern?*

Exercise 6 *Design a better algorithm for MIS, possibly using the other mentioned reduction rules (mirroring etc.). Analyze this algorithm in the standard way and via Measure & Conquer.*

Exercise 7 *Can you see an alternative, promising measure for MIS?*

5 Lower bounds

Despite the big improvements in the running time bounds, it might be that our refined analysis is still far from being tight. Hence, it is natural to ask for (exponential) lower bounds. Notice that we are concerned with lower bounds on the complexity of a particular algorithm, and not with lower bounds on the complexity of an algorithmic problem. A lower bound may give an idea of how far the analysis is from being tight.

In this section we prove a $\Omega(2^{n/4})$ lower bound on the running time of `mis`. The large gap between the upper and lower bound for `mis` suggests the possibility that the analysis of that algorithm can be further refined (possibly by measuring the size of the subproblems in a further refined way).

Theorem 4 *The running time of* `mis` *is* $\Omega(2^{n/4}) = \Omega(1.18^n)$.

Proof. Consider the graph G_k consisting of $k = n/4$ copies of a 4-clique (see Figure 6). We let $P(k)$ be the number of subproblems generated by `mis` to solve MIS on G_k. Consider any clique $C = \{a, b, c, d\} \in G_k$. The algorithm might branch at a. In the subproblem where a is discarded, the algorithm removes b, c, and d via folding. In the other subproblem the algorithm removes $N[a] = \{a, b, c, d\}$. Hence in both cases C is deleted from the graph, leaving an instance G_{k-1} which is solved recursively. We thus obtain the following recurrences:

$$P(k) \geq \begin{cases} 2P(k-1) & \text{if } k \geq 1; \\ 1 & \text{if } k = 0. \end{cases}$$

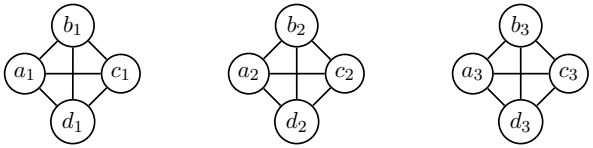
We can conclude that $P(k) \geq 2^k = 2^{n/4}$. □

Exercise 8 *Find a larger lower bound on the running time of* `mis`*. Hint:* $\Omega(3^{n/6}) = \Omega(1.20^n)$*, maybe better.*

Exercise 9 *Consider the variant of* `mis` *where the algorithm, after the base case, branches on connected components when possible. Can you find a good lower bound on the running time of this modified algorithm?*

Remark 1 *Typically finding lower bounds on connected graphs is much more complicated.*

Figure 6 Example of the lower bound graph G_k for $k = 3$.



6 Quasiconvex Analysis of Backtracking Algorithms

When the number of distinct weights grows, there is a computational problem which one has to face. In fact, both the number of recurrences and the space of candidate weights tend to grow exponentially in the number of weights. Of course the best weights need to be computed only once, and this computation has no impact on the actual behavior of the algorithm. Still, this can be a problem during the design of the algorithm, when having a quick feedback is important. In this section we outline a general way to cope with the optimization of the weights (for a given set of recurrences), described by Eppstein [8].

6.1 Multivariate Recurrences

Consider a collection of integral *measures* $m_1, \ldots, m_d$, describing different aspects of the size of the problem considered. For example, in the analysis of `mis` in Section 4.2 we used $m_1 = n_3$ and $m_2 = n_{\geq 4}$. Let $P(m_1, \ldots, m_d)$ be the number of base instances generated by the algorithm to solve a problem with measures $m_1, \ldots, m_d$. Consider a given branching step b, and let $\delta^b_{i,j}$ be the decrease of the ith measure of the jth subproblem. The following multivariate recurrence holds

$$\begin{aligned} P(m_1, \ldots, m_d) \leq\ & P(m_1 - \delta^b_{1,1}, \ldots, m_d - \delta^b_{d,1}) + \ldots \\ & + P(m_1 - \delta^b_{1,h(b)}, \ldots, m_d - \delta^b_{d,h(b)}) \end{aligned}$$

Remark 2 *Some of the $\delta^b_{i,j}$'s might be negative. For example, when we delete one edge incident to a node of degree 4, $n_{\geq 4}$ decreases but n_3 grows.*

Solving multivariate recurrences is typically rather complicated. A common alternative is turning them into univariate recurrences by considering a linear combination of the measures (*aggregated measure*)

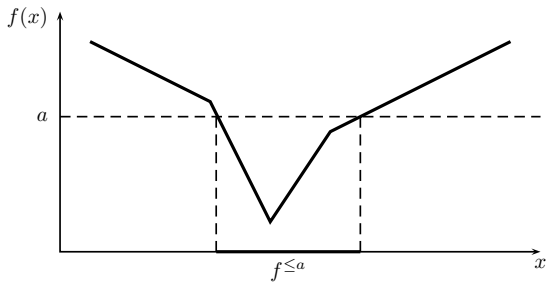
$$m(w) = w_1\, m_1 + \ldots + w_d\, m_d$$

Here $w = (w_1, \ldots, w_d)$ plays a role analogous to the weights ω_i in the analysis of `mis`. In particular, in Section 4.2 we set $w_1 = \omega \in (0, 1]$ and $w_2 = 1$.

The weights w_i are in general rational, possibly negative, numbers. However, they need to satisfy the constraint $\delta^b_j := \sum_i w_i\, \delta^b_{i,j} > 0$ for every branching b and corresponding subproblem j. In words, the aggregated measure $m(w)$ decreases in each subproblem[3]. For example, in the analysis of `mis`, this condition is satisfied for every $\omega \in [0.5, 1]$.

[3]In the degenerate case $h(b) = 1$ (a unique subproblem), $\delta^b_1 = \delta^b_{1,1} \geq 0$ is allowed (provided that the branching depth can be bounded in an alternative way).

Figure 7 Example of quasiconvex function, with a corresponding convex level set.



The resulting set of univariate recurrences can be solved in the standard way (for fixed weights). In particular, for each branching b, we compute the (unique) positive root $\lambda^b(w)$ of function

$$f^b(x,w) := 1 - \sum_{j=1}^{h(b)} x^{-\sum_{i=1}^{d} w_i \delta_{i,j}^b}.$$

This gives a running time bound of the kind $O^*(\lambda(w)^{\sum_i w_i m_i})$ where $\lambda(w) := \max_b\{\lambda^b(w)\}$.

6.2 Quasiconvexity

Function $\lambda(w)$ has a very special property, which simplifies considerably its minimization. We recall that a function $f : D \to \mathbb{R}$, with $D \subseteq \mathbb{R}^d$ convex, is *quasiconvex* if its level set

$$f^{\leq a} := \{x \in D : f(x) \leq a\}$$

is convex for any $a \in \mathbb{R}$. An example of quasiconvex (but not convex) function is given in Figure 7.

Theorem 5 *Function $\lambda(w)$ is quasiconvex over $\mathbb{R}^d$.*

Proof. The maximum over a finite number of quasiconvex functions is quasiconvex. Hence it is sufficient to show that each $\lambda^b(w)$ is quasiconvex. Recall that $\lambda^b(w)$ is the unique positive root of $f^b(x,w) = 1 - \sum_j x^{-\sum_i w_i \delta_{i,j}^b}$. Define $g^b(x,w) = \sum_j x^{-\sum_i w_i \delta_{i,j}^b}$. From the monotonicity of $f^b(x,w)$

$$\lambda^{b,\leq a} = \{w \in \mathbb{R}^d : f^b(x,w) \geq 0\} = \{w \in \mathbb{R}^d : g^b(x,w) \leq 1\} = g^{b,\leq 1}.$$

Function g^b is the sum of convex functions, and hence is convex. Then trivially its level sets, including $g^{b,\leq 1}$, are convex. □

Corollary 1 *Function $\lambda(w)$ is quasiconvex over any convex $D \subseteq \mathbb{R}^d$.*

Proof. It follows from the proof of Theorem 5, and the fact that the intersection of convex sets is convex. □

6.3 Applications to Measure & Conquer

We can use Theorem 5 to optimize the weights in a much faster way with respect to exhaustive grid search. Suppose we define a set of linear constraints on the weights such that:

(a) the size of each subproblem does not increase;

(b) $m(w) \leq n$, where n is a *standard* measure for the problem.

This gives a convex domain of weights w. On that domain we can compute the minimum value $\lambda(\tilde{w})$ of the quasiconvex function $\lambda(w)$. The resulting running time is $O^*(\lambda(\tilde{w})^n)$.

There are known techniques to find efficiently the minimum of a quasi-convex function (see e.g. [8]). We successfully applied [12, 13, 14] the following, very fast and easy to implement, approach based on *randomized local search* (in simulated annealing style):

- We start from any feasible initial value w;
- We add to w a random vector $\Delta\, w$ in a given range $[-\Delta, \Delta]^d$;
- If the resulting w' is feasible and gives $\lambda(w') \leq \lambda(w)$, we set $w = w'$;
- We iterate the process, reducing the value of Δ if no improvement is achieved for a *large* number of steps;
- The process halts when Δ drops below a given value Δ'.

Appendix D contains a C++ program applying this method to the optimization of the weights in the analysis of Section 4.3.

Remark 3 *The local search algorithm above does not guarantee closeness to the optimal $\lambda(\tilde{w})$. However it is accurate in practice. More important, it always provides* feasible *upper bounds on the running time.*

7 Other Examples of Measure & Conquer

In this section we briefly describe other (more or less explicit) applications of the Measure & Conquer approach.

The first non-trivial algorithm for the minimum dominating set problem (MDS) is based on Measure & Conquer [17, 18][4]. The basic idea is developing an algorithm for the minimum set cover problem (MSC). This algorithm is analyzed by measuring the size of the subproblems in terms of the sum of the number n of sets and number m of elements. The resulting running time is $O^*(1.381^{n+m})$. The size of the MSC formulation of a MDS instance on n nodes is $2n$. It follows that MDS can be solved in $O^*(1.381^{2n}) = O^*(1.803^n)$ time. The same algorithm is re-analyzed in [10, 13], using a refined measure which assigns different weights to sets of different size and elements of different frequency. This way the time bound is refined to $O^*(1.527^n)$, an impressive improvement.

A similar, but more complex measure is used in [12] to develop the first better-than-trivial algorithm for the connected version of MDS, where the dominating set is required to induce a connected graph. Here, besides cardinalities and frequencies, the measure takes into account the local connectivity properties of the original graph.

[4] In the same year, slower but better-than-trivial algorithms for MDS were independently developed in [15, 22].

In a paper on 3-coloring and related problems [2], Beigel and Eppstein consider a reduction to constraint satisfaction, and measure the size of the constraint satisfaction problem with a linear combination of the number of variables with three and four values in their domain, respectively. A more sophisticated measure is introduced by Eppstein in the context of cubic-TSP [7]: let F be a given set of *forced* edges, that is edges that we assume belonging to the optimum solution. For an input cubic graph $G = (V, E)$, the author measures the size of the problem in terms of $|V| - |F| - |C|$, where C is the set of 4-cycles which form connected components of $G - F$.

Gupta et al. [19] used Measure & Conquer while analyzing exact algorithms for finding maximal induced subgraphs of fixed node degree. Razgon [23], using a non-standard measure, derived the first non-trivial algorithm breaking the $O^*(2^n)$ barrier for the feedback vertex set problem (see also [9]). Kowalik [20] used Measure & Conquer in his branching algorithm for the edge coloring problem. The analysis of Gasper-Liedloff's algorithm for the independent dominating set problem in [16] is based on Measure & Conquer. Another example is the paper by Kratsch and Liedloff on the minimum dominating clique problem [21]. We are also aware of a number of other (still unpublished) papers using the same kind of approach.

Measure & Conquer can be used also as a tool to prove tighter combinatorial bounds. For example, using this kind of approach and the same measure which is mentioned above for `MDS`, Fomin et al. [14] proved that the number of minimal dominating sets in a graph is $O^*(1.721^n)$. Based on this result, they also derived the first non-trivial exact algorithms for the domatic number problem and for the minimum-weight dominating set problem. The bounds on the number of minimal feedback vertex sets (or maximal induced forests) obtained in [9] are also based on Measure & Conquer.

Of course, a non-standard measure can be used to design better algorithms in the standard way: one considers the tight recurrences for a given algorithm (and measure), and tries to design better branching and reduction rules for the corresponding cases. A very recent work by van Rooij and Bodlaender goes in this direction [24].

References

[1] R. Beigel. Finding maximum independent sets in sparse and general graphs. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 856–857, 1999.

[2] R. Beigel and D. Eppstein. 3-coloring in time O(1.3289^n). *Journal of Algorithms*, 54(2):168–204, 2005.

[3] J. Chen, I. Kanj, and W. Jia. Vertex cover: further observations and further improvements. *Journal of Algorithms*, 41:280–301, 2001.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the Association for Computing Machinery*, 5:394–397, 1962.

[5] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the Association for Computing Machinery*, 7:201–215, 1960.

[6] C. Ebenegger, P. L. Hammer, and D. de Werra. Pseudo-boolean functions and stability of graphs. *Annals of Discrete Mathematics*, 19:83–98, 1984.

[7] D. Eppstein. The Traveling Salesman Problem for Cubic Graphs. *Journal of Graph Algorithms and Applications*, 11(1):61–81, 2007.

[8] D. Eppstein. Quasiconvex analysis of backtracking algorithms. *ACM Transactions on Algorithms*, 2(4):492–509, 2006.

[9] F. V. Fomin, S. Gaspers, A. V. Pyatkin, and I. Razgon. On the Minimum Feedback Vertex Set Problem: Exact and Enumeration Algorithms. *Algorithmica*, 52(2):293–307, 2008.

[10] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: domination - a case study. In *International Colloquium on Automata, Languages and Programming (ICALP)*, pages 191–203, 2005.

[11] F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and conquer: a simple $O(2^{0.288\,n})$ independent set algorithm. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 18–25, 2006.

[12] F. V. Fomin, F. Grandoni, and D. Kratsch. Solving connected dominating set faster than 2^n. *Algorithmica,* 52(2):153–166, 2008.

[13] F. V. Fomin, F. Grandoni, and D. Kratsch. A Measure & Conquer Approach for the Analysis of Exact Algorithms. To appear in *Journal of the ACM*.

[14] F. V. Fomin, F. Grandoni, A. Pyatkin, and A. Stepanov. Combinatorial bounds via measure and conquer: Bounding minimal dominating sets and applications. *ACM Transactions on Algorithms,* 5(1): 2008.

[15] F. V. Fomin, D. Kratsch, and G. J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, pages 199–210, 2004.

[16] S. Gaspers and M. Liedloff. A branch-and-reduce algorithm for finding a minimum independent dominating set in graphs. In *Graph-Theoretic Concepts in Computer Science (WG)*, pages 78–89, 2006.

[17] F. Grandoni. *Exact Algorithms for Hard Graph Problems.* PhD thesis, Università di Roma "Tor Vergata", Roma, Italy, Mar. 2004.

[18] F. Grandoni. A note on the complexity of minimum dominating set. *Journal of Discrete Algorithms*, 4(2):209–214, 2006.

[19] S. Gupta, V. Raman, and S. Saurabh. Fast exponential algorithms for maximum -regular induced subgraph problems. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 139–151. 2006.

[20] L. Kowalik. Improved edge-coloring with three colors. In *Graph-Theoretic Concepts in Computer Science (WG)*, pages 90–101. 2006.

[21] D. Kratsch and M. Liedloff. An exact algorithm for the minimum dominating clique problem. *Theoretical Computer Science,* 385(1-3):226–240, 2007.

[22] B. Randerath and I. Schiermeyer. Exact algorithms for MINIMUM DOMINATING SET. Technical Report zaik-469, Zentrum für Angewandte Informatik, Köln, Germany, 2004.

[23] I. Razgon. Exact computation of maximum induced forest. In *Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 160–171, 2006.

[24] J. van Rooij and H. L. Bodlaender. Design by Measure and Conquer, A Faster Exact Algorithm for Dominating Set. In *Symposium on Theoretical Aspects of Computer Science,* pages 657–668, 2008.

[25] D. Zuckerman. Linear degree extractors and the inapproximability of max clique and chromatic number. In *ACM Symposium on the Theory of Computing (STOC)*, pages 681–690, 2006.

Appendix A

```
#include <cstdlib>
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>

#define PRECISION 40
#define MAXN 100

using namespace std;

/*
branchFactor() receives in input a branching vector
It returns the corresponding branching factor "bf"
This is done via doubling + binary search. The desired value satisfies
1-sum_{j=0}^{n-1} bf^(-V[i])=0
The binary search is anyway interrupted when the value is accurate enough
*/
double branchFactor(int n, double* V){

  double left=0.0;
  double right=1.0;
  double f;
  //we compute an upper bound right on the branching factor via doubling
  do{
    right = right*2;
    f=1.0;
    for(int j=0; j<n; j++) {
      f -= pow(right, -V[j]);
    }
  }while(f<=0);
  //we compute the branching factor via binary search
  double bf;
  for(int i=0; i<PRECISION; i++) {
    bf = (left+right)/2.0;
```

```
    f=1.0;
    for(int j=0; j<n; j++) {
      f -= pow(bf, -V[j]);
    }
    if(f==0) return bf;
    if(f>0) right = bf;
    else left = bf;
  }
  return right;//this way we return an upper bound
}


int main() {

  int n;
  double V[MAXN];
  double d;

  while(1) {
    cout << endl << "# of branchings:";
    cin >> n;
    if(n<=0 || n>MAXN) exit(1);

    for(int i=0; i<n; i++) {
      cout << "delta(" << i << "):";
      cin >> d;
      if(d<0) exit(1);
      V[i]=d;
    }

    double bf = branchFactor(n, V);
    printf("bf=%.8f (%.8f)\n", bf, log(bf)/log(2));
  }
  return 0;
}
```

Appendix B

```
#include <stdlib.h>
...

using namespace std;

double branchFactor(int n, double* V){
  ...
}
```

```
/*
lambda is the value of the branching factor for given (feasible) weights.
"stampa" is used to print or not some details
*/
double computeLambda(double a, int stampa) {


  double V[10];//we assume that we branch on at most 10 subproblem, which is the c
  double lambda, lambda_max= -1;

  //We don't need to consider the base case and folding

  //Branch at degree >=5
  //n3 and n4 are the neighbors of degree 3 and 4
  //n5 are the neighbors of degree at least 5
  for(int n3=0; n3<=5; n3++){
    for(int n4=0; n4<=5-n3; n4++){
      int n5 = 5-n3-n4;
      V[0]=1+n3*a+n4*(1-a)+n5*0;
      V[1]=1+n3*a+n4*1+n5*1;
      lambda = branchFactor(2, V);
      if(stampa) printf("branch at 5: 1+%d*a+%d*(1-a)+%d*0/1+%d*a+%d*1+%d*1=%.8lf\
      lambda_max = MAX(lambda_max, lambda);
    }
  }

  //Branch at degree 4
  for(int n3=0; n3<=4; n3++){
    int n4 = 4-n3;
    V[0]=1+n3*a+n4*(1-a);
    V[1]=1+n3*a+n4*1;
    lambda = branchFactor(2, V);
    if(stampa) printf("branch at 4: 1+%d*a+%d*(1-a)/1+%d*a+%d*1=%.8lf\n", n3, n4,
    lambda_max = MAX(lambda_max, lambda);
  }


  //Branch at degree 3
  V[0]=4*a;
  V[1]=4*a;
  lambda = branchFactor(2, V);
  if(stampa) printf("branch at 3: 4a/4a=%.8lf\n", lambda);
  lambda_max = MAX(lambda_max, lambda);


  if(stampa) printf("\nlambda_max(a=%.8lf)=%.8lf\n", a, lambda_max);

  return lambda_max;
}
```

```
//Here we put the constraints on the weights: in our case a\in [0.5,1.0]
bool feasible(double a){
  if(a>=0.5 && a<=1.0) return true;
  else return false;
}

int main() {

  double lambdamin = 1000;
  double amin = 1000;

  //we search for the best a in a grid with offset 1/grid
  int grid=100;
  for(int i=1; i<=grid; i++){
    double a=(double)i/grid;
    if(feasible(a)){
      double lambda = computeLambda(a,0);
      printf("lambda(a=%lf)=%.8lf\n", a, lambda);
      if(lambda < lambdamin){
        lambdamin = lambda;
        amin = a;
      }
    }
  }
  printf("\n\nBEST BOUND\n");
  computeLambda(amin,1);
//   printf("\nlambdamin(amin=%lf)=%.8lf\n", amin, lambdamin);


  system("PAUSE");
  return 0;

}
```

Appendix C

```
#include <stdlib.h>
...

using namespace std;

double branchFactor(int n, double* V){
  ...
}



/*
```

```
/*
lambda is the value of the branching factor for given (feasible) weights.
"stampa" is used to print or not some details
*/
double computeLambda(double a3, double a4, int stampa) {


  double V[10];//we assume that we branch on at most 10 subproblem, which is the c
  double lambda, lambda_max= -1;

  //We don't need to consider the base case and folding

  //Branch at degree >=6
  //Here n6 are the neighbors of degree at least 6
  for(int n3=0; n3<=6; n3++){
    for(int n4=0; n4<=6-n3; n4++){
      for(int n5=0; n5<=6-n3-n4; n5++){
        int n6 = 6-n3-n4-n5;
        V[0]=1+n3*a3+n4*(a4-a3)+n5*(1-a4)+n6*0;
        V[1]=1+n3*a3+n4*a4+n5*1+n6*1;
        lambda = branchFactor(2, V);
        if(stampa) printf("branch at 6: 1+%d*a3+%d*(a4-a3)+%d*(1-a4)+%d*0/1+%d*a3+%
                          n3, n4, n5, n6, n3, n4, n5, n6, lambda);
        lambda_max = MAX(lambda_max, lambda);
      }
    }
  }

  //Branch at degree 5
  for(int n3=0; n3<=5; n3++){
    for(int n4=0; n4<=5-n3; n4++){
      int n5 = 5-n3-n4;
      V[0]=1+n3*a3+n4*(a4-a3)+n5*(1-a4);
      V[1]=1+n3*a3+n4*a4+n5*1;
      lambda = branchFactor(2, V);
      if(stampa) printf("branch at 5: 1+%d*a3+%d*(a4-a3)+%d*(1-a4)/1+%d*a3+%d*a4+%
                        n3, n4, n5, n3, n4, n5, lambda);
      lambda_max = MAX(lambda_max, lambda);
    }
  }

  //Branch at degree 4
  for(int n3=0; n3<=4; n3++){
    int n4 = 4-n3;
    V[0]=a4+n3*a3+n4*(a4-a3);
    V[1]=a4+n3*a3+n4*a4;
    lambda = branchFactor(2, V);
    if(stampa) printf("branch at 4: a4+%d*a3+%d*(a4-a3)/a4+%d*a3+%d*a4=%.8lf\n",
                      n3, n4, n3, n4, lambda);
    lambda_max = MAX(lambda_max, lambda);
  }

  //Branch at degree 3
  int n3 = 3;
```

```
  V[0]=a3+n3*a3;
  V[1]=a3+n3*a3;
  lambda = branchFactor(2, V);
  if(stampa) printf("branch at 3: a3+%d*a3/a3+%d*a3=%.8lf\n",
                          n3, n3,lambda);
  lambda_max = MAX(lambda_max, lambda);


  if(stampa) printf("\nlambda_max(a3=%.8lf,a4=%.8lf)=%.8lf\n", a3, a4, lambda_max)

  return lambda_max;
}



//Here we put the constraints on the weights
bool feasible(double a3, double a4){
  if(a4<=1 && a3<=a4 && a3>0 && 2*a3>=a4 && 2*a4>=1 && a4>=1-a3) return true;
  else return false;
}

int main() {

  double lambda_min = 1000;
  double a3min = 1000;
  double a4min = 1000;

  //we search for the best a in a grid with offset 1/grid
  int grid=100;
  for(int i=1; i<=grid; i++){
    for(int j=1; j<=grid; j++){
      double a3=(double)i/grid;
      double a4=(double)j/grid;
      if(feasible(a3,a4)){
        double lambda = computeLambda(a3,a4,0);
        printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3, a4, lambda);
        if(lambda < lambda_min){
          lambda_min = lambda;
          a3min = a3;
          a4min = a4;
        }
      }
    }
  }
  printf("\n\nBEST BOUND\n");
  computeLambda(a3min,a4min,1);

  system("PAUSE");
  return 0;

}
```

Appendix D

```
#include <stdlib.h>
...

using namespace std;


double branchFactor(int n, double* V){
  ...
}

double uniform(double left, double right) {
  return left + (right-left)*((double)rand())/RAND_MAX;
}


double computeLambda(double a3, double a4, int stampa) {
  ...
}


bool feasible(double a3, double a4){
  ...
}

int main() {

  double lambda_min;
  double a3min;
  double a4min;

  //compute an initial random solution
  do{
    a3min = uniform(0.0,1.0);
    a4min = uniform(0.0,1.0);
  }while(!feasible(a3min,a4min));
  lambda_min = computeLambda(a3min,a4min,0);
  printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3min, a4min, lambda_min);


  double deltamin = 0.0001; //final step
  double delta = 0.01; //initial step
  int counter = 0; //measures for how long we don't make any progress
  do{
    double a3rand = a3min + delta*uniform(-1.0,1.0);
    double a4rand = a4min + delta*uniform(-1.0,1.0);
    counter++; //a new proposal is generated
    if(feasible(a3rand,a4rand)) {
      double lambda_rand = computeLambda(a3rand,a4rand,0);
      if(lambda_rand < lambda_min) {
        counter = 0;
```

```
            lambda_min = lambda_rand;
            a3min = a3rand;
            a4min = a4rand;
            printf("lambda(a3=%lf,a4=%lf)=%.8lf\n", a3min, a4min, lambda_min);
          }
        }
        if(counter >= 1000){
          counter = 0;
          delta = delta/2.0;
          printf("\ndelta=%.10f\n\n", delta);
        }
      }while(delta >= deltamin);

      printf("\n\nBEST BOUND\n");
      computeLambda(a3min,a4min,1);

      system("PAUSE");
      return 0;

}
```

Thore Husfeldt

A Taxonomic Introduction to Exact Algorithms

A Taxonomic Introduction to Exact Algorithms

Thore Husfeldt

Lecture notes for AGAPE 2009 Spring School on Fixed Parameter and Exact Algorithms, May 25-29 2009, Lozari, Corsica (France).[1]

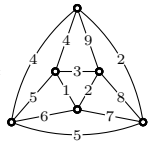
This document attempts to survey techniques that appear in exact, exponential-time algorithmics using the taxonomy developed by Levitin. The purpose is to force the exposition to adopt an alternative perspective over previous surveys, and to form an opinion of the flexibility of Levitin's framework. I have made no attempt to be comprehensive.

[1]Alpha release from 19 maj 2009. Errors are to be expected, and there are no references. Proceed with caution.

Brute force

A brute force algorithm simply evaluates the definition, typically leading to exponential running times.

TSP. For a first example, given a weighted graph like

$G =$ 

with n vertices $V = \{v_1, \ldots, v_n\}$ (sometimes called "cities") the *traveling salesman problem* is to find a shortest Hamiltonian path from the first to the last city, i.e., a path that starts at $s = v_1$, ends at $t = v_n$, includes every other vertex exactly once, and travels along edges whose total weight is minimal. Formally, we want to find

$$\min_{\pi} \sum_{i=1}^{n-1} w(\pi(i), \pi(i+1)) ,$$

where the sum is over all permutations π of $\{1, 2, \ldots, n\}$ that fix 1 and n. When the weights are uniformly 1, the problem reduces to deciding if a Hamiltonian path at all.

This above expression can be evaluated within a polynomial factor of $n!$ operations. In fact, because of certain symmetries it suffices to examine $(n-2)!$ permutations, and each of these requires take $O(n)$ products and sums. On the other hand, it's not trivial to iterate over precise these permutations in time $O((n-2)!)$. We will normally want to avoid these considerations, since they only contribute a polynomial factor, and write somewhat imprecisely $O^*(n!)$, where $O^*(f(n))$ means $O(n^c f(n))$ for some constant c.

Independent set. A subset of vertices $U \subseteq V$ in an n-vertex graph $G = (V, E)$ is *independent* if no edge from E has both its enpoints in U. Such a set can be found by considering all subsets (and checking independence of each), in time $O^*(2^n)$.

3-Satisfiability. A Boolean formula ϕ on variables $x_1, \ldots, x_n$ is on 3-conjunctive normal form if it conists of a conjunction of m *clauses*, each of the form $(a \vee b \vee c)$, where each of the *literals* a, b, c is a single variable or the negation of a single variable. The satisfiability problem for this class of formulas is to decide if ϕ admits a satisfying assignment. This can be decided by considering all assignments, in time $O^*(2^n)$. (Note that m can be assumed to be polynomial in n, otherwise ϕ would include duplicate clauses.)

FIGURE 1. A bipartite graph and 2 of its 3 perfect matchings.

Perfect matchings. A *perfect matching* in a graph $G = (V, E)$ is an edge subset $M \subseteq E$ that includes every vertex as an endpoint exactly once; in other words

$$|M| = \tfrac{1}{2}|V| \quad \bigcup M = V.$$

In fact, famously, a matching can be found in polynomial time, so we are interested in the counting version of this problem: how many perfect matchings does G admit? From the definition, this still takes $O^*(2^m)$ time.

We will look at this problem for bipartite graphs as well as for general graphs.

These are all difficult problems, typically hard for NP or # P, so we cannot expect to devise algorithms that run in polynomial time. Instead, we will improve the exponential running time. For example, for some problems we will find vertex-exponential time algorithms, i.e., algorithms with running time $\exp(O(n))$ instead of $\exp(O(m))$ or $O^*(n!)$ $O^*(n^n)$. Other algorithms will improve the base of the exponent, for example from $O^*(2^n)$ to $O(1.732^n)$.

2

Greedy

A greedy algorithm does "the obvious thing" for a given ordering, the hard part is figuring out which *ordering. A canonical example is interval scheduling.*

In exponential time, we can consider all *orderings. This leads to running times around n! and is seldom better than brute force, so this class of algorithms does not seem to play a role in exponential time algorithmics. An important exception is given as an exercise.*

3

Recursion

Recurrences express the solution to the problem in terms of solutions of subproblems. Recursive algorithms compute the solution by applying the recurrence until the problem instance is trivial.

1. Decrease and conquer

Decrease and conquer reduces the instance size by a constant, or a constant factor. Canonical examples include binary search in a sorted list, graph traversal, or Euclid's algorithm.

In exponential time, we produce several smaller instances (instead of just one), which we can use this to exhaust the search space. Maybe "exhaustive decrease and conquer" is a good name for this variant—this way, the technique becomes an umbrella of exhaustive search techniques such as branch-and-bound.

3-Satisfiability. An instance to 3-Satisfiability includes at least one clause with 3 literals. (Otherwise it's an instance of 2-Satisfiability, which can be solved in polynomial time.) Pick such a clause and construct three new instances:

T⋆⋆: set the first literal to true,
FT⋆: set the first literal to false and the second to true,
FFT: set the first two literals to false and the third to true,

These three possibilites are disjoint and exhaust the satisfying assignments. (In particular, FFF is not a satisfying assignment.)

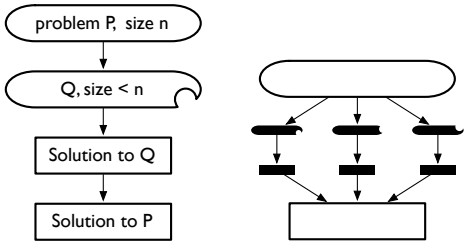


FIGURE 1. Decrease and conquer with one (left) and many (right) subproblems.

Each of these assignments resolves the clause under consideration, and maybe more, so some cleanup is required. In any case, the number of free variables is decreased by at least 1, 2, or 3, respectively. We can recurse on the three resulting three instances, so the running time satisfies

$$T(n) = T(n-1) + T(n-2) + T(n-3) + O(n+m)\,.$$

The solution to this recurrence is $O(1.8393^n)$. (The analysis of this type of algorithm is one of the most actively researched topics in exact exponential-time algorithmics and very rich.)

Independent set. Let v be a vertex of with at least three neighbours. (If no such vertex exists, the independent set problem is easy.) Construct two new instances to independent set:

- $G[V-v]$**:** the input graph with v removed. If $I \not\ni v$ is an independent set in G then it is also an independent set in $G[V-v]$.
- $G[V-N(v)]$**:** the input graph with v and its neighbours removed. If $I \ni v$ is an independent set in G, then none of v's neighbours belong to I, so that $I - \{v\}$ is an independent set in $G[V-N(v)]$.

These two possibilities are disjoint and exhaust the independent sets.

We recurse on the two resulting instances, so the running time is no worse than

$$T(n) = T(n-1) + T(n-4) + O(n+m)\,.$$

The solution to this recurrence is $O(1.3803^n)$.

TSP. Galvanized by our successes we turn to TSP.

For each $T \subseteq V$ and $v \in T$, denote by $\mathrm{OPT}(T,v)$ the minimum weight of a path from s to v that consists of exactly the vertices in T. To construct $\mathrm{OPT}(T,v)$ for all $s \in T \subseteq V$ and all $v \in T$, the algorithm starts with $\mathrm{OPT}(\{s\},s) = 0$, and evaluates the recurrence

$$\mathrm{OPT}(T,v) = \min_{u \in T \setminus \{v\}} \mathrm{OPT}(T \setminus \{v\}, u) + w(u,v)\ . \tag{1}$$

While this is correct, there is no improvement over brute force: the running time is given by

$$T(n) = n \cdot T(n-1)$$

which solves to $O(n!)$. However, we will revisit this recurrence later.

2. Divide and conquer

The divide and conquer idea partitions the instance into two smaller instances of roughly half the original size and solves them recursively. Mergesort is a canonical example.

An essential question is how *to partition the instance into smaller instances. In exponential time, we simply consider* all *such partitions. This leads to running times of the form*

$$T(n) = 2^n n^{O(1)} T\big(\tfrac{1}{2}n\big),$$

which is $O(c^n)$, and the space is polynomial in n. Maybe "exponential divide and conquer" is a good name for this idea.

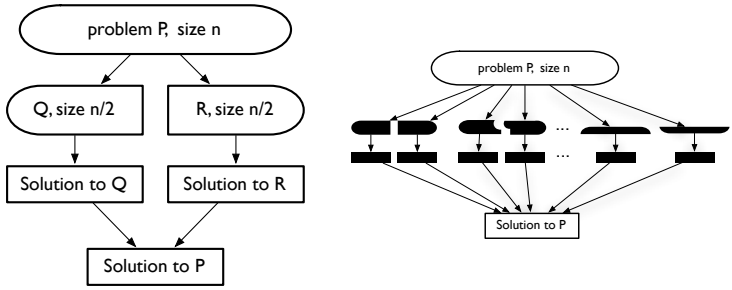


FIGURE 2. Divide and conquer with one (left) and an exponential number (right) of divisions.

TSP. Let $\mathrm{OPT}(U, s, t)$ denote the shortest path from s to t that uses exactly the vertives in U. Then we have the recurrence

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,, \tag{2}$$

where the minimum is over all subsets $S, T \subseteq U$ and vertices $m \in U$ such that $s \in S$, $t \in T$, $S \cup T = U$, $S \cap T = \{m\}$, and $|S| = \lfloor \frac{1}{2}n \rfloor + 1$, $|T| = n - |S| + 1$.

The divide and conquer solution continues using this recurrence until the sets U become trivial. At each level of the recursion, the algorithm considers $(n-2)\binom{n-2}{\lceil (n-2)/2 \rceil}$ partitions and recurses on two instances with fewer than $\frac{1}{2}n+1$ cities. Thus, the running time is

$$T(n) = (n-2) \cdot \binom{n-2}{\lceil (n-2)/2 \rceil} \cdot 2 \cdot T(n/2+1)\,,$$

which solves to $O(4^n n^{\log n})$.

The space required on each recursion level to enumerate all partitionings is polynomial. Since the recursion depth is polynomial (in fact, logarithmic) in n, so the algorithm uses polynomial space.

4

Transformation

Transformations compute a problem by computing a different problem in its stead.

1. Moebius inversion

For a function f on subsets define

$$g(X) = \sum_{Y \subseteq X} f(Y)\,.$$

Then

$$f(X) = \sum_{Y \subseteq X} (-1)^{|X \setminus Y|} g(Y)\,.$$

This result is called Moebius inversion, or, in a special case, the *principle of inclusion–exclusion*. For a proof see Dr. Kaski's presentation.

TSP. We'll do Hamiltonian path, because the idea stands cleaner.

For $X \subseteq V$ with $s, t \in X$, let $g(X)$ denote the number of walks of length n from s to t using only vertices from X. (A walk can use the same vertex many times, or once, or not at all.) Although it is not obvious, $g(X)$ can be computed in polynomial time for each $X \subseteq V$; the value is given in row s and column t of A^n, where A is the adjacency matrix of $G[X]$.

Now, let $f(X)$ denote the number of walks of length n from s to t that use exactly the vertices from X. In particular, $f(V)$ is the number of Hamiltonian paths from s to t. By Moebius inversion,

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y)\,.$$

so the number of Hamiltonian paths can be counted in time $O^*(2^n)$ and polynomial space.

To make this work for TSP, we need to handle every total distance separately, details omitted.

Perfect matchings. For $Y \subseteq V$ Let $f(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using exactly the vertices in Y (all of them); the number of perfect matchings is then given by $f(V)$. For a moment, let $g(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using only the vertices in Y (but not necessarily all of them). Then, by Moebius inversion

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y)\,.$$

Since $g(Y)$ is easy to compute for given Y, we can count the number of perfect matchings in time $O^*(2^n)$.

For bipartite graphs, we can do slightly better. So, let $V = L \cup R$ with $|L| = |R| = \frac{1}{2}n$ and assume all edges have an endpoint in L and an enpoint in R. Now, for $Y \subseteq R$ (rather than $Y \subseteq V$), let $g(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using *all* the vertices in L and *some* of the vertices in Y, and let $f(Y)$ denote the number of ways of picking $\frac{1}{2}n$ edges using *all* the vertices in L and *all* the vertices in Y. Then

$$f(V) = \sum_{Y \subseteq R} (-1)^{|R \setminus Y|} g(Y)\,,$$

in particular, the sum has only $2^{n/2}$ terms. For each $Y \subseteq R$, the value $g(Y)$ is easy enough to compute: if vertex $v_i \in L$ has d_i neighbours in R then

$$g(Y) = d_1 \cdots d_{n/2}\,.$$

Thus, the total running time is $O^*(2^{n/2}) = O(1.732^n)$. See fig. 1 for an example.

In fact, this is a famous results in combinatorics, the *Ryser formula* for the permanent, often expressed in terms of a 01-matrix A of dimension $k \times k$ as

$$\operatorname{per} A = \sum_{Y \subseteq \{1,\ldots,k\}} (-1)^{k-|Y|} \prod_{i=1}^{k} \sum_{j \in Y} A_{ij}. \tag{3}$$

The connection is that A is the upper right quarter of the adjacency matrix of G.

2. Finding triangles

The number of triangles of undirected d-vertex graph T is given by

$$\tfrac{1}{6} \operatorname{tr} A^3\,,$$

where A denotes the adjacency matrix of T and tr, the *trace*, is the sum of the diagonal entries. To see this, observe that the ith diagonal entry counts the number of paths of length 3 from the ith vertex to itself, and each triangle contributes six-fold to such entries (once for every corner, and once for every direction).

To compute $A^3 = A \cdot A \cdot A$ we need two matrix multiplications, which takes time $O(d^\omega)$, $\omega < 3$ (see Dr. Kaski's presentation).

Independent set. We want to find an independent set of size k in $G = (V, E)$, and now we assume for simplicity that 3 divides k.

Construct $G' = (V', E')$, where each vertex $v \in V'$ corresponds to an independent set in G of size $\frac{1}{3}k$. Two vertices are joined by an edge $uv \in E'$ if their corresponding sets form an independent set of size $\frac{2}{3}k$. The crucial feature is that a triangle in G' corresponds to an independent set of size k in G. The graph G' has $\binom{n}{k/3} \leq n^{k/3}$ vertices, so the whole algorithm takes time $O^*(n^{\omega k/3})$, rather than the obvious $\binom{n}{k}$.

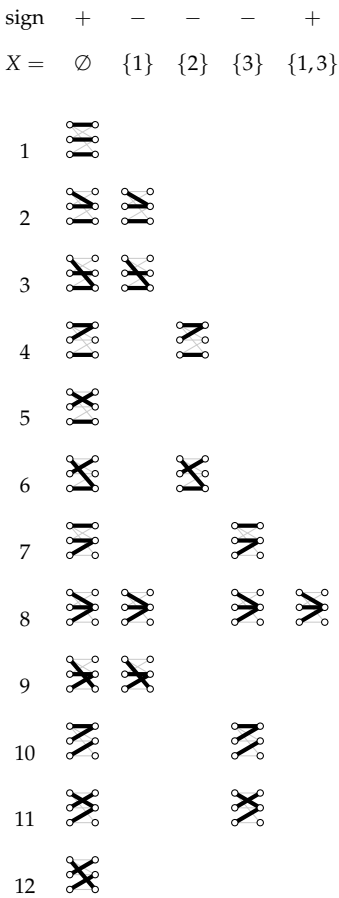


FIGURE 1. The input graph has three perfect matchings, in columns 1, 5, and 12. The first row shows all $12 = 3 \cdot 2 \cdot 2$ ways to map the left vertices to the right. Every row of the table shows the mappings that avoid various vertex subsets X, drawn as $\circ$. We omit the rows whose contribution is zero, like $X = \{1,2\}$, $X = \{2,3\}$ and $X = \{1,2,3\}$. Of particular interest is column 8, which is subtracted twice and later added again. The entire calculation is $12 - 4 - 2 - 4 + 1 + 0 + 0 - 0$, with is indeed 3.

Perfect matchings. The next example is somewhat more intricate, and uses both transformations from this section.

We return to perfect matchings, but now in regular graphs. Let $G[n = r; m = k]$ denote the number of induced subgraphs of G with r vertices and k edges. For such a graph, the number of ways to pick $\frac{1}{2}n$ edges is $k^{n/2}$, so we can rewrite

$$f(V) = \sum_{Y \subseteq V} (-1)^{|V \setminus Y|} g(Y) = \sum_{k=1}^{m} \sum_{r=2}^{n} (-1)^r G[n = r; m = k] k^{n/2} .$$

Thus, we have reduced the problem to computing $G[n = r; m = k]$ for given r and k, and we'll now do this faster than in the obvious 2^n iterations.

We are tempted to do the following: Construct a graph T where every vertex corresponds to a subgraph of G induced by a vertex subset $U \subseteq V$ with $\frac{1}{3}r$ vertices and $\frac{1}{6}k$ edges. Two vertices in T are joined by an edge if there are $\frac{1}{6}k$ edges between their corresponding vertex subsets. Then we would like to argue that every triangle in T corresponds to an induced subgraph of G with r edges and k edges. This, of course, doesn't quite work because (1) the three vertex subsets might overlap and (2) the edges do not necessarily partition into such six equal-sized families. Once identified, these problems are easily adressed.

The construction is as follows. Partition the vertices of G into three sets V_0, V_1, and V_2 of equal size, assuming 3 divides n for readability. Our plan is to build a large tripartite graph T whose vertices correspond to induced subgraphs of G that are entirely contained in one the V_i.

Some notation: An induced subraph of G has r_1 vertices in V_1, k_1 edges with both endpoints in V_1, and k_{12} edges between V_1 and V_2. Define r_2, r_3, k_2 , k_3, k_{23}, and k_{13} similarly. We will solve the problem of computing $G[n = r; m = k]$ separately for each choice of these parameters such that $r_1 + r_2 + r_3 = r$ and $k_1 + k_2 + k_3 + k_{12} + k_{23} + k_{13} = k$. We can crudely bound the number of such new problems by $n^3 + m^6$, i.e., a polynomial in the input size.

The tripartite graph T is now defined as follows: There is a vertex for every induced subgraph $G[U]$, provided that U is entirely contained in one of the V_i, and contains exactly r_i vertices and k_i edges. An edge joins the vertices corresponding to $U_i \subseteq V_i$ and $U_j \subseteq V_j$ if $i \neq j$ and there are exactly k_{ij} edges between U_i and U_j in G. The graph T has at most $3 \cdot 2^{n/3}$ vertices and $3 \cdot 2^{2n/3}$ edges. Every triangle in T uniquely corresponds to an induced subgraph $G[U_1 \cup U_2 \cup U_3]$ in G with the parameters described in the previous paragraph.

The total running time is $O^*(n^{\omega k/3}) = (1.732^n)$.

5

Iterative improvement

Iterative improvement plays a vital role in efficient algorithms and includes important ideas such as the augmentating algorithms used to solve maximum flow and bipartite matching algorithms, the Simplex method, and local search heuristics. So far, very few of these ideas have been explored in exponential time algorithmics.

1. Local search

3-Satisfiability. Start with a random assignment to the variables. If all clauses are satisfied, we're done. Otherwise, pick a falsified clause uniformly at random, pick one of its literals unformly at random, and negate it. Repeat this local search step $3n$ times. After that, start over with a fresh random assignment. This proces finds a satisfying assignment (if there is one) in time $O^*\left((\frac{4}{3})^n\right)$ with high probability.

The analysis considers the number d of differences between the current assignment A and a particular satisfying assignment A^* (the Hamming distance). In the local search steps, the probability that the distance is decreased by 1 is at least $\frac{1}{3}$ (namely, when we pick exactly the literal where A and A^* differ), and the probability that the distance is increased by 1 is at most $\frac{2}{3}$. So we can pessimistically estimate the probability $p(d)$ of reducing the distance to 0 when we start at distance d ($0 \leq d \leq n$) by standard methods from the analysis of random walks in probability theory to

$$p(d) = 2^{-d}.$$

(Under the rug one finds an argument that we can safely terminate this random walk after $3n$ steps without messing up the analysis too much.)

The probability that a 'fresh' random assignment has distance d to A^* is

$$\binom{n}{d} 2^{-n},$$

so the total probabilty that the algorithm reaches A^* from a random assignment is at least

$$\sum_{d=0}^{n} \binom{n}{d} 2^{-n-d} = \frac{1}{2^n} \sum_{d=0}^{n} \binom{n}{d} 2^{-d} = \frac{1}{2^n}(1+\tfrac{1}{2})^n = (\tfrac{3}{4})^n.$$

Especially, in expectation, we can repeat this proces and arrive at A^* or some other satisfying assignment after $(\frac{4}{3})^n$ trails.

6

Time–Space tradeoffs

Time–space tradeoffs avoid redundant computation, typically "recomputation," by storing values in large tables.

1. Dynamic programming over the subsets

Dynamic programming consists of describing the problem (or a more general form of it) recursively in an expression that involves only few varying parameters, and then compute the answer for each possible value of these parameters, using a table to avoid redundant computation. A canonical example is Knapsack.

In exponential time, the dynamic programme can consider all subsets (of vertices, for example). This is, in fact, one of the earliest applications of dynamic programming, dating back to Bellman's original work in the early 1960s.

TSP. We'll solve TSP in $O(2^n n^2)$, a bound that is still the best known. We go back to the decrease and conquer recurrence

$$\mathrm{OPT}(T, v) = \min_{u \in T \setminus \{v\}} \mathrm{OPT}(T \setminus \{v\}, u) + w(u, v) \ .$$

The usual dynamic programming trick kicks in: The values $\mathrm{OPT}(T, v)$ are stored a table when they are computed to avoid redundant recomputation, an idea sometimes called *memoisation*. The space and time requirements are within a polynomial factor of 2^n, the number of subsets $T \subseteq V$. Figure 2 shows the first few steps.

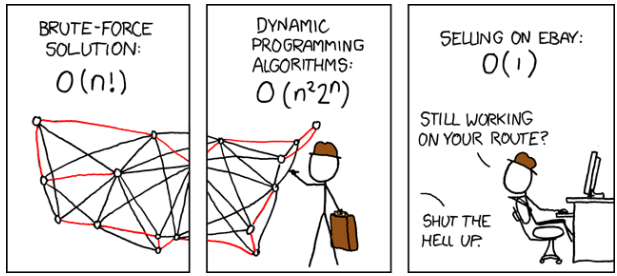


FIGURE 1. The dynamic programming algorithm for TSP mentioned in *xkcd* nr. 399.

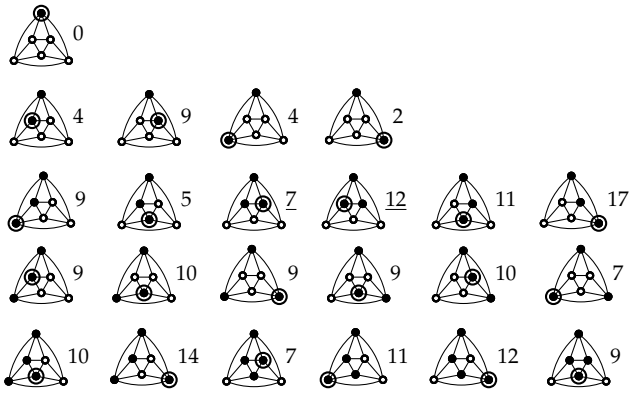


FIGURE 2. The first few steps of filling out a table for $\mathrm{OPT}(T, v)$ for the example graph. The starting vertex s is at the top, v is circled, and T consists of the black vertices. At this stage, the values of $\mathrm{OPT}(T, v)$ have been computed for all $|T| \leq 3$, and we just computed the value 9 at the bottom right by inspecting the two underlined cases. The "new" black vertex has been reached either via a weight 2 edge, for a total weight of $2 + 7$, or via a weight 1 edge for a total weight of $12 + 1$. The optimum value for this subproblem is 9.

It is instructive to see what happens if we start with the divide and conquer recurrence instead:

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,;$$

recall that S and T are a balanced vertex partition of U. We build a large table containing the value of $\mathrm{OPT}(X, u, v)$ for all vertex subsets $X \subseteq V$ and all pairs of vertives u, v. This table has size $2^n n^2$, and the entry corresponding to a subset X of size k can be computed by accessing 2^k other table entries corresponding to smaller sets. Thus, the total running time is within a polynomial factor of

$$\sum_{k=0}^{n} \binom{n}{k} 2^k = (2+1)^n = 3^n.$$

We observe that the benefit from memoisation is smaller compared to the decrease and conquer recurrence, which spent more time in the recursion ("dividing") and less time assembling solutions ("conquering").

2. Dynamic programming over a tree decomposition

The second major application of dynamic programming is over the tree decomposition of a graph. We don't cover that here, for lack of time.

3. Meet in the middle

TSP. If the input graph is 4-regular (i.e., every vertex has exactly 4 neighbours), it makes sense to enumerate the different Hamiltonian paths by making one of three choices at every vertex, for a total of at most $O^*(3^n)$ paths, instead of considering the $O^*(n!)$ different permutations. Of course, the dynamic programming solution is still faster, but we can do even better using a different time–space trade-off.

We turn again to the "divide and conquer" recurrence,

$$\mathrm{OPT}(U, s, t) = \min_{m,S,T} \mathrm{OPT}(S, s, m) + \mathrm{OPT}(T, m, t)\,.$$

This time we evaluate it by building a table for all choices of m and $T \ni t$ with $|T| = n - \lfloor \frac{1}{2}n \rfloor$. No recursion is involved, we brutally check all paths from m to t of length $|T|$, in time $O^*(3^{n/2})$. After this table is completed we iterate over all choices of $S \ni s$ with $|S| = \lfloor \frac{1}{2}n \rfloor + 1$ the same way, using $3^{n/2}$ iterations. For each S and m we check our dictionary for the entry corresponding to m and $V - T$.

It is instructive to compare this idea to the dynamic programming approach. There, we used the recurrence relation at every level. Here, we use it only at the top. In particular, the meet-in-the-middle idea is qualitatively different from the concept of using memoisation to save some overlapping recursive invocations.

4. Fast Transforms

→ Dr. Kaski

5. Fast Matrix multiplication

→ Dr. Kaski

7

Exercises

An graph can be *k-coloured* if each vertex can be coloured with one of k different colours such that no edge connects vertices of the same colour.

This set of exercises asks you do solve the k-colouring problem in various ways for a graph with n vertices and m edges

Exercise 1. Using brute force, in time $O^*(n^k)$.

Exercise 2. Using a greedy algorithm, in time $O^*(n!)$.

Exercise 3. Using decrease-and-conquer, in time in time $O^*(((1+\sqrt{5})/2)^{n+m})$. *Hint*: That's the solution to the "Finonacci" recurrence $T(s) = T(s-1) + T(s-2)$.

Exercise 4. Using divide-and-conquer, in time $O^*(9^n)$.

Exercise 5. Using Moebius inversion, in time $O^*(3^n)$. *Hint:* $\sum_{i=0}^{n} \binom{n}{i} 2^i = (2+1)^n$.

Exercise 6. Using dynamic programming over the subsets, in time $O^*(3^n)$.

Exercise 7. Using Yates's algorithm and Moebius inversion, in time $O^*(2^n)$.

Petteri Kaski

Linear and bilinear transformations for moderately exponential algorithms

LINEAR AND BILINEAR TRANSFORMATIONS FOR MODERATELY EXPONENTIAL ALGORITHMS (LECTURE NOTES FOR AGAPE 09)

PETTERI KASKI

1. Introduction

Many basic tasks in algebra can be reduced to the problem of evaluating a set of linear or bilinear forms over a ring R (for example, over the integers).

In the linear case, the task is to evaluate

$$y_i = \sum_{j=1}^{n} a_{ij} x_j \quad \text{for } 1 \leq i \leq m, \tag{1}$$

where the a_{ij} are coefficients in R, and the x_j are elements of R given as input.

In the bilinear case, the task is to evaluate

$$z_i = \sum_{j=1}^{n} \sum_{k=1}^{s} b_{ijk} x_j y_k \quad \text{for } 1 \leq i \leq m, \tag{2}$$

where the b_{ijk} are coefficients in R, and the x_j and y_k are elements of R given as input.

Here the coefficients a_{ij} and b_{ijk} are understood to be static (that is, part of the problem definition) and not part of the formal input. Put otherwise, (1) is essentially the task of multiplying a given vector with an implicit $m \times n$ matrix, and (2) asks us to evaluate a bilinear product of given two vectors. For example, taking the average $\frac{1}{n} \sum_{j=1}^{n} x_j$ reduces to (1) with $m = 1$, and the inner product $\sum_{j=1}^{n} x_j y_j$ reduces (2) with $m = 1$ and $n = s$.

Exercise 1. *Express the task of multiplying two complex numbers, $x_1 + x_2\Im$ and $y_1 + y_2\Im$, as the task of evaluating a set of two bilinear forms.*

Exercise 2. *Express the task of multiplying two polynomials of degrees p and q, respectively, as the task of evaluating a set of $p + q + 1$ bilinear forms.*

Exercise 3. *Express the task of multiplying two matrices of sizes $p \times q$ and $q \times r$, respectively, as the task of evaluating a set of $p \cdot r$ bilinear forms.*

In this lecture we study algorithms based on "faster-than-obvious" evaluation strategies for specific linear and bilinear transformations. Our measure of efficiency in this setting is the number of basic arithmetic operations in the ring R.

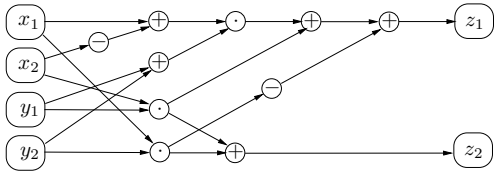
Exercise 4. *The obvious way to multiply two complex numbers requires four real multiplications:*

$$(x_1 + x_2\Im)(y_1 + y_2\Im) = x_1 y_1 - x_2 y_2 + (x_1 y_2 + y_1 x_2)\Im.$$

Show that three real multiplications suffice.

In many cases it is possible to view an evaluation strategy as an *R-arithmetic circuit* that transforms a given input into the desired output via a circuit of (i) arithmetic gates (addition, negation, multiplication) and (ii) constant gates taking values in R. We assume that all arithmetic gates have a fan-in of at most two.

Example 5. *A real arithmetic circuit for multiplying two complex numbers with three multiplications.*

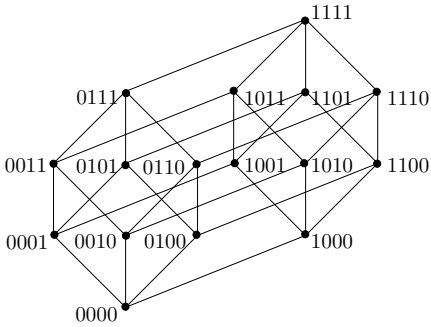


Note that multiplication in an arbitrary ring R need not be commutative, in which case one must explicitly indicate the left and right inputs to a multiplication gate.

2. Two examples of fast evaluation

2.1. **Yates's algorithm.** The n-dimensional binary hypercube (the n-cube) is the graph whose vertices are the binary strings $s = (s_1, s_2, \ldots, s_n) \in \{0,1\}^n$, and any two vertices are joined by an edge if and only if they differ in exactly one position.

Example 6. *The n-cube for $n = 4$.*



Consider the following task. Let $k : \{0,1\}^2 \to R$ and $x : \{0,1\}^n \to R$ be given as input. We must output the function $y : \{0,1\}^n \to R$, defined by

$$y(t) \stackrel{\text{def}}{=} \sum_{s \in \{0,1\}^n} \left(\prod_{i=1}^{n} k(t_i, s_i) \right) x(s) \quad \text{for } t \in \{0,1\}^n. \tag{3}$$

Observe that (3) in fact asks us to evaluate a set of 2^n linear forms, one for each vertex t of the n-cube. Thus, we are looking at an instance $y(t) = \sum_s a(t,s)x(s)$ of (1), where the coefficients $a(t,s) = \prod_{i=1}^{n} k(t_i, s_i)$ are determined by the auxiliary input k.

A direct evaluation of (3) takes $O(4^n n)$ ring operations.

To arrive at more efficient evaluation, one possibility is to view $\prod_{i=1}^{n} k(t_i, s_i)$ in (3) as the "weight" of a "walk" from s to t in the n-cube, where each step $s_i \mapsto t_i$ contributes the weight $k(t_i, s_i)$. In particular, we can view $y(t)$ as the weighted sum of "messages" $x(s)$ transmitted along walks to t. A direct evaluation of (3) corresponds to considering each individual walk separately, but it turns out that the walks can be processed in aggregate using dynamic programming.

Let us make more precise the notion of a "walk" in this context. Let s and t be any two vertices of the n-cube. The *walk* from s to t is a sequence of n steps, where step $i = 1, 2, \ldots, n$ consists of the rule $s_i \mapsto t_i$ applied to position i.

Example 7. *The walk from* $s = (0, 1, 1, 0)$ *to* $t = (1, 1, 0, 0)$ *is* $(0 \mapsto 1, 1 \mapsto 1, 1 \mapsto 0, 0 \mapsto 0)$. *The sequence of vertices visited by the walk appears below.*

$$\begin{array}{llllllll} 0 & (0 \mapsto 1) & 1 & & 1 & & 1 & & 1 \\ 1 & & 1 & (1 \mapsto 1) & 1 & & 1 & & 1 \\ 1 & & 1 & & 1 & (1 \mapsto 0) & 0 & & 0 \\ 0 & & 0 & & 0 & & 0 & (0 \mapsto 0) & 0 \end{array}$$

Observe that a walk uniquely determines the sequence of vertices it visits.

Exercise 8. *Conclude that for any vertex* $u \in \{0,1\}^n$ *there are exactly* 2^n *walks that are at* u *after exactly* j *steps,* $0 \le j \le n$.

The intuition in the following algorithm is that $z_j(u)$ contains an aggregate of the walks that are *at* u *after exactly* j *steps*. In particular, these walks originate from the vertices $(s_1, \ldots, s_j, u_{j+1}, \ldots, u_n)$ for $s_1, \ldots, s_j \in \{0,1\}$.

The algorithm proceeds in n rounds. First, for every $s \in \{0,1\}^n$, set

$$z_0(s) \stackrel{\text{def}}{=} x(s). \tag{4}$$

In round $j = 1, 2, \ldots, n$, for every $u \in \{0,1\}^n$, set

$$\begin{aligned} &z_j(u_1, \ldots, u_n) \stackrel{\text{def}}{=} \\ &\quad k(u_j, 0) \cdot z_{j-1}(u_1, \ldots, u_{j-1}, 0, u_{j+1}, \ldots, u_n) \\ &\quad + k(u_j, 1) \cdot z_{j-1}(u_1, \ldots, u_{j-1}, 1, u_{j+1}, \ldots, u_n). \end{aligned} \tag{5}$$

Lemma 9. *For all* $j = 0, 1, \ldots, n$ *and* $u \in \{0,1\}^n$ *it holds that*

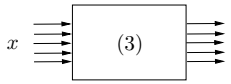
$$z_j(u_1, \ldots, u_n) = \sum_{s_1, \ldots, s_j \in \{0,1\}} \left(\prod_{i=1}^{j} k(u_i, s_i) \right) x(s_1, \ldots, s_j, u_{j+1}, \ldots, u_n)\,. \tag{6}$$

Proof. By induction on j. □

Exercise 10. *Give a full proof for Lemma 9.*

In particular, (3) and (6) imply that $z_n(t) = y(t)$ for all $t \in \{0,1\}^n$. Thus, *Yates's algorithm* given by (4) and (5) evaluates (3) in $O(2^n n)$ ring operations.

Observe that we may think of (4) and (5) as a specification of an arithmetic circuit with inputs for x and outputs for $y = z_n$.



Exercise 11. *Draw Yates's algorithm as an arithmetic circuit when $n = 3$ and $k(0,0) = 1$, $k(1,0) = 1$, $k(0,1) = 0$, $k(1,1) = 1$.*

Exercise 12. *Let A and B be $m \times n$ and $p \times q$ matrices, respectively. The Kronecker product $A \otimes B$ is the $mp \times nq$ matrix defined by*

$$A \otimes B \stackrel{\text{def}}{=} \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}.$$

Express (3) *as a matrix–vector product, where the matrix is obtained via Kronecker products. Using this representation, design a recursive version of Yates's algorithm.*

Exercise 13. *Extend Yates's algorithm to evaluate sums of the form*

$$y(t_1, \ldots, t_n) = \sum_{s_1=0}^{m_1-1} \sum_{s_2=0}^{m_2-1} \cdots \sum_{s_n=0}^{m_n-1} k_1(t_1, \ldots, t_n, s_1) k_2(t_2, \ldots, t_n, s_2) \cdots k_n(t_n, s_n) x(s_1, \ldots, s_n)$$

where $t_i = 0, 1, \ldots, m_i - 1$ for $i = 1, 2, \ldots n$.

Exercise 14. *Consider the one-dimensional discrete Fourier transform for a sequence $x(0), x(1), \ldots, x(2^n - 1)$ of length 2^n:*

$$y(t) = \sum_{s=0}^{2^n-1} \exp\left(\frac{2\pi\Im st}{2^n}\right) x(s) \quad \text{for } t = 0, 1, \ldots, 2^n - 1.$$

Reduce to the form in the previous exercise.

2.2. **Strassen's algorithm.** How many multiplications does it take to multiply two 2×2 matrices? A direct evaluation takes 8 multiplications:

$$X = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \quad Y = \begin{bmatrix} e & f \\ g & h \end{bmatrix}, \quad XY = \begin{bmatrix} ae + bg & af + bh \\ ce + dg & cf + dh \end{bmatrix}.$$

However, as was discovered by Strassen in 1968, 7 multiplications suffice:

$$XY = \begin{bmatrix} q_5 + q_4 - q_2 + q_6 & q_1 + q_2 \\ q_3 + q_4 & q_1 + q_5 - q_3 - q_7 \end{bmatrix},$$

where

$$\begin{aligned} q_1 &= a(f - h), & q_5 &= (a + d)(e + h), \\ q_2 &= (a + b)h, & q_6 &= (b - d)(g + h), \\ q_3 &= (c + d)e, & q_7 &= (a - c)(e + f), \\ q_4 &= d(g - e). \end{aligned}$$

Exercise 15. *Express Strassen's formula as an arithmetic circuit with three levels of gates (addition, multiplication, addition).*

Now, suppose we want to multiply two $n \times n$ matrices, X and Y. Assuming n is even (if not, insert a row and column of 0s to both matrices), we can partition X and Y each into four $\lceil n/2 \rceil \times \lceil n/2 \rceil$ submatrices

$$X = \begin{bmatrix} A & B \\ C & D \end{bmatrix}, \qquad Y = \begin{bmatrix} E & F \\ G & H \end{bmatrix}$$

whereby the product XY can be expressed in similarly partitioned form as

$$XY = \begin{bmatrix} AE + BG & AF + BH \\ CE + DG & CF + DH \end{bmatrix}.$$

Déjà vu? Indeed, Strassen's discovery tells us how to compute the matrix product XY using only 7 multiplications of $\lceil n/2 \rceil \times \lceil n/2 \rceil$ matrices! By recursion, we obtain that two $n \times n$ matrices can be multiplied in

$$T(n) = 7\,T(\lceil n/2 \rceil) + O(n^2) \tag{7}$$

arithmetic operations, that is, $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$. Computing directly from the definition takes $\Theta(n^3)$ operations.

Can one do better than $O(n^{2.81})$? Put otherwise, for which exponents α do there exist algorithms such that one can multiply $n \times n$ matrices in time $O(n^\alpha)$? Obviously $\alpha \geq 2$. Let ω be the greatest lower bound for such exponents α. We say that ω is the *exponent of matrix multiplication*. Currently the best upper bound is $\omega \leq 2.376$ (see §5.3), but many conjecture that $\omega = 2$.

3. Transformations on the subset lattice

Let $U = \{1, 2, \ldots, n\}$ and denote by 2^U the set of all subsets of U. The basic set operations ($A \cup B$, $A \cap B$, $A \setminus B$, ...) on 2^U induce natural bilinear transformations on functions $f : 2^U \to R$. We study two such transformations, the *union product* and the *disjoint union product*.

3.1. **The union product.** Let $f : 2^U \to R$ and $g : 2^U \to R$. Define the *union product* $f \cup g : 2^U \to R$ by

$$(f \cup g)(S) \stackrel{\text{def}}{=} \sum_{A \cup B = S} f(A)g(B) \quad \text{for } S \subseteq U, \tag{8}$$

where the sum is understood to range over all pairs $A, B \subseteq U$ such that $A \cup B = S$.

Observe that (8) is an instance of (2).

Exercise 16. $(f \cup g) \cup h = f \cup (g \cup h)$.

Exercise 17. $(f_1 \cup f_2 \cup \cdots \cup f_k)(S) = \sum_{A_1 \cup A_2 \cup \cdots \cup A_k = S} f_1(A_1) f_2(A_2) \cdots f_k(A_k)$.

Exercise 18. *For a given $S \subseteq U$, how many pairs (A, B) are there such that $A \cup B = S$? Given f and g as input, how many arithmetic operations does it take to compute $f \cup g$ directly from the definition?*

3.2. **Moebius inversion.** The principle of *Moebius inversion* states that any finite partially ordered set $(X, \leq)$ has a pair of mutually inverse linear transformations, the *zeta transform* and the *Moebius transform*, for manipulating functions $f : X \to R$. In particular, this applies to $(2^U, \subseteq)$, and will reduce $f \cup g$ into a pointwise product amenable for fast evaluation.

Lemma 19. *A finite nonempty set has equally many subsets of even and odd size.*

Proof. Let $x \in U$. Partition the subsets of U into pairs by associating $S \subseteq U \setminus \{x\}$ with $S \cup \{x\}$. Exactly one set in each pair has even (odd) size. □

Exercise 20. *Give an algebraic proof for Lemma 19. Hint: the Binomial Theorem $(x + y)^n = \sum_{i=0}^{n} \binom{n}{i} x^i y^{n-i}$.*

Let $f : 2^U \to R$. Define the *zeta transform* $f\zeta : 2^U \to R$ by

$$(f\zeta)(T) \stackrel{\text{def}}{=} \sum_{S \subseteq T} f(S) \quad \text{for } T \subseteq U. \tag{9}$$

Define the *Moebius transform* $f\mu : 2^U \to R$ by

$$(f\mu)(T) \stackrel{\text{def}}{=} \sum_{S \subseteq T} (-1)^{|S|+|T|} f(S) \quad \text{for } T \subseteq U. \tag{10}$$

Observe that both (9) and (10) are examples of (1). Also observe that if we are working over an abstract ring R, then "1" in (10) refers to the multiplicative identity element of R.

Exercise 21. *Given f as input, how many arithmetic operations does it take to compute $f\zeta$ directly from the definition?*

For a logical proposition P, we use Iverson's bracket notation $[P]$ as a shorthand for 1 when P is true, and for 0 when P is false. This notation will be convenient when simplifying nested sums, such as in the proof of the following lemma.

Lemma 22. *The zeta- and Moebius transforms on $(2^U, \subseteq)$ are mutual inverses.*

Proof. We show that $f\zeta\mu = f$ and leave $f\mu\zeta = f$ as Exercise 23. Consider an arbitrary $S \subseteq U$. We have

$$\begin{aligned}
(f\zeta\mu)(S) &= \sum_{Q \subseteq S} (-1)^{|Q|+|S|} (f\zeta)(Q) && \text{(definition)} \\
&= \sum_{Q \subseteq S} (-1)^{|Q|+|S|} \sum_{P \subseteq Q} f(P) && \text{(definition)} \\
&= \sum_{Q} [Q \subseteq S](-1)^{|Q|+|S|} \sum_{P} [P \subseteq Q] f(P) && \text{(to brackets)} \\
&= \sum_{Q,P} [Q \subseteq S](-1)^{|Q|+|S|} [P \subseteq Q] f(P) && \text{(expand)} \\
&= (-1)^{|S|} \sum_{P} f(P) \sum_{Q} (-1)^{|Q|} [P \subseteq Q][Q \subseteq S] && \text{(collect)} \\
&= (-1)^{|S|} \sum_{P} f(P) \sum_{Q} (-1)^{|Q|} [P \subseteq Q \subseteq S] && \text{(simplify brackets)} \\
&= (-1)^{|S|} \sum_{P} f(P) [P \subseteq S] \sum_{X \subseteq S \setminus P} (-1)^{|P|+|X|} && (X \stackrel{\text{def}}{=} Q \setminus P) \\
&= (-1)^{|S|} \sum_{P} f(P) (-1)^{|P|} [P = S] && \text{(Lemma 19)} \\
&= f(S)\,. && \text{(simplify sum)}
\end{aligned}$$

□

Exercise 23. *Show that $f\mu\zeta = f$.*

Exercise 24. *What do the matrices associated with ζ and μ look like if we consider the subsets indexing the rows and columns of the matrices in lexicographic order? Construct the matrices for small values of n. By looking at the matrices, can you devise a fast recursion for computing $f\zeta$ and $f\mu$ given f as input?*

Exercise 25. *Generalize the zeta transform to functions $f : X \to R$, where X is an arbitrary finite set equipped with a partial order $\leq$. Prove that the zeta transform on X is invertible. Derive an explicit form for the Moebius transform on your favourite partially ordered set, say, the set of positive divisors of the integer n, partially ordered by divisibility.*

Define the *pointwise product* $f \cdot g : 2^U \to R$ by $(f \cdot g)(S) \stackrel{\text{def}}{=} f(S)g(S)$ for $S \subseteq U$.

Lemma 26. $(f \cup g)\zeta = (f\zeta) \cdot (g\zeta)$.

Proof. For a $T \subseteq U$, we have

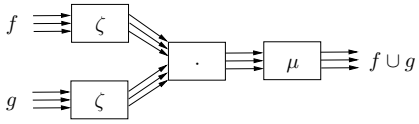
$$\begin{aligned}
((f \cup g)\zeta)(T) &= \sum_{S \subseteq T} (f \cup g)(S) && \text{(definition)} \\
&= \sum_{S \subseteq T} \sum_{A \cup B = S} f(A)g(B) && \text{(definition)} \\
&= \sum_{S} [S \subseteq T] \sum_{A,B} [A \cup B = S] f(A)g(B) && \text{(to brackets)} \\
&= \sum_{S,A,B} [S \subseteq T][A \cup B = S] f(A)g(B) && \text{(expand)} \\
&= \sum_{A,B} f(A)g(B) \sum_{S} [S \subseteq T][A \cup B = S] && \text{(collect)} \\
&= \sum_{A,B} f(A)g(B) \sum_{S} [A \cup B = S \subseteq T] && \text{(simplify brackets)} \\
&= \sum_{A,B} f(A)g(B)[A \cup B \subseteq T] && \text{(simplify sum)} \\
&= \sum_{A,B} f(A)g(B)[A \subseteq T][B \subseteq T] && \text{(simplify brackets)} \\
&= \sum_{A} [A \subseteq T] f(A) \sum_{B} [B \subseteq T] g(B) && \text{(collect)} \\
&= \sum_{A \subseteq T} f(A) \sum_{B \subseteq T} g(B) && \text{(to sums)} \\
&= (f\zeta)(T)(g\zeta)(T)\,. && \text{(definition)}
\end{aligned}$$

□

By Moebius inversion, we can recover the union product by taking the Moebius transform on both sides:

$$f \cup g = ((f\zeta) \cdot (g\zeta))\mu\,. \tag{11}$$

Again we may think of (11) as a circuit specification:



Exercise 27. $(f_1 \cup f_2 \cup \cdots \cup f_k)\zeta = (f_1\zeta) \cdot (f_2\zeta) \cdot \ldots \cdot (f_k\zeta)$.

Exercise 28. *How many operations does it take to directly evaluate the right-hand side of* $\underbrace{f \cup f \cup \cdots \cup f}_{k \text{ terms}} = (f\zeta)^{\cdot k}\mu$ *? What if we only want* $(f \cup f \cup \cdots \cup f)(U)$*?*

3.3. **Disjoint union product (subset convolution).** Let us consider a variant of the union product where we require the unions $A \cup B = S$ to be disjoint, that is, that $A \cap B = \emptyset$ must also hold.

For brevity, let us write $A \mathbin{\dot\cup} B = S$ as a shorthand for $A \cup B = S$ and $A \cap B = \emptyset$. More generally, let us write $A_1 \mathbin{\dot\cup} A_2 \mathbin{\dot\cup} \cdots \mathbin{\dot\cup} A_k = S$ for $A_1 \cup A_2 \cup \cdots \cup A_k = S$ and $A_i \cap A_j = \emptyset$ for all $1 \leq i < j \leq k$.

Define the *disjoint union product* $f \mathbin{\dot\cup} g : 2^U \to R$ by

$$(f \mathbin{\dot\cup} g)(S) \stackrel{\text{def}}{=} \sum_{A \dot\cup B = S} f(A)g(B) \quad \text{for } S \subseteq U. \tag{12}$$

The union product is also called the *subset convolution* because of the "convolution-like" equivalent form

$$(f \mathbin{\dot\cup} g)(S) = \sum_{T \subseteq S} f(T)g(S \setminus T)\,.$$

Again observe that (12) is an instance of (2).

Exercise 29. $(f \mathbin{\dot\cup} g) \mathbin{\dot\cup} h = f \mathbin{\dot\cup} (g \mathbin{\dot\cup} h)$.

Exercise 30. $(f_1 \mathbin{\dot\cup} f_2 \mathbin{\dot\cup} \cdots \mathbin{\dot\cup} f_k)(S) = \sum_{A_1 \dot\cup A_2 \dot\cup \cdots \dot\cup A_k = S} f_1(A_1)f_2(A_2)\cdots f_k(A_k)$.

Exercise 31. *Given f and g as input, how many arithmetic operations does it take to compute $f \mathbin{\dot\cup} g$ directly using the convolution form?*

Let us now reduce $f \mathbin{\dot\cup} g$ via "polynomial extension" to the union product. The following fact will provide the crux of the argument.

Lemma 32. *For $A, B, S \subseteq U$ we have $A \cup B = S$ and $A \cap B = \emptyset$ if and only if $A \cup B = S$ and $|A| + |B| = |S|$.*

Let w be a polynomial indeterminate, and denote by $R^{\langle w \rangle}$ the associated univariate polynomial ring with coefficients in R. For a polynomial $p = \sum_i a_i w^i \in R^{\langle w \rangle}$, denote by $\{w^j\}p$ the coefficient of the monomial w^j in p, that is, $\{w^j\}p = a_j$.

Exercise 33. *Can we use an R-arithmetic circuit to simulate an $R^{\langle w \rangle}$-arithmetic circuit? What if we introduce an $R^{\langle w \rangle}$-arithmetic gate $\{w^j\}$ that takes as input a polynomial and outputs (as a polynomial of degree zero) the coefficient of the monomial w^j for a constant j?*

Exercise 34. *Can we replace computations with explicit polynomials by computations with evaluations of such polynomials in sufficiently many distinct points $w = w_0, w_1, \ldots, w_d$ to enable recovery of the polynomial coefficients (as necessary) via interpolation? Can we interpolate in an arbitrary ring?*

Let us now proceed with the reduction. Suppose we are given $f : 2^U \to R$ and $g : 2^U \to R$ as input. Define $f^{\langle w \rangle} : 2^U \to R^{\langle w \rangle}$ and $g^{\langle w \rangle} : 2^U \to R^{\langle w \rangle}$ by $f^{\langle w \rangle}(S) = f(S)w^{|S|}$ and $g^{\langle w \rangle}(S) = g(S)w^{|S|}$ for $S \subseteq U$.

Note in the following equality that the union product on the right-hand side is evaluated in $R^{\langle w\rangle}$.

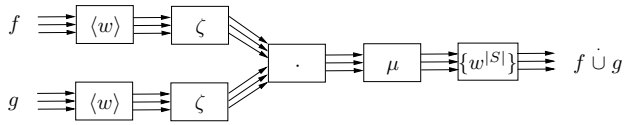
Lemma 35. $(f \dot\cup g)(S) = \{w^{|S|}\}(f^{\langle w\rangle} \cup g^{\langle w\rangle})(S)$.

Proof.

$$\begin{aligned}
\{w^{|S|}\}(f^{\langle w\rangle} \cup g^{\langle w\rangle})(S) &= \{w^{|S|}\} \sum_{A\cup B=S} f^{\langle w\rangle}(A) g^{\langle w\rangle}(B) && \text{(definition)}\\
&= \{w^{|S|}\} \sum_{A\cup B=S} f(A) w^{|A|} g(B) w^{|B|} && \text{(definition)}\\
&= \sum_{A\cup B=S} \{w^{|S|}\} f(A) w^{|A|} g(B) w^{|B|} && \text{(linearity)}\\
&= \sum_{A\cup B=S} \{w^{|S|}\} f(A) g(B) w^{|A|+|B|} && \text{(collect)}\\
&= \sum_{A,B} [A\cup B = S] \{w^{|S|}\} f(A) g(B) w^{|A|+|B|} && \text{(to brackets)}\\
&= \sum_{A,B} [A\cup B = S][|A|+|B| = |S|] f(A) g(B) && \text{(definition)}\\
&= \sum_{A,B} [A\cup B = S][A\cap B = \emptyset] f(A) g(B) && \text{(Lemma 32)}\\
&= \sum_{\substack{A\cup B=S\\ A\cap B=\emptyset}} f(A) g(B) && \text{(to sums)}\\
&= \sum_{A\dot\cup B=S} f(A) g(B) && \text{(definition)}\\
&= (f \dot\cup g)(S)\,. && \text{(definition)}
\end{aligned}$$

□

Again we may view the result as a circuit specification:



Exercise 36. $(f_1 \dot\cup f_2 \dot\cup \cdots \dot\cup f_k)(S) = \{w^{|S|}\}(f_1^{\langle w\rangle} \cup f_2^{\langle w\rangle} \cup \cdots \cup f_k^{\langle w\rangle})(S)$.

Exercise 37. *Consider the disjoint union product. Suppose we relax $A\cup B = S$ to $A, B \subseteq S$, but still require that $A\cap B = \emptyset$. Study the resulting "disjoint packing" product. Can you reduce the disjoint packing product to the disjoint union product?*

4. Two examples of algebraization

We consider two examples of algebraization and associated time–space tradeoffs enabled by fast evaluation.

4.1. **Graph coloring.** Let G be an undirected loopless graph with vertex set V. A set $I \subseteq V$ is *independent* in G if no two vertices in I are joined by an edge of G. We say that G is *k-colorable* if there exist independent sets $I_1, I_2, \ldots, I_k$ such that $I_1 \dot\cup I_2 \dot\cup \cdots \dot\cup I_k = V$. Such an ordered tuple $(I_1, I_2, \ldots, I_k)$ is a (proper) *k-coloring* of G. Indeed, the intuition is that the vertices in I_i have "color" i; a coloring is proper if and only if the ends of every edge have distinct colors.

The *k-coloring problem* asks, given a graph G and a positive integer k as input, whether G has a proper k-coloring.

4.2. **Algebraizing graph coloring.** Denote by $\mathbb{Z}$ the ring of integers. Let $f : 2^V \to \mathbb{Z}$ be the indicator function for independent sets in G, that is, $f(I) = [I \text{ is independent in } G]$ for $I \subseteq V$. Note that f is implicitly defined by the input G.

The number of distinct proper k-colorings of G is, by an iterated application of the disjoint union product,

$$\sum_{I_1 \dot\cup I_2 \dot\cup \cdots \dot\cup I_k = V} f(I_1) f(I_2) \cdots f(I_k) = \{w^{|V|}\}((f^{\langle w \rangle}\zeta)^{\cdot k}\mu)(V)\,. \tag{13}$$

Assuming that $|V| = n$ and $k = O(n)$, a direct evaluation of the right-hand side of (13) requires $O^*(3^n)$ time and $O^*(1)$ space, where $O^*(\cdot)$ hides a multiplicative factor polynomial in n.

4.3. **A time-space tradeoff via fast Moebius inversion.** The evaluation of product forms such as (13) can be expedited if sufficient space is available.

Lemma 38. *The zeta transform on the subset lattice can be computed in $O(2^n n)$ ring operations given space for $O(2^n)$ ring elements.*

Proof. Recall that we assume $U = \{1, 2, \ldots, n\}$. Identify $S \subseteq U$ with the binary string $s = (s_1, s_2, \ldots, s_n) \in \{0,1\}^n$ by $i \in S$ if and only if $s_i = 1$ for $1 \leq i \leq n$. In particular, we have $[S \subseteq T] = \prod_{i=1}^n [s_i \leq t_i]$. Thus, Yates's algorithm with $x(s) \leftarrow f(S)$ and $k(b, a) \leftarrow [a \leq b]$ in (3) uses $O(2^n n)$ ring operations to compute $(f\zeta)(T) = y(t)$ for all $T \subseteq U$. Storage for $O(2^n)$ ring elements suffices because z_j depends only on z_{j-1} in (5). □

Exercise 39. *Show that the Moebius transform admits a similar tradeoff.*

Theorem 40. *The union product can be computed in $O(2^n n)$ ring operations given space for $O(2^n)$ ring elements.*

Proof. We take advantage of (11) and Lemma 38. Given f and g as input, compute $f\zeta$ and $g\zeta$, take the pointwise product $(f\zeta) \cdot (g\zeta)$, and finally compute the Moebius transform $((f\zeta) \cdot (g\zeta))\mu$ to obtain $f \cup g$. Each of the three steps takes $O(2^n n)$ operations. □

Exercise 41. *Show that the disjoint union product can be computed in $O(2^n n^2)$ ring operations given space for $O(2^n)$ ring elements.*

Thus, given $O^*(2^n)$ space, we can solve graph coloring in $O^*(2^n)$ time by evaluating the right-hand side of (13).

Exercise 42. *Can you give other examples of natural "partitioning problems" that can be solved using product forms such as* (13)*?*

Exercise 43. *Algebraize the task of counting connected spanning subgraphs of a given graph with n vertices. Develop an algorithm with $O^*(2^n)$ running time. Hint: A graph with e edges has exactly 2^e spanning subgraphs, each of which partitions into one or more connected components.*

4.4. **Maximum satisfiability (MAX-SAT).** Let $x_1, x_2, \ldots, x_n$ be variables taking values in $\{0,1\}$. A *literal* is a variable x_i or its negation $\bar{x}_i$. A positive literal x_i (respectively, a negative literal $\bar{x}_i$) is *satisfied* by an assignment of values to the variables if $x_i = 1$ (respectively, $x_i = 0$). A *clause* of length k (a *k-clause*) is a set of k literals. A clause is *satisfied* if at least one of its literals is satisfied.

The *k-satisfiability* problem (k-SAT) asks, given a collection of k-clauses as input, whether there exists an assignment of values to the variables such that all input clauses are satisfied. The *maximum k-satisfiability* problem (MAX-k-SAT) asks, given a collection of k-clauses as input, for the maximum number of input clauses that can be satisfied by an assignment of values to the variables.

4.5. **Algebraizing MAX-SAT.** Suppose a collection of m clauses over n variables has been given as input. Let us view an assignment of values to the variables $x_1, x_2, \ldots, x_n$ as an n-tuple $t = (t_1, t_2, \ldots, t_n) \in \{0,1\}^n$, where t_i is the value assigned to x_i for $1 \le i \le n$.

For an assignment $t \in \{0,1\}^n$, denote by $N(t)$ the number of input clauses satisfied by t. Introduce the generating function

$$G(w) = \sum_{t \in \{0,1\}^n} w^{N(t)}. \tag{14}$$

Observe that $G(w)$ is a polynomial of degree at most m with nonnegative integer coefficients that sum to 2^n. In particular, the degree of $G(w)$ is the maximum number of input clauses that can be satisfied by an assignment.

4.6. **A time-space tradeoff via fast matrix multiplication.** We now restrict to the case $k = 2$ (MAX-2-SAT). Assume that 3 divides n (if not, insert new variables). Partition the variables arbitrarily into three types A, B, C so that there are exactly $n/3$ variables of each type. Partition the input clauses into types A, B, C so that a clause of type T does **not** contain a variable of type T. Because $k = 2$, such a partition of the input clauses always exists.

We can now split an assignment $t \in \{0,1\}^n$ into three sub-assignments $a, b, c \in \{0,1\}^{n/3}$ for the variables of each type. The number of satisfied input clauses splits accordingly into

$$N(t) = N_C(a,b) + N_B(a,c) + N_A(b,c) \tag{15}$$

where N_C, N_B, and N_A count the number of satisfied input clauses of each respective type. In particular, N_T is independent of the sub-assignment to variables of type T.

We proceed to split $G(w)$ using (15) and recover a matrix product. Let $N_C^{\langle w \rangle}$, $N_B^{\langle w \rangle}$, $N_A^{\langle w \rangle}$ be matrices of size $2^{n/3} \times 2^{n/3}$ with entries defined for $a, b, c \in \{0,1\}^{n/3}$ by

$$N_C^{\langle w \rangle}(a,b) \stackrel{\text{def}}{=} w^{N_C(a,b)}, \quad N_B^{\langle w \rangle}(a,c) \stackrel{\text{def}}{=} w^{N_B(a,c)}, \quad N_A^{\langle w \rangle}(b,c) \stackrel{\text{def}}{=} w^{N_A(b,c)}.$$

We now have

$$\begin{aligned}
G(w) &= \sum_{a,b,c} w^{N_C(a,b)+N_B(a,c)+N_A(b,c)} && \text{(split to types)} \\
&= \sum_{a,b,c} w^{N_C(a,b)} w^{N_B(a,c)} w^{N_A(b,c)} && \text{(expand)} \\
&= \sum_{a,c} w^{N_B(a,c)} \sum_{b} w^{N_C(a,b)} w^{N_A(b,c)} && \text{(collect)} \qquad (16) \\
&= \sum_{a,c} N_B^{\langle w \rangle}(a,c) \sum_{b} N_C^{\langle w \rangle}(a,b) N_A^{\langle w \rangle}(b,c) && \text{(in matrix form)} \\
&= \sum_{a,c} N_B^{\langle w \rangle}(a,c) \big(N_C^{\langle w \rangle} N_A^{\langle w \rangle}\big)(a,c). && \text{(to matrix product)}
\end{aligned}$$

A trivial algorithm for MAX-2-SAT runs in $O^*(2^n)$ time and $O^*(1)$ space. Using (16), we can now trade space for time by first constructing the matrices $N_C^{\langle w \rangle}$, $N_B^{\langle w \rangle}$, $N_A^{\langle w \rangle}$ and then using fast matrix multiplication to determine the product $N_C^{\langle w \rangle} N_A^{\langle w \rangle}$ in $O((2^{n/3})^{\omega+\epsilon})$ ring operations for any fixed $\epsilon > 0$, which leads to $O^*(2^{(\omega+\epsilon)n/3})$ time and $O^*(2^{2n/3})$ space for MAX-2-SAT.

Exercise 44. *Observe that one can carry out the computations in* (16) *with evaluations of the generating function $G(w)$ at $m+1$ distinct points $w = w_0, w_1, \ldots, w_m$, and then recover $G(w)$ via interpolation.*

5. Further exercises and remarks

5.1. **Transforms on the subset lattice.** The following exercises develop and relate to each other some further transformations on the subset lattice.

Define the *up-zeta transform* $f\zeta' : 2^U \to R$ by

$$f\zeta'(T) \stackrel{\text{def}}{=} \sum_{T \subseteq S} f(S) \quad \text{for } T \subseteq U.$$

Define the *up-Moebius transform* $f\mu' : 2^U \to R$ by

$$f\mu'(T) \stackrel{\text{def}}{=} \sum_{T \subseteq S} (-1)^{|S|+|T|} f(S) \quad \text{for } T \subseteq U.$$

Exercise 45. *Observe that in matrix form we obtain ζ' from ζ by taking the transpose. Similarly for μ' and μ.*

Define the *complement transform* $f\kappa : 2^U \to R$ by

$$(f\kappa)(S) \stackrel{\text{def}}{=} f(U \setminus S) \quad \text{for } S \subseteq U.$$

Define the *odd-negation transform* $f\sigma : 2^U \to R$ by

$$(f\sigma)(S) \stackrel{\text{def}}{=} (-1)^{|S|} f(S) \quad \text{for } S \subseteq U.$$

Exercise 46. $\zeta' = \kappa\zeta\kappa$, $\mu' = \kappa\mu\kappa$, $\mu = \sigma\zeta\sigma$, $\zeta = \sigma\mu\sigma$.

Exercise 47. $\zeta'\mu = \zeta\kappa\sigma$, $\zeta\mu' = \zeta'\sigma\kappa$, $\mu'\zeta = \mu\sigma\kappa$, $\mu\zeta' = \mu'\kappa\sigma$.

Exercise 48. *Present a natural definition for $\delta \stackrel{\text{def}}{=} \zeta\kappa$. Show that $\delta = \zeta'\sigma\zeta$.*

Exercise 49. *Define the intersection product* $f \cap g : 2^U \to R$ *by* $(f \cap g)(S) \stackrel{\text{def}}{=} \sum_{A \cap B = S} f(A)g(B)$ *for* $S \subseteq U$*. Show that* $(f \cap g)\kappa = (f\kappa) \cup (f\kappa)$*. Take the up-zeta transform of* $f \cap g$ *and simplify to a pointwise product. Establish* $(f \cap g) \cap h = f \cap (g \cap h)$*.*

Exercise 50. *Define the difference product* $f \setminus g : 2^U \to R$ *by* $(f \setminus g)(S) \stackrel{\text{def}}{=} \sum_{A \setminus B = S} f(A)g(B)$ *for* $S \subseteq U$*. Show that* $f \setminus g = f \cap (g\kappa)$*.*

Exercise 51. *Define the Walsh–Hadamard transform (the Fourier transform on the n-cube)* $f\phi : 2^U \to R$ *by* $(f\phi)(S) \stackrel{\text{def}}{=} \sum_T (-1)^{|S \cap T|} f(T)$ *for* $S \subseteq U$*. Show that* $f\phi^2 = 2^n f$*. Is* ϕ *invertible if* R *is an arbitrary ring?*

Exercise 52. *For* $A, B \subseteq U$*, define the symmetric difference* $A \oplus B \stackrel{\text{def}}{=} (A \setminus B) \cup (B \setminus A)$*. Define the symmetric difference product* $f \oplus g : 2^U \to R$ *by* $(f \oplus g)(S) \stackrel{\text{def}}{=} \sum_{A \oplus B = S} f(A)g(B)$ *for* $S \subseteq U$*. Show that* $(f \oplus g)\phi = (f\phi) \cdot (g\phi)$*. Establish* $(f \oplus g) \oplus h = f \oplus (g \oplus h)$*.*

Exercise 53. *Reduce the disjoint union product to the symmetric difference product.*

Exercise 54. *Let* $f : 2^U \to R$ *and* $k : 2^U \to R$*. Define the k-intersection transform* $f\tau_k^\cap : 2^U \to R$ *by* $(f\tau_k^\cap)(S) = \sum_T k(S \cap T)f(T)$ *for* $S \subseteq U$*. Show that* $f\tau_k^\cap = ((k\mu) \cdot (f\zeta'))\zeta$*.*

Exercise 55. *Observe that* ϕ *reduces to* $\tau_k^\cap$ *for a specific* k*. Simplify the pointwise product form of the k-intersection product when* k *is* 1 *on sets of size* j*, and* 0 *elsewhere.*

Exercise 56. *Define similar k-union, k-difference, and k-symmetric difference transforms. Reduce each to a pointwise product form.*

Exercise 57. *Are all the transforms in this section computable in* $O^*(2^n)$ *ring operations given space for* $O^*(2^n)$ *ring elements?*

5.2. **Trimming.** This section investigates "trimming" of transforms on 2^U when the input and output are restricted to subsets of 2^U.

Our first objective is a "closure trimming lemma" for fast Moebius inversion.

Let $\mathcal{F} \subseteq 2^U$. Denote by $\uparrow\mathcal{F}$ the *up-closure* of $\mathcal{F}$, that is, the set consisting of the sets in $\mathcal{F}$ and all their supersets in 2^U. Denote by $\downarrow\mathcal{F}$ the *down-closure* of $\mathcal{F}$, that is, the set consisting of the sets in $\mathcal{F}$ and all their subsets in 2^U. Define the *elementwise complement* of $\mathcal{F}$ by $U\setminus\mathcal{F} \stackrel{\text{def}}{=} \{U\setminus S : S \in \mathcal{F}\}$.

Exercise 58. $U\setminus\uparrow\mathcal{F} = \downarrow U\setminus\mathcal{F}, \quad U\setminus\downarrow\mathcal{F} = \uparrow U\setminus\mathcal{F}$*.*

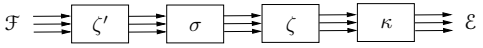
Lemma 59. *There exist algorithms that, given* $\mathcal{E} \subseteq 2^U$ *and* $\mathcal{F} \subseteq 2^U$ *as input, construct an R-arithmetic circuit with input gates for* $f : \mathcal{F} \to R$ *and output gates that evaluate to any of* $f\zeta : \mathcal{E} \to R$*,* $f\zeta' : \mathcal{E} \to R$*,* $f\mu : \mathcal{E} \to R$*, or* $f\mu' : \mathcal{E} \to R$*, with construction time any of* $O^*(|\uparrow\mathcal{E}| + |\uparrow\mathcal{F}|)$*,* $O^*(|\uparrow\mathcal{E}| + |\downarrow\mathcal{F}|)$*,* $O^*(|\downarrow\mathcal{E}| + |\uparrow\mathcal{F}|)$*, or* $O^*(|\downarrow\mathcal{E}| + |\downarrow\mathcal{F}|)$*.*

Proof. Because $\mu = \sigma\zeta\sigma$ and $\mu' = \sigma\zeta'\sigma$, it suffices to consider ζ and ζ' only. Because $\zeta' = \kappa\zeta\kappa$, it suffices to consider ζ only. To evaluate $f\zeta : \mathcal{E} \to R$ it suffices to consider $f : \downarrow\mathcal{E} \to R$ because f outside $\downarrow\mathcal{E}$ does not affect $f\zeta$ in $\mathcal{E}$.

Time $O^*(|{\downarrow}\mathcal{E}| + |\mathcal{F}|)$. Consider Yates's algorithm with $k(0,0) = 1$, $k(0,1) = 0$, $k(1,0) = 1$, and $k(1,1) = 1$, that is, the output is the zeta transform of the input. In this case any walk in the n-cube that has a $1 \mapsto 0$ step has weight 0 because $k(0,1) = 0$. Thus, such walks may be discarded. Furthermore, all walks to vertices in $\mathcal{E}$ that do not have a $1 \mapsto 0$ step traverse only vertices in ${\downarrow}\mathcal{E}$. Thus, Yates's algorithm in fact specifies an R-arithmetic circuit of size $O(n|{\downarrow}\mathcal{E}|)$ with inputs and outputs indexed by ${\downarrow}\mathcal{E}$. The construction is completed by connecting the inputs in $\mathcal{F}$ with corresponding input gates in the zeta circuit; any inputs to the zeta circuit not in $\mathcal{F}$ are forced to 0.

Time $O^*(|\mathcal{E}| + |{\uparrow}\mathcal{F}|)$. Observe that $f\zeta$ vanishes outside ${\uparrow}\mathcal{F}$. Furthermore, all walks to ${\uparrow}\mathcal{F}$ from $\mathcal{F}$ stay within ${\uparrow}\mathcal{F}$. Restrict Yates to ${\uparrow}\mathcal{F}$ and output 0 outside ${\uparrow}\mathcal{F}$.

Time $O^*(|{\uparrow}\mathcal{E}| + |{\downarrow}\mathcal{F}|)$. Observe that $\zeta = \zeta'\sigma\zeta\kappa$. Let us develop the right-hand side into a sequential circuit by "meeting in the middle."



Starting from the left, given the inputs at $\mathcal{F}$, we have that the output of ζ' vanishes outside ${\downarrow}\mathcal{F}$. Starting from the right, the outputs at $\mathcal{E}$ require input at $U\backslash\mathcal{E}$ for κ. Thus, we require input for ζ at ${\downarrow}U\backslash\mathcal{E}$ to evaluate the outputs at $U\backslash\mathcal{E}$. The odd-negation layer σ thus requires input and output at ${\downarrow}U\backslash\mathcal{E}$. We now connect the outputs of the left part with the corresponding inputs of the right part at $({\downarrow}\mathcal{F}) \cap ({\downarrow}U\backslash\mathcal{E})$, and force any remaining inputs of the right part to 0. Observe that ${\downarrow}U\backslash\mathcal{E} = U\backslash{\uparrow}\mathcal{E}$ and that $|U\backslash{\uparrow}\mathcal{E}| = |{\uparrow}\mathcal{E}|$. □

Exercise 60. *Open problem: Can one evaluate $f\zeta : \mathcal{E} \to R$ for given $\mathcal{E}, \mathcal{F} \subseteq 2^U$ and $f : \mathcal{F} \to R$ using an R-arithmetic circuit of size $O^*(|{\uparrow}\mathcal{E}| + |\mathcal{F}|)$ or $O^*(|\mathcal{E}| + |{\downarrow}\mathcal{F}|)$?*

Let us sketch an application of Lemma 59 in counting long paths in graphs. Recalling Exercise 49, consider the intersection product $f \cap g : \mathcal{E} \to R$ for given $f : \mathcal{F} \to R$ and $g : \mathcal{G} \to R$. We have

$$f \cap g = ((f\zeta') \cdot (g\zeta'))\mu'.$$

Observe that $(f\zeta') \cdot (g\zeta')$ vanishes outside $({\downarrow}\mathcal{F}) \cap ({\downarrow}\mathcal{G})$. Thus, Lemma 59 implies that we can evaluate $f \cap g$ in $O^*(|{\downarrow}\mathcal{E}| + |{\downarrow}\mathcal{F}| + |{\downarrow}\mathcal{G}|)$ ring operations, given space for $O^*(|{\downarrow}\mathcal{E}| + |{\downarrow}\mathcal{F}| + |{\downarrow}\mathcal{G}|)$ ring elements. In particular, letting $\mathcal{E} = \{\emptyset\}$, we can evaluate the sum

$$(f \cap g)(\emptyset) = \sum_{\substack{A \in \mathcal{F}, B \in \mathcal{G} \\ A \cap B = \emptyset}} f(A)g(B) \tag{17}$$

in $O^*(|{\downarrow}\mathcal{F}| + |{\downarrow}\mathcal{G}|)$ ring operations, given space for $O^*(|{\downarrow}\mathcal{F}| + |{\downarrow}\mathcal{G}|)$ ring elements.

Now consider an undirected graph G with vertex set V, $|V| = n$. Suppose we want to compute the number of (s,t)-paths of length k for given G, $1 \leq k \leq n-1$, and distinct $s, t \in V$. (The length of a path is the number of edges in it.) For simplicity, we assume that k is even.

Observe that every (s,t)-path of even length k has a unique midpoint $v \in V \setminus \{s,t\}$ that splits the path into two halves of equal length; that is, into an (s,v)-path and a (v,t)-path, both of length $k/2$.

We proceed to "count in halves" via (17). Let $v \in V \setminus \{s,t\}$ and $U = V \setminus \{v\}$. Let $f_v(A)$ be the number of (s,v)-paths in G with vertex set $A \cup \{v\}$, and let $g_v(B)$ be the number of (v,t)-paths in G with vertex set $B \cup \{v\}$.

Exercise 61. *Design a dynamic programming algorithm that computes* $f_v : \binom{U}{k/2} \to \mathbb{Z}$ *and* $g_v : \binom{U}{k/2} \to \mathbb{Z}$ *in time* $O^*(\binom{n}{k/2})$.

It now follows from (17) that we can count in time and space $O^*(\binom{n}{k/2})$ the number of (s,t)-paths of length k in G, that is,

$$\sum_{v \in V \setminus \{s,t\}} \sum_{\substack{A,B \in \binom{V \setminus \{v\}}{k/2} \\ A \cap B = \emptyset}} f_v(A) g_v(B).$$

5.3. **Bibliography.** Donald Knuth's *The Art of Computer Programming* (Volume 2: Seminumerical Algorithms, 3rd ed., Addison–Wesley, 1998) gives a comprehensive introduction to arithmetic algorithms. More specialized texts include *Computational Complexity of Algebraic and Numeric Problems* by A. Borodin and I. Munro (Elsevier, 1975), *Polynomial and Matrix Computations* by D. Bini and V. Y. Pan (Volume 1: Fundamental algorithms, Birkhäuser, 1994), and *Algebraic Complexity Theory* by P. Bürgisser, M. Clausen, and M. Amin Shokrollahi (Springer, 1997).

Strassen's algorithm appears in [V. Strassen, *Numer. Math.* 13 (1969) 354–356]. Yates's algorithm is due to F. Yates [*The Design and Analysis of Factorial Experiments*, Imperial Bureau of Soil Sciences, Harpenden 1937]. Currently the asymptotically fastest algorithm for matrix multiplication is due to D. Coppersmith and S. Winograd [*J. Symbolic Comp.* 9 (1990) 251–280].

An introductory text covering basic combinatorial techniques such as Moebius inversion is *A Course in Combinatorics* by J. H. van Lint and R. M. Wilson (Cambridge University Press, 1992).

The union product (in a dual form) together with fast Moebius inversion was introduced in [R. Kennes, *IEEE Transactions on Systems, Man, and Cybernetics* 22 (1991) 201–223]. The disjoint union product was introduced in A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto [Proc. 39th ACM Symposium on Theory of Computing, 2007, 67–74].

A. Björklund, T. Husfeldt, and M. Koivisto [*SIAM J. Comput.*, to appear; see also Proc. 47th IEEE Symposium on Foundations of Computer Science, 2006, 575–582 and 583–590] discovered that graph coloring and other partitioning problems can be solved in time $O^*(2^n)$ using inclusion–exclusion. Algebraization of MAX-2-CSP via matrix multiplication is due to R. Williams [*Theoret. Comput.Sci.* 348 (2005) 357–365] and M. Koivisto [*Inform. Proc. Lett.* 98 (2006) 22–24].

The closure trimming lemma (Lemma 59) is an unpublished extension of results in A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto [Proc. 25th Annual Symposium on Theoretical Aspects of Computer Science, 2008, 85–96]. The k-intersection transform in Exercise 54 is an unpublished slightly stronger version of results in A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto [arXiv:0809.2489]. The application to counting paths in graphs is from A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto [arXiv:0904.3093]; see [arXiv:0904.3251] for an application to evaluation of permanents.

Helsinki Institute for Information Technology HIIT, University of Helsinki, Department of Computer Science, P.O. Box 68, 00014 University of Helsinki, Finland
E-mail address: petteri.kaski@cs.helsinki.fi

Dieter Kratsch

Branching algorithms

Branching Algorithms

Dieter Kratsch

Abstract

We provide an introduction to the design and analysis of moderately exponential-time branching algorithms via the study of a collection of such algorithms among them algorithms for Maximum Independent Set, SAT and 3-SAT. The tools for simple running time analysis are presented. The limits of such an analysis including lower bounds for the worst-case running time of some particular branching algorithms are discussed. Some exercices are given.

Contents

1 A First Independent Set Algorithm

We first study a simple branching algorithm for the Maximum Independent Set algorithm. The algorithm uses the standard branching rule in MIS algorithms. "For every vertex v there is a maximum independent set containing v, or there is a maximum independent set not containing v." Note that when deciding that v is in the independent set then all its neighbors cannot be in it and thus they can be deleted safely. Therefore we may write the standard branching rule of the IS problem, that can be applied to any vertex v, as follows:

$$\text{mis}(G) = \max(1 + \text{mis}(G - N[v]), \text{mis}(G - v)).$$

The algorithm applies this rule to any vertex v of degree at most three in the current graph as long as this is possible. The recursion is stopped when the current graph has maximum degree two, i.e. it is a union of cycles and paths. In such a graph a maximum independent set can be computed in polynomial time.

It is not difficult to see that this algorithm is correct and indeed returns the size of a maximum independent set of the input graph. Usually correctness of branching algorithms is not hard to see (if not obvious).

How shall one estimate the overall running time of such a branching algorithm? There is a well-established procedure for such an analysis that is based on using linear recurrences. For our algorithm let $T(n)$ be the largest number of leaves (graphs of maximum degree two) of an input graph on n vertices for which the polynomial time algorithm is called. Then the running time of the algorithm is $O^*(T(n))$. The branching rule implies

$$T(n) \leq T(n-1) + T(n-d(v)-1) \leq T(n-1) + T(n-4).$$

Thus we shall say that our branching rule has branching vector $(1, d(v)+1)$ and the worst case is achieved if $d(v) = 3$. The corresponding branching vector is $(1, 4)$.

It is knwon that all basic solutions of such a linear recurrence are of the form α^n where α is a root of the polynomial

$$x^n = x^{n-1} + x^{n-4}.$$

Since we want to upper bound the running time we are interested in the largest solution of the characteristic polynomial. Fortunately it is known that this is always the unique positive real root of the polynomial. Using some system like Maple, Mathematica, Matlab etc. we obtain that our algorithm has running time $O^*(1.3803^n)$.

This analysis does not seem to take into account what the algorithm is really doing. Somehow with this tool we can analyze branching algorithms without understanding well what happens during an execution. But can this really provide the worst-case running time?

2 Fundamental Notions and Tools

We set up the scenario of a typical (moderately exponential time) branching algorithm.

Such an algorithm consists of a collection of *reduction rules* and *branching rules*. There are also halting rules and it needs to be specified which rule to apply on a particular instance. A reduction rule simplifies an instance (without branching). A branching rule recursively calls subproblems of the instance. The correctness of a branching algorithm follows from the correctness of all its reduction and branching rules (which in many cases is easy to see).

Search trees are very useful to illustrate an execution of a branching algorithm and to facilitate the time analysis of a branching algorithm. A *search tree* of an execution of a branching algorithm is obtained as follows: assign the root node of the search to the input of the problem; recursively assign a child to a node for each smaller instance reached by applying a branching rule to the instance of the node. Note that we do not assign a child to a node when an reduction rule is applied. Hence as long as the algorithm applies reduction rules to an instance the instance simplifies but the instance corresponds to the same node of the search tree.

What is the running time of a particular execution of the algorithm on an input instance? To obtain an easy answer, we assume that during its execution the running time of the algorithm corresponding to one node is polynomial. Under this assumption, that is satisfied for all branching algorithms to be presented, the running time of an execution is equal to the number of nodes of the corresponding search tree up to a polynomial factor.

Thus analysing the worst-case running time of a branching algorithm means to determine the maximum number of nodes in a search tree corresponding to the execution of the algorithm on an input of size n, where n is e.g. the number of vertices of a graph, the number of variables of a boolean formula.

The time analysis of branching algorithms is based on upper bounding the number of leaves of a search tree of an input of size n. Let $T(n)$ be the maximum number of leaves on any search tree of an input of size n. Now each branching rule is analyzed separately and finally we use the worst-case time over all branching rules as upper bound of the running time of the algorithm.

Let b be a branching rule of the algorithm to be analyzed. Consider an application of b to any instance of size n. Suppose it branches into $r \geq 2$ subproblems of size at most $n-t_1$, $n-t_2$, $\ldots$, $n-t_r$, for all instances of size $n \geq \max\{t_i : i = 1, 2, \ldots r\}$. Then we call $\vec{b} = (t_1, t_2, \ldots t_r)$ the *branching vector* of branching rule b. Hence

$$T(n) \leq T(n-t_1) + T(n-t_2) + \cdots + T(n-t_r).$$

It is known that the largest solution of any linear recurrence obtained by a branching algorithm is the unique positive real root of

$$x^n - x^{n-t_1} - x^{n-t_2} - \cdots - x^{n-t_r} = 0.$$

Sometimes this root is called the *branching factor* of branching vector $\vec{b}$.

Having computed the branching factors α_i for every branching vector b_i we simply take the largest base α_i to achieve an upper bound of the running time: $\alpha = \max_i \alpha_i$. Then $T(n) = O^*(\alpha^n)$ and the running time of the branching algorithm is $O^*(\alpha^n)$.

3 The Second Independent Set algorithm

The algorithm consists of one branching rule which is based on the fact that if I is a maximal independent set of G, then if v is not in I, then at least one of the neighbors is in I. This is because otherwise $I \cup \{v\}$ would be an independent set, which contradicts the maximality of I. Hence the algorithm picks a vertex of minimum degree and for each vertex from its closed neighborhood it recursively computes a maximal independent set of the current graph.

```
1  int mis(G = (V, E)) {
2      if(|V| = 0) return 0;
3      choose a vertex v of minimum degree in G
4          return 1 + max{mis(G − N[y]) : y ∈ N[v]};
5  }
```

To analyze the running time let G be the input graph of a subproblem. Suppose the algorithm branches on a vertex v of minimum degree $d(v)$. Let $y_1, y_2, \ldots y_{d(v)}$ be the neighbors of v in G. Thus for solving the subproblem G the algorithm recursively solves the subproblems $G - N[x]$, $G - N[y_1]$, ... ,$G - N[y_{d(v)}]$. Hence we obtain immediately the recurrence

$$T(n) \leq T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(y_i) - 1).$$

Since the algorithm is branching on a vertex of minimum degree, we have: for all $i = 1, 2, \ldots, d(v)$, $d(v) \leq d(v_i)$, $n - d(v_i) - 1 \leq n - d(v) - 1$ and, by the monotonicity of T, $T(n - d(v_i) - 1) \leq T(n - d(v) - 1)$. Consequently

$$T(n) \leq T(n - d(v) - 1) + \sum_{i=1}^{d(v)} T(n - d(v) - 1) \leq (d(v) + 1)T(n - d(v) - 1)$$

Taking $s = d(v) + 1$, we establish the recurrence $T(s) \leq s\,T(n-s)$, which has the solution $T(s) = s^{n/s}$. Since this function has its maximum value for integral s when $s = 3$, we obtain $T(n) \leq 3^{n/3}$. Hence the running time of the algorithm is $O^*(3^{n/3})$.

To mention some features of the algorithm. Any set of vertices selected for an independent set in a leave of the search tree is a maximal independent set of the input graph; and each maximal independent set corresponds to at least one leaf of the search tree. Thus the algorithm can be used to enumerate all maximal independent sets of a graph in time $O^*(3^{n/3})$, and hence a graph on n vertices has $O^*(3^{n/3})$ maximal independent sets. This provides a new and simpler proof the well-known combinatorial theorem of Moon and Moser. Since the bound is tight we also obtain that the worst-case running time of the algorithm is $\Theta^*(3^{n/3})$.

4 The Third Independent Set algorithm

We discuss various fundamental ideas of branching algorithms for the independent set problem and use them to construct a Third Independent Set algorithm.

The first one is a reduction rule called *domination rule*.

Lemma 1. *Let $G = (V, E)$ be a graph, let v and w be adjacent vertices of G such that $N[v] \subseteq N[w]$. Then*

$$\alpha(G) = \alpha(G - w).$$

Proof. We have to prove that G has a maximum independent set not containing w. Let I be a maximum independent set of G such that $w \in I$. Since $w \in I$ no neighbor of v except w belongs to I. Hence $I - w + v$ is an independent set of G, and thus a maximum independent set of G not containing w. □

Now let us study the branching rules of our algorithm. The *standard branching* has already been discussed:

$$\alpha(G) = \max(\alpha(G - N[v]), \alpha(G - v)).$$

Lemma 2. *Let $G = (V, E)$ be a graph and let v be a vertex of G. If no maximum independent set of G contains v then every maximum independent set of G contains at least two vertices of $N(v)$.*

Proof. Every maximum independent set of G is also a maximum independent set of $G - v$. Suppose there is a maximum independent set I of $G - v$ containing at most one vertex of $N(v)$. If I contains no vertex of $N[v]$ then

$I + v$ is independent and thus I is not a maximum independent set, contradiction. Otherwise, let $I \cap N(v) = \{w\}$. Then $I - w + v$ is an independent set of G, and thus there is a maximum independent set of G containing v, contradiction. □

Using the above Lemma, standard branching has been refined recently. Let $N^2(v)$ be the set of vertices in distance 2 to v in G, i.e. the set of the neighbors of the neighbors of v, except v itself. A vertex $w \in N^2(v)$ is called a *mirror* of v if $N(v) \setminus N(w)$ is a clique. Calling $M(v)$ the set of mirrors of v in G, the standard branching rule can be refined via mirrors.

Lemma 3. *Let $G = (V, E)$ be a graph and v a vertex of G. Then*

$$\alpha(G) = \max(\alpha(G - N[v]), \alpha(G - (M(v) + v)).$$

Proof. If G has any maximum independent set containing v then $\alpha(G) = \alpha(G - N[v])$ and the lemma is true. Otherwise suppose that no maximum independent set of G contains v. Then every maximum independent set of G contains two vertices of $N(v)$. Since w is a mirror the vertex subset $N(v) \setminus N(w)$ is a clique, and thus at least one vertex of every maximum independent set belongs to $N(w)$. Consequently, no maximum independent set contains w, and thus w can be safely discarded. □

We call the corresponding rule *mirror branching*.

Lemma 2 also implies the following reduction rule that we call *simplicial rule*.

Lemma 4. *Let $G = (V, E)$ be a graph and v be a vertex of G such that $N[v]$ is a clique. Then*

$$\alpha(G) = 1 + \alpha(G - N[v]).$$

Proof. If G has a maximum independent set containing v then the lemma is true. Otherwise, by Lemma 2 a maximum independent set must contain two vertices of the clique $N(v)$, which is impossible. □

Sometimes our algorithm uses yet another branching rule. Let $S \subseteq V$ be a (small) separator of the graph G, i.e. $G - S$ is disconnected. Then for any maximum independent set I of G, $I \cap S$ is an independent set of G. Thus we may branch into all possible independent sets of S.

Lemma 5. *Let G be a graph, let S be a separator of G and let $\mathcal{I}(S)$ be the set of all independent subsets of S. Then*

$$\alpha(G) = \max_{A \in \mathcal{I}(S)} |A| + \alpha(G - (S \cup N[A])).$$

Our algorithm uses the corresponding *separator branching* only under the following circumstances: the separator S is the set $N^2(v)$ and this set is of size at most 2. Thus the branching is done in at most 4 subproblems and for each of it is easy to find out the optimal choice among the vertices of $N[v]$.

The third Independent Set algorithm will be presented and analyzed during the talk.

5 Two Algorithms for SAT

First we present the rules of the DPLL algorithm to solve the SAT problem from the early sixties. This branching algorithm has triggered a lot of research in the SAT community and its ideas are used in modern SAT solvers.

Then we study the algorithm of Monien and Speckenmeyer which was the first one with a proven upper bound of $O^*(c^n)$ with $c < 2$ for 3-SAT. Indeed this algorithm solves $k - SAT$ for any fixed $k \geq 3$ in time $O^*({c_k}^n)$ with $c_k < 2$, where c_k depends on k.

The algorithm recursively computes CNF formulas obtained by a partial truth assignment of the input k-CNF formula, i.e. by fixing the boolean value of some variables and literals, respectively, of F. Given any partial truth assignment t of the k-CNF formula F the corresponding k-CNF formula F' is obtained by removing all clauses containing a true literal, and by removing all false literals. Hence the instance of any subproblem generated by the algorithm is a k-CNF formula. The size of a k-CNF formula is its number of variables.

We first study the branching rule of the algorithm. Let F be any k-CNF formula and let $c = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_t)$ be any clause of F. Branching on clause c means to branch into the following t subproblems obtained by fixing the boolean values of some literals as described below:

- F_1 : $\ell_1 = \text{true}$
- F_2 : $\ell_1 = \text{false}, \ell_2 = \text{true}$
- F_3 : $\ell_1 = \text{false}, \ell_2 = \text{false}, \ell_3 = \text{true}$
- F_t : $\ell_1 = \text{false}, \ell_2 = \text{false}, \cdots, \ell_{t-1} = \text{false}, \ell_t = \text{true}$

The branching rule says that F is satisfiable iff at least one F_i, $i = 1, 2, \ldots, t$ is satisfiable, and this obviously is correct. Hence recursively solving all subproblem instances F_i we can decide whether F is satisfiable.

Suppose F has n variables. Since the boolean values of i variables of F are fixed to obtain the instance F_i, $i = 1, 2, \ldots, t$, the number of (non fixed) variables of F_i is $n - i$. Therefore the branching vector of this rule

is $(1, 2, \ldots, t)$. To obtain the branching factor of $(1, 2, \ldots, t)$ we solve the linear recurrence

$$T(n) \leq T(n-1) + T(n-2) + \cdots + T(n-t)$$

by computing the unique positive real root of

$$x^t = x^{t-1} + x^{t-2} - x^{t-3} - \cdots 1 = 0,$$

which is equivalent to

$$x^{t+1} - 2x^t + 1 = 0.$$

For any clause of size t we denote the branching factor β_t. Then $\beta_2 \approx 1.6181$, $\beta_3 \approx 1.8393$, $\beta_4 \approx 1.9276$ and $\beta_5 \approx 1.9660$.

We note that on a clause of size 1, there is only one subproblem and thus this is indeed a reduction rule. By adding some simple reduction rules for termination saying that a formula containing an empty clause is unsatisfiable and that the empty formula is satisfiable we would obtain a first branching algorithm consisting essentially of the above branching rule. Of course we may also add the reduction rule saying that if the formula is in 2-CNF then a polynomial time algorithm will be used to decide whether it is satisfiable. The running time of such a simple branching algorithm is $O^*({\beta_k}^n)$ since given a k-CNF as input all instances generated by the branching algorithm are k-CNF, and thus every clause the algorithm branches on has size $t \leq k$.

Notice that the branching factor β_t depends on the size t of the clause c chosen to branch on. Hence it is natural to aim at branching on clauses of as small size as possible. Thus for every CNF formula being an instance of a subproblem the algorithm chooses a clause of minimum size to branch on. Using some nice logic insights one can guarantee that for an input k-CNF the algorithm always branches on a clause of size at most $k-1$ (except possibly the very first branching). Such a branching algorithm solves k-SAT in time $O^*({\alpha_k}^n)$ where $\alpha_k = \beta_{k-1}$. Hence the algorithm solves 3-SAT in time $O(1.6181^n)$.

6 Worst-Case Running Time and Lower Bounds

Lower bounds for the worst-case running time of branching algorithms are of interest since the current tools for the running time analysis of branching algorithms (including Measure & Conquer) seem not strong enough to establish the worst-case running time.

A lower bound of $\Omega^*(c^n)$ to the (unknown) worst-case running time of a particular branching algorithm is established by constructing instances and showing that the algorithm needs running time $\Omega^*(c^n)$ on those instances. Clearly the goal is that lower and upper bound of the worst-case running time of a particular algorithm are close.

Theorem 1. *The first Independent Set algorithm has worst case running time* $\Theta^*(\alpha^n)$*, where* α *is the branching factor of* $(1, 4)$*.*

Proof. We need to prove the lower bound $\Omega^*(\alpha^n)$. To do this, consider the graph $G_n = (\{1, 2, \ldots, n\}, E)$, where $\{i, j\} \in E \Leftrightarrow |i - j| \leq 3$. For this graph we assume that algorithm will solve ties by always choosing the leftmost remaining vertex to branch on, which always has degree 3. Hence on G_i, the algorithm branches into G_{i-1} and G_{i-4}. Thus if the search tree generated on G_n has $T(n)$ leaves then $T(n) \leq T(n-1)+T(n-4)$, and thus $T(n) = \Omega(\alpha^n)$. □

Suppose we modify the first Independent Set algorithm such that it branches on a maximum degree vertex. This will not change the upper bound analysis, however the lower bound does not apply anymore. Is this a coincidence or has the modified algorithm really a better worst-case running time?

7 Memorization

Memorization in branching algorithms has been introduced by M. Robson. The goal is to speed up the algorithm by storing already computed results in a database to look them up instead of recomputing them many times on different branches of the search tree.

The technique can be used to obtain algorithms with better upper bounds on the running time. Unfortunately the technique leads to algorithms needing exponential space, while the original branching algorithm needs only polynomial space.

8 Branch & Recharge

This is a new approach to construct and analyse branching algorithms. The key idea is to explicitely use weights in the algorithm to guarantee that the running time is governed by few recurrences; and thus running time analysis is easy. On the other hand, correctness is no longer obvious and needs a careful analysis of the branching algorithm. A typical operation of such an algorithm is a redistribution of the weights called recharging.

In the algorithm to be presented, every vertex is assigned a weight of 1 at the beginning. A value $\epsilon > 0$ is fixed depending on the problem. Then by a recharging procedure, it is guaranteed that in each branching on any vertex v the overall weight of the input decreases by 1 when not taking vertex v in the solution set S, and it decreases by $1 + \epsilon$ when selecting v in S. Hence the only branching vector of the algorithm is $(1, 1 + \epsilon)$ and one immediately obtains the running time.

Exercices

The exercices are ordered from easy ones to research problems.

1. The HAMILTONIAN CIRCUIT problem can be solved in time $O^*(2^n)$ via dynamic programming or inclusion-exclusion. Construct a $O^*(3^{m/3})$ branching algorithm deciding whether a graph has a hamiltonian circuit, where m is the number of edges.
2. Let $G = (V, E)$ be a bicolored graph, i.e. its vertices are either red or blue. Construct and analyze branching algorithms that for input G, k_1, k_2 decide whether the bicolored graph G has an independent set I with k_1 red and k_2 blue vertices. What is the best running time you can establish?
3. Construct a branching algorithm for the 3-COLORING problem, i.e. for given graph G it decides whether G is 3-colorable. The running time should be $O^*(3^{n/3})$ or even $O^*(c^n)$ for some $c < 1.4$.
4. Construct a branching algorithm for the DOMINATING SET problem on graphs of maximum degree 3.
5. Is the following statement true for all graphs G. If w is a mirror of v and there is a maximum independent set of G not containing v, then there is a maximum independent set containing neither v nor w.
6. Modify the first IS algorithm such that it always branches on a maximum degree vertex. Provide a lower bound. What is the worst-case running time of this algorithm?
7. Construct a $O^*(1.49^n)$ branching algorithm to solve 3-SAT.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press, 2001.

[2] J. Kleinberg, E. Tardos, *Algorithm Design*, Addison-Wesley, 2005.

[3] R.L. Graham, D.E. Knuth, O. Patashnik *Concrete Mathematics*, Addison-Wesley, 1989.

[4] R. Beigel and D. Eppstein. 3-coloring in time $O(1.3289^n)$. *Journal of Algorithms* 54:168–204, 2005.

[5] M. Davis, H. Putnam. A computing procedure for quantification theory. *J. ACM* 7:201–215, 1960.

[6] D. Eppstein. The traveling salesman problem for cubic graphs. In *Workshop on Algorithms and Data Structures (WADS)*, pages 307–318, 2003.

[7] F. V. Fomin, S. Gaspers, and A. V. Pyatkin. Finding a minimum feedback vertex set in time $O(1.7548^n)$, in Proceedings of the 2nd International Workshop on Parameterized and Exact Computation (IWPEC 2006), Springer LNCS vol. 4169, pp. 184–191.

[8] F. V. Fomin, P. Golovach, D. Kratsch, J. Kratochvil and M. Liedloff. Branch and Recharge: Exact algorithms for generalized domination. Proceedings of WADS 2007, Springer, 2007, LNCS 4619, pp. 508–519.

[9] F. V. Fomin, F. Grandoni, D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS* 87:47–77, 2005.

[10] F. V. Fomin, F. Grandoni, D. Kratsch. Measure and Conquer: A Simple $O(2^{0.288\,n})$ Independent Set Algorithm. Proceedings of SODA 2006, ACM and SIAM, 2006, pp. 508–519.

[11] K. Iwama. Worst-case upper bounds for k-SAT. *Bulletin of the EATCS* 82:61–71, 2004.

[12] K. Iwama and S. Tamaki. Improved upper bounds for 3-SAT. Proceedings of SODA 2004, ACM and SIAM, 2004, p. 328.

[13] B. Monien, E. Speckenmeyer. Solving satisfiability in less than 2^n steps. Discrete Appl. Math. 10: 287–295, 1985.

[14] I. Razgon. Exact computation of maximum induced forest. *Proceedings of the 10th Scandinavian Workshop on Algorithm Theory (SWAT 2006).* Springer LNCS vol. 4059, p. 160-171.

[15] J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms* 7(3):425–440, 1986.

[16] R. Tarjan and A. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

[17] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. *Combinatorial Optimization – Eureka, You Shrink*, Springer LNCS vol. 2570, 2003, pp. 185–207.

[18] G. J. Woeginger. Space and time complexity of exact algorithms: Some open problems. Proceedings of the *1st International Workshop on Parameterized and Exact Computation (IWPEC 2004)*, Springer LNCS vol. 3162, 2004, pp. 281–290.

[19] G. J. Woeginger. Open problems around exact algorithms. *Discrete Applied Mathematics*, 156:397–405, 2008.

Dániel Marx

FPT algorithmic techniques

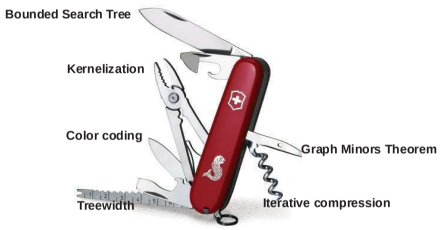
FPT algorithmic techniques

Dániel Marx
Budapest University of Technology and Economics, Hungary

AGAPE'09 Spring School on Fixed Parameter and Exact Algorithms
May 25-26, 2009, Lozari, Corsica (France)

FPT algorithmic techniques

- Significant advances in the past 20 years or so (especially in recent years).
- Powerful toolbox for designing FPT algorithms:



Goals

- Demonstrate techniques that were successfully used in the analysis of parameterized problems.
- There are two goals:
  - Determine quickly if a problem is FPT.
  - Design fast algorithms.
- Warning: The results presented for particular problems are not necessarily the best known results or the most useful approaches for these problems.
- Conventions:
  - Unless noted otherwise, k is the parameter.
  - O^* notation: $O^*(f(k))$ means $O(f(k) \cdot n^c)$ for some constant c.
  - Citations are mostly omitted (only for classical results).
  - We gloss over the difference between decision and search problems.

Kernelization

Kernelization

Definition: **Kernelization** is a polynomial-time transformation that maps an instance (I, k) to an instance (I', k') such that

- (I, k) is a yes-instance if and only if (I', k') is a yes-instance,
- $k' \leq k$, and
- $|I'| \leq f(k)$ for some function $f(k')$.

Simple fact: If a problem has a kernelization algorithm, then it is FPT.
Proof: Solve the instance (I', k') by brute force.

Converse: Every FPT problem has a kernelization algorithm.
Proof: Suppose there is an $f(k)n^c$ algorithm for the problem.

- If $f(k) \leq n$, then solve the instance in time $f(k)n^c \leq n^{c+1}$, and output a trivial yes- or no-instance.
- If $n < f(k)$, then we are done: a kernel of size $f(k)$ is obtained.

Kernelization for VERTEX COVER

General strategy: We devise a list of reduction rules, and show that if none of the rules can be applied and the size of the instance is still larger than $f(k)$, then the answer is trivial.

Reduction rules for VERTEX COVER instance (G, k):

Rule 1: If v is an isolated vertex $\Rightarrow (G \setminus v, k)$
Rule 2: If $d(v) > k \Rightarrow (G \setminus v, k-1)$

If neither Rule 1 nor Rule 2 can be applied:

- If $|V(G)| > k(k+1) \Rightarrow$ There is no solution (every vertex should be the neighbor of at least one vertex of the cover).
- Otherwise, $|V(G)| \leq k(k+1)$ and we have a $k(k+1)$ vertex kernel.

Kernelization for VERTEX COVER

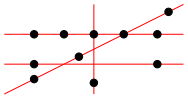
Let us add a third rule:

Rule 1: If v is an isolated vertex $\Rightarrow (G \setminus v, k)$
Rule 2: If $d(v) > k \Rightarrow (G \setminus v, k-1)$
Rule 3: If $d(v) = 1$, then we can assume that its neighbor u is in the solution $\Rightarrow (G \setminus (u \cup v), k-1)$.

If none of the rules can be applied, then every vertex has degree at least 2.
Now $|E(G)| \leq k^2$ implies $|V(G)| \leq k^2$.

COVERING POINTS WITH LINES

Task: Given a set P of n points in the plane and an integer k, find k lines that cover all the points.



Note: We can assume that every line of the solution covers at least 2 points, thus there are at most n^2 candidate lines.

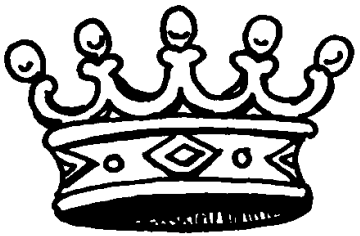
Reduction Rule:
If a candidate line covers a set S of more than k points $\Rightarrow (P \setminus S, k-1)$.

If this rule cannot be applied and there are still more than k^2 points, then there is no solution $\Rightarrow$ Kernel with at most k^2 points.

Kernelization

- Kernelization can be thought of as a polynomial-time preprocessing before attacking the problem with whatever method we have. "It does no harm" to try kernelization.
- Some kernelizations use lots of simple reduction rules and require a complicated analysis to bound the kernel size. . .
- . . . while other kernelizations are based on surprising nice tricks (Next: Crown Reduction and the Sunflower Lemma).
- Possibility to prove lower bounds (S. Saurabh's lecture).

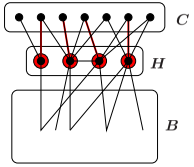
Crown Reduction



Crown Reduction

Definition: A **crown decomposition** is a partition $C \cup H \cup B$ of the vertices such that

- C is an independent set,
- there is no edge between C and B,
- there is a matching between C and H that covers H.



Crown rule for VERTEX COVER:
The matching needs to be covered and we can assume that it is covered by H (makes no sense to use vertices of C)
$\Rightarrow (G \setminus (H \cup C), k - |H|)$.

Crown Reduction

Key lemma:

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$, $\Rightarrow$ No solution!
- find a crown decomposition, $\Rightarrow$ Reduce!
- or conclude that the graph has at most $3k$ vertices.
  $\Rightarrow 3k$ vertex kernel!

This gives a 3k vertex kernel for VERTEX COVER.

Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

For the proof, we need the classical Kőnig's Theorem.

$\tau(G)$: size of the minimum vertex cover
$\nu(G)$: size of the maximum matching (independent set of edges)

Theorem: [Kőnig, 1931] If G is **bipartite,** then

$$\tau(G) = \nu(G)$$

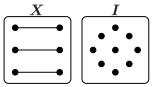
Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof: Find (greedily) a maximal matching; if its size is at least $k+1$, then we are done. The rest of the graph is an independent set I.

Find a maximum matching/minimum vertex cover in the bipartite graph between X and I.



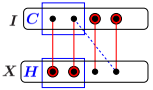
Proof

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

- find a matching of size $k+1$,
- find a crown decomposition,
- or conclude that the graph has at most $3k$ vertices.

Proof: Case 1: The minimum vertex cover contains at least one vertex of X
$\Rightarrow$ There is a crown decomposition.

Case 2: The minimum vertex cover contains only vertices of $I \Rightarrow$ It contains every vertex of I
$\Rightarrow$ There are at most $2k+k$ vertices.



Dual of Vertex Coloring

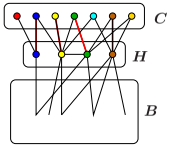
Also known as Saving k Colors.

Task: Given a graph G and an integer k, find a vertex coloring with $|V(G)| - k$ colors.

Crown rule for Dual of Vertex Coloring:

Suppose there is a crown decomposition for the **complement graph** $\overline{G}$.

- C is a clique in G: each vertex needs a distinct color.
- Because of the matching, H can be colored using only these $|C|$ colors.
- These colors cannot be used for B.
- $(G \setminus (H \cup C), k - |H|)$



Crown Reduction for DUAL OF VERTEX COLORING

Use the key lemma for the complement $\overline{G}$ of G:

Lemma: Given a graph G without isolated vertices and an integer k, in polynomial time we can either

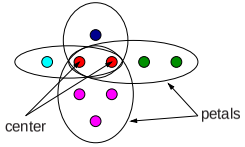
- find a matching of size $k+1$, $\Rightarrow$ YES: we can save k colors!
- find a crown decomposition, $\Rightarrow$ Reduce!
- or conclude that the graph has at most $3k$ vertices.
  $\Rightarrow$ $3k$ vertex kernel!

This gives a 3k vertex kernel for DUAL OF VERTEX COLORING.

Sunflower Lemma

Sunflower lemma

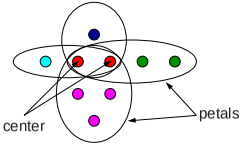
Definition: Sets $S_1, S_2, \ldots, S_k$ form a **sunflower** if the sets $S_i \setminus (S_1 \cap S_2 \cap \cdots \cap S_k)$ are disjoint.



Lemma: [Erdős and Rado, 1960] If the size of a set system is greater than $(p-1)^d \cdot d!$ and it contains only sets of size at most d, then the system contains a sunflower with p petals. Furthermore, in this case such a sunflower can be found in polynomial time.

Sunflowers and d-HITTING SET

d-HITTING SET: Given a collection $\mathcal{S}$ of sets of size at most d and an integer k, find a set S of k elements that intersects every set of $\mathcal{S}$.

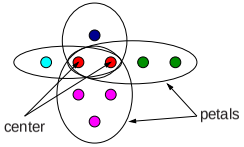


Reduction Rule: If $k+1$ sets form a sunflower, then remove these sets from $\mathcal{S}$ and add the center C to $\mathcal{S}$ (S does not hit one of the petals, thus it has to hit the center).

If the rule cannot be applied, then there are at most $k^d \cdot d! = O(k^d)$ sets.

Sunflowers and d-HITTING SET

d-HITTING SET: Given a collection $\mathcal{S}$ of sets of size at most d and an integer k, find a set S of k elements that intersects every set of $\mathcal{S}$.



Reduction Rule (variant): Suppose $k+1$ sets form a sunflower.

- If the sets are disjoint $\Rightarrow$ No solution.
- Otherwise, remove one of the sets.

If the rule cannot be applied, then there are at most $k^d \cdot d! = O(k^d)$ sets.

Graph Minors

Neil Robertson

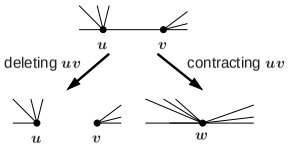
Paul Seymour

Graph Minors

- Some consequences of the Graph Minors Theorem give a quick way of showing that certain problems are FPT.
- However, the function $f(k)$ in the resulting FPT algorithms can be HUGE, completely impractical.
- History: motivation for FPT.
- Parts and ingredients of the theory are useful for algorithm design.
- New algorithmic results are still being developed.

Graph Minors

Definition: Graph H is a **minor** G ($H \leq G$) if H can be obtained from G by deleting edges, deleting vertices, and contracting edges.

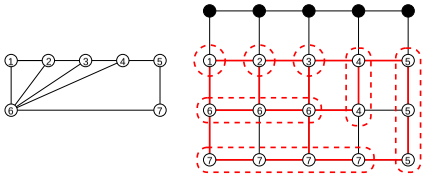


Example: A triangle is a minor of a graph G if and only if G has a cycle (i.e., it is not a forest).

Graph minors

Equivalent definition: Graph H is a **minor** of G if there is a mapping ϕ that maps each vertex of H to a connected subset of G such that

- $\phi(u)$ and $\phi(v)$ are disjoint if $u \neq v$, and
- if $uv \in E(G)$, then there is an edge between $\phi(u)$ and $\phi(v)$.



Minor closed properties

Definition: A set $\mathcal{G}$ of graphs is **minor closed** if whenever $G \in \mathcal{G}$ and $H \leq G$, then $H \in \mathcal{G}$ as well.

Examples of minor closed properties:

planar graphs
acyclic graphs (forests)
graphs having no cycle longer than k
empty graphs

Examples of not minor closed properties:

complete graphs
regular graphs
bipartite graphs

Forbidden minors

Let $\mathcal{G}$ be a minor closed set and let $\mathcal{F}$ be the set of "minimal bad graphs": $H \in \mathcal{F}$ if $H \notin \mathcal{G}$, but every proper minor of H is in $\mathcal{G}$.

Characterization by forbidden minors:

$$G \in \mathcal{G} \iff \forall H \in \mathcal{F}, H \not\leq G$$

The set $\mathcal{F}$ is the **obstruction set** of property $\mathcal{G}$.

Theorem: [Wagner] A graph is planar if and only if it does not have a K_5 or $K_{3,3}$ minor.

In other words: the obstruction set of planarity is $\mathcal{F} = \{K_5, K_{3,3}\}$.

Does every minor closed property have such a finite characterization?

Graph Minors Theorem

Theorem: [Robertson and Seymour] Every minor closed property $\mathcal{G}$ has a finite obstruction set.

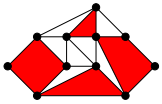
Note: The proof is contained in the paper series "Graph Minors I–XX".
Note: The size of the obstruction set can be astronomical even for simple properties.

Theorem: [Robertson and Seymour] For every fixed graph H, there is an $O(n^3)$ time algorithm for testing whether H is a minor of the given graph G.

Corollary: For every minor closed property $\mathcal{G}$, there is an $O(n^3)$ time algorithm for testing whether a given graph G is in $\mathcal{G}$.

Applications

PLANAR FACE COVER: Given a graph G and an integer k, find an embedding of planar graph G such that there are k faces that cover all the vertices.



One line argument:

For every fixed k, the class $\mathcal{G}_k$ of graphs of yes-instances is minor closed.

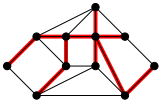
$\Downarrow$

For every fixed k, there is a $O(n^3)$ time algorithm for PLANAR FACE COVER.

Note: non-uniform FPT.

Applications

k-LEAF SPANNING TREE: Given a graph G and an integer k, find a spanning tree with **at least** k leaves.



Technical modification: Is there such a spanning tree for at least one component of G?

One line argument:

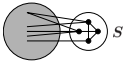
For every fixed k, the class $\mathcal{G}_k$ of no-instances is minor closed.

$\Downarrow$

For every fixed k, k-LEAF SPANNING TREE can be solved in time $O(n^3)$.

$\mathcal{G} + k$ vertices

Let $\mathcal{G}$ be a graph property, and let $\mathcal{G} + kv$ contain graph G if there is a set $S \subseteq V(G)$ of k vertices such that $G \setminus S \in \mathcal{G}$.



Lemma: If $\mathcal{G}$ is minor closed, then $\mathcal{G} + kv$ is minor closed for every fixed k.
$\Rightarrow$ Finding the smallest k such that a given graph is in $\mathcal{G} + kv$ is FPT.

- If $\mathcal{G}$ = forests $\Rightarrow$ $\mathcal{G} + kv$ = graphs that can be made acyclic by the deletion of k vertices $\Rightarrow$ FEEDBACK VERTEX SET is FPT.
- If $\mathcal{G}$ = planar graphs $\Rightarrow$ $\mathcal{G} + kv$ = graphs that can be made planar by the deletion of k vertices (k-apex graphs) $\Rightarrow$ k-APEX GRAPH is FPT.
- If $\mathcal{G}$ = empty graphs $\Rightarrow$ $\mathcal{G} + kv$ = graphs with vertex cover number at most k $\Rightarrow$ VERTEX COVER is FPT.

Two types of problems

We have to solve some problems.

Typically **minimization** problems: VERTEX COVER, HITTING SET, DOMINATING SET, covering/stabbing problems, graph modification problems, ...

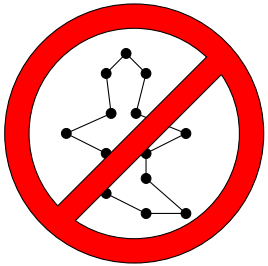
Bounded search trees, iterative compression

We have to find something nice hidden somewhere.

Typically **maximization** problems: k-PATH, DISJOINT TRIANGLES, k-LEAF SPANNING TREE, ...

Color coding, matroids

Forbidden subgraphs



Forbidden subgraphs

General problem class: Given a graph G and an integer k, transform G with at most k modifications (add/remove vertices/edges) into a graph having property $\mathcal{P}$.

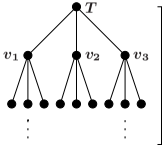
Example:
TRIANGLE DELETION: make the graph triangle-free by deleting at most k vertices.

Branching algorithm:

6. If the graph is triangle-free, then we are done.
6. If there is a triangle $v_1v_2v_3$, then at least one of v_1, v_2, v_3 has to be deleted $\Rightarrow$ We branch into 3 directions.

TRIANGLE DELETION

Search tree:



height $\leq k+1$

The search tree has at most 3^k leaves and the work to be done is polynomial at each step $\Rightarrow O^*(3^k)$ time algorithm.

Note: If the answer is "NO", then the search tree has **exactly** 3^k leaves.

Hereditary properties

Definition: A graph property $\mathcal{P}$ is **hereditary** if for every $G \in \mathcal{P}$ and induced subgraph G' of G, we have $G' \in \mathcal{P}$ as well.

Examples: triangle-free, bipartite, interval graph, planar

Observation: Every hereditary property $\mathcal{P}$ can be characterized by a (finite or infinite) set $\mathcal{F}$ of forbidden induced subgraphs:

$$G \in \mathcal{P} \Leftrightarrow \forall H \in \mathcal{F}, H \not\subseteq_{\text{ind}} G$$

Theorem: If $\mathcal{P}$ is hereditary and can be characterized by a **finite** set $\mathcal{F}$ of forbidden induced subgraphs, then the graph modification problems corresponding to $\mathcal{P}$ are FPT.

Hereditary properties

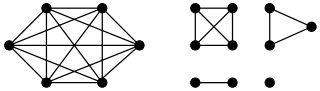
Theorem: If $\mathcal{P}$ is hereditary and can be characterized by a **finite** set $\mathcal{F}$ of forbidden induced subgraphs, then the graph modification problems corresponding to $\mathcal{P}$ are FPT.

Proof:

- Suppose that every graph in $\mathcal{F}$ has at most r vertices. Using brute force, we can find in time $O(n^r)$ a forbidden subgraph (if exists).
- If a forbidden subgraph exists, then we have to delete one of the at most r vertices or add/delete one of the at most $\binom{r}{2}$ edges $\Rightarrow$ Branching factor is a constant c depending on $\mathcal{F}$.
- The search tree has at most c^k leaves and the work to be done at each node is $O(n^r)$.

CLUSTER EDITING

Task: Given a graph G and an integer k, add/remove at most k edges such that every component is a clique in the resulting graph.



Property $\mathcal{P}$: every component is a clique.

Forbidden induced subgraph:

$O^*(3^k)$ time algorithm.

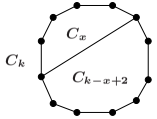
CHORDAL COMPLETION

Definition: A graph is **chordal** if it does not contain an induced cycle of length greater than 3.

CHORDAL COMPLETION: Given a graph G and an integer k, add at most k edges to G to make it a chordal graph.

The forbidden induced subgraphs are the cycles of length greater 3
$\Rightarrow$ Not a finite set!

Lemma: At least $k-3$ edges are needed to make a k-cycle chordal.
Proof: By induction. $k=3$ is trivial.



C_x: $x-3$ edges
C_{k-x+2}: $k-x-1$ edges
C_k: $(x-3)+(k-x-1)+1 =$
$k-3$ edges

CHORDAL COMPLETION

Algorithm:

- Find an induced cycle C of length at least 4 (can be done in polynomial time).
- If no such cycle exists $\Rightarrow$ Done!
- If C has more than $k+3$ vertices $\Rightarrow$ No solution!
- Otherwise, one of the

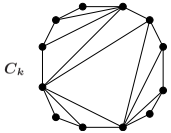
$$\binom{|C|}{2} - |C| \leq (k+3)(k+2)/2 - k = O(k^2)$$

  missing edges has to be added $\Rightarrow$ Branch!

Size of the search tree is $k^{O(k)}$.

CHORDAL COMPLETION – *more efficiently*

Definition: Triangulation of a cycle.



Lemma: Every chordal supergraph of a cycle C contains a triangulation of the cycle C.

Lemma: The number of ways a cycle of length k can be triangulated is exactly the $(k-2)$th Catalan number

$$C_{k-2} = \frac{1}{k-1}\binom{2(k-2)}{k-2} \leq 4^k.$$

CHORDAL COMPLETION – *more efficiently*

Algorithm:

- Find an induced cycle C of length at least 4 (can be done in polynomial time).
- If no such cycle exists $\Rightarrow$ Done!
- If C has more than $k+3$ vertices $\Rightarrow$ No solution!
- Otherwise, one of the $\leq 4^{|C|}$ triangulations has to be in the solution $\Rightarrow$ Branch!

Claim: Search tree has at most $T_k = 4^k$ leaves.
Proof: By induction. Number of leaves is at most

$$T_k \leq 4^{|C|} \cdot T_{k-|C|} \leq 4^{|C|} \cdot 4^{k-|C|} = 4^k.$$

Iterative compression

Iterative compression

- A surprising small, but very powerful trick.
- Most useful for deletion problems: delete k things to achieve some property.
- Demonstration: ODD CYCLE TRANSVERSAL aka BIPARTITE DELETION aka GRAPH BIPARTIZATION: Given a graph G and an integer k, delete k vertices to make the graph bipartite.
- Forbidden induced subgraphs: odd cycles. There is no bound on the size of odd cycles.

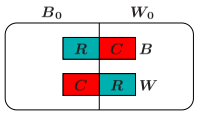
BIPARTITE DELETION

Solution based on iterative compression:

- **Step 1:**
  Solve the **annotated problem** for bipartite graphs:
  Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.
- **Step 2:**
  Solve the **compression problem** for general graphs:
  Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.
- **Step 3:**
  Apply the magic of iterative compression...

Step 1: The annotated problem

Given a bipartite graph G, two sets $B, W \subseteq V(G)$, and an integer k, find a set S of at most k vertices such that $G \setminus S$ has a 2-coloring where $B \setminus S$ is black and $W \setminus S$ is white.



Find an arbitrary 2-coloring (B_0, W_0) of G.
$C := (B_0 \cap W) \cup (W_0 \cap B)$ should change color, while
$R := (B_0 \cap B) \cup (W_0 \cap W)$ should remain the same color.

Lemma: $G \setminus S$ has the required 2-coloring if and only if S separates C and R, i.e., no component of $G \setminus S$ contains vertices from both $C \setminus S$ and $R \setminus S$.

Step 1: The annotated problem

Lemma: $G \setminus S$ has the required 2-coloring if and only if S separates C and R, i.e., no component of $G \setminus S$ contains vertices from both $C \setminus S$ and $R \setminus S$.

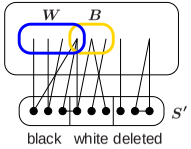
Proof:
$\Rightarrow$ In a 2-coloring of $G \setminus S$, each vertex either remained the same color or changed color. Adjacent vertices do the same, thus every component either changed or remained.

$\Leftarrow$ Flip the coloring of those components of $G \setminus S$ that contain vertices from $C \setminus S$. No vertex of R is flipped.

Algorithm: Using max-flow min-cut techniques, we can check if there is a set S that separates C and R. It can be done in time $O(k|E(G)|)$ using k iterations of the Ford-Fulkerson algorithm.

Step 2: The compression problem

Given a graph G, an integer k, and a set S' of $k+1$ vertices such that $G \setminus S'$ is bipartite, find a set S of k vertices such that $G \setminus S$ is bipartite.



Branch into 3^{k+1} cases: each vertex of S' is either black, white, or deleted.
Trivial check: no edge between two black or two white vertices.
Neighbors of the black vertices in S' should be white and the neighbors of the white vertices in S' should be black.

Step 3: Iterative compression

How do we get a solution of size $k+1$?

We get it for free!

Let $V(G) = \{v_1, \ldots, v_n\}$ and let G_i be the graph induced by $\{v_1, \ldots, v_i\}$.

For every i, we find a set S_i of size k such that $G_i \setminus S_i$ is bipartite.

- For G_k, the set $S_k = \{v_1, \ldots, v_k\}$ is a trivial solution.
- If S_{i-1} is known, then $S_{i-1} \cup \{v_i\}$ is a set of size $k+1$ whose deletion makes G_i bipartite $\Rightarrow$ We can use the compression algorithm to find a suitable S_i in time $O(3^k \cdot k|E(G_i)|)$.

Step 3: Iterative Compression

Bipartite-Deletion(G, k)

1. $S_k = \{v_1, \ldots, v_k\}$
2. for $i := k+1$ to n
3. Invariant: $G_{i-1} \setminus S_{i-1}$ is bipartite.
4. Call Compression$(G_i, S_{i-1} \cup \{v_i\})$
5. If the answer is "NO" $\Rightarrow$ return "NO"
6. If the answer is a set $X \Rightarrow S_i := X$
7. Return the set S_n

Running time: the compression algorithm is called n times and everything else can be done in linear time
$\Rightarrow O(3^k \cdot k|V(G)| \cdot |E(G)|)$ time algorithm.

Color coding

Color coding

- Works best when we need to ensure that a small number of "things" are disjoint.
- We demonstrate it on two problems:
  - Find an s-t path of length exactly k.
  - Find k vertex-disjoint triangles in a graph.
- Randomized algorithm, but can be derandomized using a standard technique.
- Very robust technique, we can use it as an "opening step" when investigating a new problem.

k-PATH

Task: Given a graph G, an integer k, two vertices s, t, find a **simple** s-t path with exactly k internal vertices.

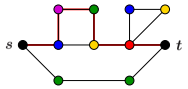
Note: Finding such a **walk** can be done easily in polynomial time.

Note: The problem is clearly NP-hard, as it contains the s-t HAMILTONIAN PATH problem.

The k-PATH algorithm can be used to check if there is a cycle of length exactly k in the graph.

k-PATH

- Assign colors from $[k]$ to vertices $V(G) \setminus \{s,t\}$ uniformly and independently at random.



- Check if there is a **colorful** s-t path: a path where each color appears exactly once on the internal vertices; output "YES" or "NO".
  - If there is no s-t k-path: no such colorful path exists $\Rightarrow$ "NO".
  - If there is an s-t k-path: the probability that such a path is colorful is

$$\frac{k!}{k^k} > \frac{(\frac{k}{e})^k}{k^k} = e^{-k},$$

thus the algorithm outputs "YES" with at least that probability.

Error probability

- If there is a k-path, the probability that the algorithm **does not** say "YES" after e^k repetitions is at most

$$(1-e^{-k})^{e^k} < \left(e^{-e^{-k}}\right)^{e^k} = 1/e \approx 0.38$$

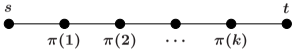
- Repeating the whole algorithm a constant number of times can make the error probability an arbitrary small constant.
- For example, by trying $100 \cdot e^k$ random colorings, the probability of a wrong answer is at most $1/e^{100}$.

It remains to see how a colorful s-t path can be found.

Method 1: Trying all permutations.
Method 2: Dynamic programming.

Method 1: Trying all permutations

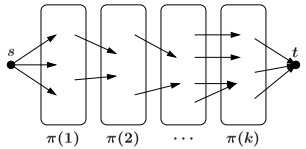
The colors encountered on a colorful s-t path form a permutation π of $\{1,2,\ldots,k\}$:



We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.

Method 1: Trying all permutations

We try all possible $k!$ permutations. For a fixed π, it is easy to check if there is a path with this order of colors.



- Edges connecting nonadjacent color classes are removed.
- The remaining edges are directed.
- All we need to check if there is a directed s-t path.
- Running time is $O(k! \cdot |E(G)|)$.

Method 2: Dyanamic Programming

We introduce $2^k \cdot |V(G)|$ Boolean variables:

$x(v, C) =$ TRUE for some $v \in V(G)$ and $C \subseteq [k]$
$\Updownarrow$
There is an s-v path where each color in C appears exactly once and no other color appears.

Clearly, $x(s, \emptyset) =$ TRUE. Recurrence for vertex v with color r:

$$x(v, C) = \bigvee_{uv \in E(G)} x(u, C \setminus \{r\})$$

If we know every $x(v, C)$ with $|C| = i$, then we can determine every $x(v, C)$ with $|C| = i + 1 \Rightarrow$ All the values can be determined in time $O(2^k \cdot |E(G)|)$.

There is a colorful s-t path $\Leftrightarrow x(v, [k]) =$ TRUE for some neighbor of t.

Derandomization

Using Method 2, we obtain a $O^*((2e)^k)$ time algorithm with constant error probability. How to make it deterministic?

Definition: A family $\mathcal{H}$ of functions $[n] \to [k]$ is a **k-perfect** family of hash functions if for every $S \subseteq [n]$ with $|S| = k$, there is a $h \in \mathcal{H}$ such that $h(x) \neq h(y)$ for any $x, y \in S$, $x \neq y$.

Instead of trying $O(e^k)$ random colorings, we go through a k-perfect family $\mathcal{H}$ of functions $V(G) \to [k]$. If there is a solution $\Rightarrow$ The internal vertices S are colorful for at least one $h \in \mathcal{H} \Rightarrow$ Algorithm outputs "YES".

Theorem: There is a k-perfect family of functions $[n] \to [k]$ having size $2^{O(k)} \log n$.

$\Rightarrow$ There is a **deterministic** $2^{O(k)} \cdot n^{O(1)}$ time algorithm for the k-PATH problem.

k-DISJOINT TRIANGLES

Task: Given a graph G and an integer k, find k vertex disjoint triangles.

Step 1: Choose a random coloring $V(G) \to [3k]$.

Step 2: Check if there is a colorful solution, where the $3k$ vertices of the k triangles use distinct colors.

- **Method 1:** try every permutation π of $[3k]$ and check if there are triangles with colors $(\pi(1), \pi(2), \pi(3)), (\pi(4), \pi(5), \pi(6)), \ldots$
- **Method 2:** dynamic programming. For $C \subseteq [3k]$ and $|C| = 3i$, let $x(C) =$ TRUE if and only if there are $|C|/3$ disjoint triangles using exactly the colors in C.

$$x(C) = \bigvee_{\{c_1, c_2, c_3\} \subseteq C} (x(C \setminus \{c_1, c_2, c_3\}) \wedge \exists \triangle \text{ with colors } c_1, c_2, c_3)$$

k-DISJOINT TRIANGLES

Step 3: Colorful solution exists with probability at least e^{-3k}, which is a lower bound on the probability of a correct answer.

Running time: constant error probability after e^{3k} repetitions $\Rightarrow$ running time is $O^*((2e)^{3k})$ (using Method 2).

Derandomization: $3k$-perfect family of functions instead of random coloring. Running time is $2^{O(k)} \cdot n^{O(1)}$.

Color coding

We have seen that color coding can be used to find paths, cycles of length k, or a set of k disjoint triangles.

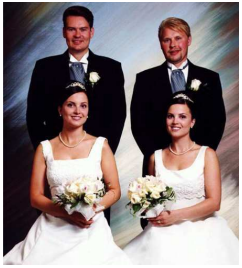
What other structures can be found efficiently with this technique?

The key is treewidth:

Theorem: Given two graph H, G, it can be decided if H is a subgraph of G in time $2^{O(|V(H)|)} \cdot |V(G)|^{O(w)}$, where w is the treewidth of G.

Thus if H belongs to a class of graphs with bounded treewidth, then the subgraph problem is FPT.

Matroid Theory



Matroid Theory

- Matroids: a classical subject of combinatorial optimization.
- Matroids lurk behind matching, flow, spanning tree, and some linear algebra problems.
- A general FPT result that can be used to show that some concrete problems are FPT.

Matroids

Definition: A set system $\mathcal{M}$ over E is a **matroid** if

(1) $\emptyset \in \mathcal{M}$.

(2) If $X \in M$ and $Y \subseteq X$, then $Y \in \mathcal{M}$.

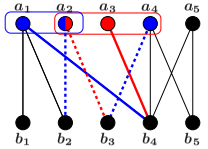
(3) If $X, Y \in M$ and $|X| > |Y|$, then $\exists e \in X$ such that $Y \cup \{e\} \in \mathcal{M}$.

Example: $\mathcal{M} = \{\emptyset, 1, 2, 3, 12, 13\}$ is a matroid.
Example: $\mathcal{M} = \{\emptyset, 1, 2, 12, 3\}$ is not a matroid.

If $x \in \mathcal{M}$, then we say that X is **independent** in matroid $\mathcal{M}$.

Transversal matroid

Fact: Let $G(A, B; E)$ be a bipartite graph. Those subsets of A that can be covered by a matching form a matroid.



(1) The empty set can be clearly covered.
(2) If X can be covered, then every subset $Y \subseteq X$ can be covered.
(3) Suppose $|X| > |Y|$ and they are covered by matchings M_X and M_Y, respectively. There is a component of $M_X \cup M_Y$ containing more red edges than blue edges. We can augment M_Y along this path.

Linear matroids

Fact: Let A be matrix and let E be the set of column vectors in A. The subsets $E' \subseteq E$ that are linearly independent form a matroid.

Proof:
(1) and (2) are clear.

(3) If $|X| > |Y|$ and both of them are linearly independent, then X spans a subspace with larger dimension than Y. Thus X contains a vector v not spanned by $Y \Rightarrow Y \cup \{v\}$ is linearly independent.

Example:

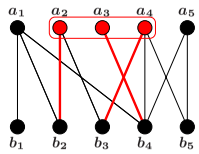
$$\begin{array}{c} a \quad b \quad c \quad d \\ \begin{pmatrix} 1 & 0 & 2 & 3 \\ 0 & 1 & 4 & 6 \end{pmatrix} \end{array} \Rightarrow \mathcal{M} = \{\emptyset, a, b, c, d, ab, ac, ad, bc, bd\}$$

Representation

- If $\mathcal{M}$ is the matroid of the columns of a matrix A, then A is a **representation** of M.
- If A is a matrix over a field $\mathbb{F}$, then $\mathcal{M}$ is **representable** over $\mathbb{F}$.
- If $\mathcal{M}$ is representable over some field $\mathbb{F}$, then $\mathcal{M}$ is **linear.**
- There are non-linear matroids (i.e., they cannot be represented over any field).

Transversal matroids are linear

Fact: Let $G(A, B; E)$ be a bipartite graph. Those subsets of A that can be covered by a matching form a linear matroid.



	a_1	a_2	a_3	a_4	a_5
b_1	?	0	0	0	0
b_2	?	?	0	0	0
b_3	0	?	0	?	0
b_4	?	0	?	?	?
b_5	0	0	0	?	?

Construct the bipartite adjacency matrix: if a_i and b_j are neighbors, then the i-th element of row j is a random integer between 1 and N.

Elements can be matched $\Rightarrow$ The determinant is nonzero with high probability (Schwartz-Zippel)

FPT result

Main result: Let $\mathcal{M}$ be a linear matroid over E, given by a representation A. Let $\mathcal{S}$ be a collection of subsets of E, each of size at most ℓ. It can be decided in randomized time $f(k, \ell) \cdot n^{O(1)}$ whether $\mathcal{M}$ has an independent set that is the union of k disjoint sets from $\mathcal{S}$.

Immediate application: k-DISJOINT TRIANGLES is (randomized) FPT (let $\mathcal{S}$ be the set of all triangles in the graph).

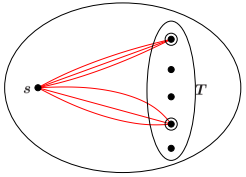
Two not so obvious applications:

- RELIABLE TERMINALS
- ASSIGNMENT WITH COUPLES

RELIABLE TERMINALS

Let D be a directed graph with a source vertex s and a subset T of vertices.

Task: Select k terminals $t_1, \ldots, t_k \in T$, and ℓ paths from s to each t_i such that these $k \cdot \ell$ paths are pairwise internally vertex disjoint.

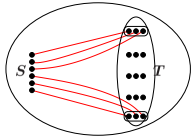


$k = 2, \ell = 3$

Theorem: The problem can be solved in randomized time $f(k, \ell) \cdot n^{O(1)}$.

RELIABLE TERMINALS

A technical trick: replace each $t \in T$ with ℓ copies, and replace s with a set S of $k \cdot \ell$ copies.



$k = 2, \ell = 3$

Now if a terminal t is selected, then we should connect the ℓ copies of t with ℓ different vertices of S.

Fact: [Perfect] Let D be a directed graph and S a subset of vertices. Those subsets X that can be reached from S by disjoint paths form a matroid.

RELIABLE TERMINALS

Fact: [Perfect] Let D be a directed graph and S a subset of vertices. Those subsets X that can be reached from S by disjoint paths form a matroid.

The problem is equivalent to finding k blocks whose union is independent in this matroid $\Rightarrow$ We can solve it in randomized time $f(k,\ell) \cdot n^{O(1)}$.

The matroid is actually a transversal matroid of an appropriately defined bipartite graph, hence it is linear and we can construct a representation for it.

ASSIGNMENT WITH COUPLES

Task: Assign people to jobs (bipartite matching).

However, the set of people includes couples and the members of a couple cannot be assigned independently (say, they want to be in the same town).

Task: Given

- a set of singles and a list of suitable jobs for each single,
- a set of couples and a list of suitable pairs of jobs for each couple,

assign a job to each single and a pair of jobs to each couple.

Theorem: ASSIGNMENT WITH COUPLES is randomized FPT parameterized by the number k of couples.

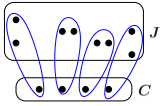
ASSIGNMENT WITH COUPLES

J: jobs, S: singles, C: couples

Let $X \subseteq J$ be in $\mathcal{M}$ if and only if S has a matching with $J \setminus X$.

Lemma: $\mathcal{M}$ is matroid.

Let $\mathcal{M}'$ be the matroid over $J \cup C$ such that $X \in \mathcal{M}' \Leftrightarrow X \cap J \in \mathcal{M}$.



For each couple $c \in C$ and suitable pair $\{j_1, j_2\}$, add triple $\{c, j_1, j_2\}$ to $\mathcal{S}$.

The k couples and all the singles can be a assigned a job

$\Updownarrow$

There are k disjoint triples in $\mathcal{S}$ whose union is independent in $\mathcal{M}'$

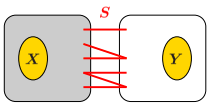
Cut problems

MULTIWAY CUT

Task: Given a graph G, a set T of vertices, and an integer k, find a set S of at most k edges that separates T (each component of $G \setminus S$ contains at most one vertex of T).

Polynomial for $|T| = 2$, but NP-hard for $|T| = 3$.

Theorem: MULTIWAY CUT is FPT parameterized by k.



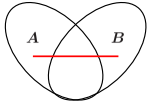
$\delta(R)$: set of edges leaving R
$\lambda(X, Y)$: minimum number of edges in an (X, Y)-separator

Submodularity

Fact: The function δ is **submodular:** for arbitrary sets A, B,

$$|\delta(A)| + |\delta(B)| \geq |\delta(A \cap B)| + |\delta(A \cup B)|$$

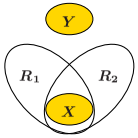
Proof: Determine separately the contribution of the different types of edges.



Submodularity

Consequence: There is a unique maximal $R_{\max} \supseteq X$ such that $\delta(R_{\max})$ is an (X, Y)-separator of size $\lambda(X, Y)$.

Proof: Let $R_1, R_2 \supseteq X$ be two sets such that $\delta(R_1), \delta(R_2)$ are (X, Y)-separators of size $\lambda := \lambda(X, Y)$.



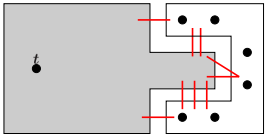
$$\underset{\lambda}{|\delta(R_1)|} + \underset{\lambda}{|\delta(R_2)|} \geq \underset{\geq \lambda}{|\delta(R_1 \cap R_2)|} + |\delta(R_1 \cup R_2)|$$

$$\Rightarrow |\delta(R_1 \cup R_2)| \leq \lambda$$

Note: Analogous result holds for a unique minimal $R_{\min}$.

MULTIWAY CUT

Intuition: Consider a $t \in T$. A subset of the solution separates t and $T \setminus \{t\}$.

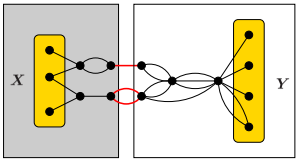


There are many such separators.

But a separator farther from t and closer to $T \setminus \{t\}$ seems to be more useful.

Important separators

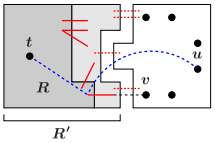
Definition: An (X, Y)-separator $\delta(R)$ $(R \supseteq X)$ is **important** if there is no (X, Y)-separator $\delta(R')$ with $R \subset R'$ and $|\delta(R')| \leq |\delta(R)|$.



Important separators

Lemma: Let $t \in T$. The MULTIWAY CUT problem has a solution S such that S contains an important $(t, T \setminus \{t\})$-separator.

Proof: Let R be the vertices reachable from t in $G \setminus S$.



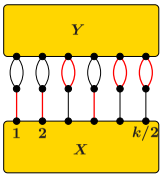
If $\delta(R)$ is not important, then there is an important separator $\delta(R')$ that dominates it. Replace S with $S' := (S \setminus \delta(R)) \cup \delta(R')$ $(|S'| \leq |S|)$.

A u-v path in $G \setminus S'$ implies a u-t path, a contradiction.

Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Example:



There are exactly $2^{k/2}$ important (X, Y)-separators of size at most k in this graph.

Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.

Proof: First we show that $R_{\max} \subseteq R$ for every important separator $\delta(R)$.

$$\begin{array}{ccccccc} |\delta(R_{\max})| & + & |\delta(R)| & \geq & |\delta(R_{\max} \cap R)| & + & |\delta(R_{\max} \cup R)| \\ \lambda & & & & \geq \lambda & & \end{array}$$

$$\Downarrow$$

$$|\delta(R_{\max} \cup R)| \leq |\delta(R)|$$

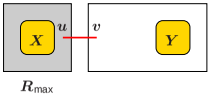
$$\Downarrow$$

If $R \neq R_{\max} \cup R$, then $\delta(R)$ is not important.

Thus the important (X, Y)- and $(R_{\max}, Y)$-separators are the same.

Important separators

Lemma: There are at most 4^k important (X, Y)-separators of size at most k.



The edge uv leaving $R_{\max}$ is either in the separator or not.

Branch 1: Edge uv is in the separator. Delete uv and set $k := k - 1$.
$\Rightarrow$ k decreases by one, λ decreases by at most 1.
Branch 2: Edge uv is not in the separator. Set $X := R_{\max} \cup \{v\}$.
$\Rightarrow$ k remains the same, λ increases by 1.

The measure $2k - \lambda$ decreases in each step.
$\Rightarrow$ Height of the search tree $\leq 2k \Rightarrow \leq 2^{2k}$ important separators.

Algorithm for MULTIWAY CUT

1. If every vertex of T is in a different component, then we are done.
2. Let $t \in T$ be a vertex with that is not separated from every $T \setminus \{t\}$.
3. Branch on a choice of an important $(\{t\}, T \setminus \{t\})$ separator S of size at most k.
4. Set $G := G \setminus S$ and $k := k - |S|$.
5. Go to step 1.

Size of the search tree:

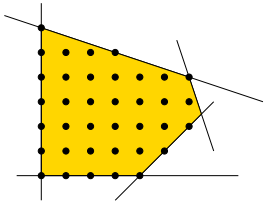
- When searching for the important separator, $2k - \lambda$ decreases at each branching.
- When choosing the next t, λ changes from 0 to positive, thus $2k - \lambda$ does not increase.

Size of the search tree is at most 2^{2k}.

Other separation problems

- Some other variants:
  - $|T|$ as a parameter
  - MULTITERMINAL CUT: pairs $(s_1, t_1), \ldots, (s_\ell, t_\ell)$ have to be separated.
  - Directed graphs
  - Planar graphs
- Useful for deletion-type problems such as DIRECTED FEEDBACK VERTEX SET (via iterative compression).
- Important separators: is it relevant for a given problem?

Integer Linear Programming



Integer Linear Programming

Linear Programming (LP): important tool in (continuous) combinatorial optimization. Sometimes very useful for discrete problems as well.

$$\begin{aligned} &\max c_1x_1 + c_2x_2 + c_3x_3 \\ &\text{s.t.} \\ x_1 + 5x_2 - x_3 &\leq 8 \\ 2x_1 - x_3 &\leq 0 \\ 3x_2 + 10x_3 &\leq 10 \\ x_1, x_2, x_3 &\in \mathbb{R} \end{aligned}$$

Fact: It can be decided if there is a solution (feasibility) and an optimum solution can be found in polynomial time.

Integer Linear Programming

Integer Linear Programming (ILP): Same as LP, but we require that every x_i is integer.

Very powerful, able to model many NP-hard problems. (Of course, no polynomial-time algorithm is known.)

Theorem: ILP with p variables can be solved in time $p^{O(p)} \cdot n^{O(1)}$.

Closest String

Task: Given strings $s_1, \ldots, s_k$ of length L over alphabet Σ, and an integer d, find a string s (of length L) such that $d(s, s_i) \leq d$ for every $1 \leq i \leq k$.

Note: $d(s, s_i)$ is the Hamming distance.

Theorem: Closest String parameterized by k is FPT.
Theorem: Closest String parameterized by d is FPT.
Theorem: Closest String parameterized by L is FPT.
Theorem: Closest String is NP-hard for $\Sigma = \{0, 1\}$.

Closest String

An instance with $k = 5$ and a solution for $d = 4$:

s_1	CBDCCACBB
s_2	ABDBCABDB
s_3	CDDBACCBD
s_4	DDABACCBD
s_5	ACDBDDCBC
	ADDBCACBD

Each column can be described by a partition $\mathcal{P}$ of $[k]$.
The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

CLOSEST STRING

Each column can be described by a partition $\mathcal{P}$ of $[k]$.
The instance can be described by an integer $c_{\mathcal{P}}$ for each partition $\mathcal{P}$: the number of columns with this type.

Describing a solution: If C is a class of $\mathcal{P}$, let $x_{\mathcal{P},C}$ be the number of type $\mathcal{P}$ columns where the solution agrees with class C.

There is a solution iff the following ILP has a feasible solution:

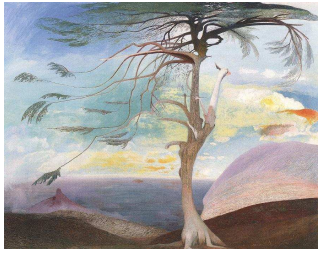
$$\sum_{C \in \mathcal{P}} x_{\mathcal{P},C} \leq c_{\mathcal{P}} \qquad \forall \text{partition } \mathcal{P}$$

$$\sum_{i \notin C, C \in \mathcal{P}} x_{\mathcal{P},C} \leq d \qquad \forall 1 \leq i \leq k$$

$$x_{\mathcal{P},C} \geq 0 \qquad \forall \mathcal{P}, C$$

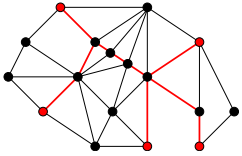
Number of variables is $\leq B(k) \cdot k$, where $B(k)$ is the no. of partitions of $[k]$
$\Rightarrow$ The ILP algorithm solves the problem in time $f(k) \cdot n^{O(1)}$.

STEINER TREE



STEINER TREE

Task: Given a graph G with weighted edges and a set S of k vertices, find a tree T of minimum weight that contains S.



Known to be NP-hard. For fixed k, we can solve it in polynomial time: we can guess the Steiner points and the way they are connected.

Theorem: STEINER TREE is FPT parameterized by $k = |S|$.

STEINER TREE

Solution by dynamic programming. For $v \in V(G)$ and $X \subseteq S$,

$c(v, X) :=$ minimum cost of a Steiner tree of X that contains v

$d(u, v) :=$ distance of u and v

Recurrence relation:

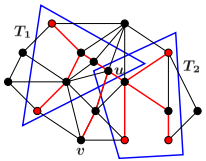
$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

STEINER TREE

Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

- $\leq$: A tree T_1 realizing $c(u, X' \setminus u)$, a tree T_2 realizing $c(u, (X \setminus X') \setminus u)$, and the path uv gives a (superset of a) Steiner tree of X containing v.

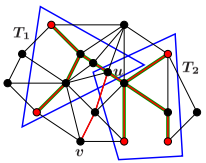


STEINER TREE

Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus X') \setminus u) + d(u, v)$$

- $\geq$: Suppose T realizes $c(v, X)$, let T' be the minimum subtree containing X. Let u be a vertex of T' closest to v. If $|X| > 1$, then there is a component C of $T \setminus u$ that contains a subset $\emptyset \subset X' \subset X$ of terminals. Thus T is the disjoint union of a tree containing $X' \setminus u$ and u, a tree containing $(X \setminus X') \setminus u$ and u, and the path uv.



STEINER TREE

Recurrence relation:

$$c(v, X) = \min_{\substack{u \in V(G) \\ \emptyset \subset X' \subset X}} c(u, X' \setminus u) + c(u, (X \setminus u) \setminus X') + d(u, v)$$

Running time:

$2^k|V(G)|$ variables $c(v, X)$, determine them in increasing order of $|X|$. Variable $c(v, X)$ can be determined by considering $2^{|X|}$ cases. Total number of cases to consider:

$$\sum_{X \subseteq T} 2^{|X|} = \sum_{i=1}^{k} \binom{k}{i} 2^i \leq (1+2)^k = 3^k.$$

Running time is $O^*(3^k)$.

Note: Running time can be reduced to $O^*(2^k)$ with clever techniques.

Conclusions

- Many nice techniques invented so far — and probably many more to come.
- A single technique might provide the key for several problems.
- How to find new techniques? By attacking the open problems!
- Needed: flexible, highly expressive problems. Solve other problems by reduction to these problems.
  - Courcelle's Theorem
  - The matroid result
  - 2SAT DELETION: given a 2SAT formula and an integer k, delete k clauses to make it satisfiable
  - Constraint Satisfaction Problems

Saket Saurabh

Kernel : Lower and Upper Bounds

Kernel: Lower and Upper Bounds

Daniel Lokshtanov* Saket Saurabh*

May 21, 2009

Abstract

Through this lecture note we try to provide a portal into the emerging filed of *kernelization*. We exhibit through examples various tools to prove both lower and upper bounds on the kernel sizes.

1 Introduction

Preprocessing (data reduction or kernelization) as a strategy of coping with hard problems is universally used in almost every implementation. The history of preprocessing, like applying reduction rules to simplify truth functions, can be traced back to the 1950's [16]. A natural question in this regard is how to measure the quality of preprocessing rules proposed for a specific problem. For a long time the mathematical analysis of polynomial time preprocessing algorithms was neglected. The basic reason for this anomaly was that if we start with an instance I of an NP-hard problem and can show that in polynomial time we can replace this with an equivalent instance I' with $|I'| < |I|$ then that would imply P=NP in classical complexity. The situation changed drastically with advent of parameterized complexity. Combining tools from parameterized complexity and classical complexity it has become possible to derive upper and lower bounds on the sizes of reduced instances, or so called *kernels*. Importance of preprocessing and the mathematical challenges it poses is beautifully expressed in the following quote by Fellows.

> *It has become clear, however, that far from being trivial and uninteresting, that pre-processing has unexpected practical power for real world input distributions, and is mathematically a much deeper subject than has generally been understood.*

Working View Point on Kernel: In parameterized complexity each problem instance comes with a parameter k and the parameterized problem is said to admit a *polynomial kernel* if there is a polynomial time algorithm (the degree of polynomial is independent of k), called a *kernelization* algorithm, that reduces the input instance down to an instance with size bounded by a polynomial $p(k)$ in k, while preserving the answer. This reduced instance is called a $p(k)$ *kernel* for the problem. If $p(k) = O(k)$, then we call it a *linear kernel*. Kernelization has been extensively studied in the realm of parameterized complexity,

*Department of Infomatics, University of Bergen, Norway. {daniello|saket.saurabh}@ii.uib.no.

resulting in polynomial kernels for a variety of problems. Notable examples include a $2k$-sized vertex kernel for Vertex Cover [6], a $355k$ kernel for Dominating Set on planar graphs [1], which later was improved to a $67k$ kernel [5], and an $O(k^2)$ kernel for Feedback Vertex Set [18] parameterized by the solution size.

2 Basic Definitions

A parameterized problem L is a subset of $\Sigma^* \times \mathbb{N}$ for some finite alphabet Σ. An instance of a parameterized problem consists of (x, k), where k is called the parameter. A central notion in parameterized complexity is *fixed parameter tractability (FPT)* which means for a given instance (x, k) solvability in time $f(k) \cdot p(|x|)$, where f is an arbitrary function of k and p is a polynomial in the input size. The notions of *kernelization* and *composition* are formally defined as follows.

Definition 1. *A* kernelization algorithm*, or in short, a kernel for a parameterized problem $Q \subseteq \Sigma^* \times \mathbb{N}$ is an algorithm that, given $(x, k) \in \Sigma^* \times \mathbb{N}$, outputs in time polynomial in $|x| + k$ a pair $(x', k') \in \Sigma^* \times \mathbb{N}$ such that (a) $(x, k) \in Q$ if and only if $(x', k') \in Q$ and (b) $|x'| + k' \leq g(k)$, where g is an arbitrary computable function. The function g is referred to as the size of the kernel. If g is a polynomial function then we say that Q admits a polynomial kernel.*

We close with some definitions from graph theory. Let $G = (V, E)$ be a graph. For a vertex v in G, we write $N_G(v)$ to denote the set of v's neighbors in G, and we write $\deg_G(v)$ to denote the *degree* of v, that is, the number of v's neighbors in G. If it is clear from the context which graph is meant, we write $N(v)$ and $\deg(v)$, respectively, for short. A graph $G' = (V', E')$ is a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. The subgraph G' is called an *induced subgraph* of G if $E' = \{\{u, v\} \in E \mid u, v \in V'\}$, in this case, G' is also called the subgraph *induced by* V' and denoted with $G[V']$. A vertex v *dominates* a vertex u if $u \in N(v)$.

3 Upper Bound Machinery

We illustrate the method of kernelization using the parameterized version of Max3Sat where given a boolean 3-CNF formula and an integer parameter k, we would like to know whether there is an assignment to the variables that satisfies at least k of the clauses. Our other examples in this Section include a kernel for d-Hitting Set using the *Sunflower Lemma*, a $4k$ sized kernel for Vertex Cover using *crown decomposition* and a 2^k kernel for (Edge) Clique Cover.

3.1 Max3Sat

Let F be a given boolean CNF 3-SAT formula with n variables and m clauses. t is well known that in any boolean CNF formula, there is an assignment that satisfies at least half of the clauses (given any assignment that doesn't satisfy half the clauses, its bitwise complement will). So if the parameter k is less than $m/2$, then there is an assignment to the variables that satisfies at least k of the clauses. Otherwise, $m \leq 2k$, and so $n \leq 6k$.

3.2 d-Hitting Set

In this Section we give a kernelization algorithm for the d-HITTING SET problem which is defined as follows:

> d-HITTING SET (d-HS) : Given a collection $\mathcal{C}$ of d element subsets of an universe U and a positive integer k, the problem is to determine whether there exists a subset $U' \subseteq U$ of size at most k such that U' contains at least one element from each set in $\mathcal{C}$.

Our kernelization algorithm is based on the following widely used *Sunflower Lemma*. We first define the terminology used in the statement of the lemma. A *sunflower* with k *petals* and a *core* Y is a collection of sets $S_1, S_2 \cdots S_k$ such that $S_i \cap S_j = Y$ for all $i \neq j$; the sets $S_i - Y$ are petals and we require that none of them be empty. Note that a family of pairwise disjoint sets is a sunflower (with an empty core).

Lemma 1 ([10])**. [Sunflower Lemma]** *Let $\mathscr{F}$ be a family of sets over an universe $\mathscr{U}$ each of cardinality s. If $|\mathscr{F}| > s!(k-1)^s$ then $\mathscr{F}$ contains a sunflower with k petals and such a sunflower can be computed in time polynomial in the size of $\mathscr{F}$ and $\mathscr{U}$.*

Now we are ready to prove the following theorem about kernelization for d-HS.

Theorem 1. *d-HS has a kernel of size $O(k^d d! d^2)$. That is, given an instance $(U, \mathcal{C}, k)$ of d-HS, we can replace it with an equivalent instance $(U, \mathcal{C}', k')$ with $|\mathcal{C}'| \leq O(k^d d! d)$ in polynomial time.*

Proof. The crucial observation is that if $\mathcal{C}$ contains a sunflower $S = \{S_1, \cdots, S_{k+1}\}$ of cardinality $k+1$ then every hitting set of $\mathcal{C}$ of size at most k must intersect with the core Y of the sunflower S, otherwise we will need hitting set of size more than k. Therefore if we let $\mathcal{C}' = (\mathcal{C} \cup Y) \setminus S$ then the instance $(U, \mathcal{C}, k)$ and $(U, \mathcal{C}', k)$ are equivalent.

Now we apply the Sunflower Lemma for all $d' \in \{1, \cdots, d\}$, repeatedly replacing sunflowers of size at least $k+1$ with their cores until the number of sets for any fixed $d' \in \{1, \cdots, d\}$ is at most $O(k^{d'} d'!)$. Summing over all d we obtain the desired kernel of size $O(k^d d! d)$. □

3.3 Crown Decomposition : Vertex Cover

In this Section we introduce a *crown decomposition* based kernelization for VERTEX COVER. It is based on a connection between matchings and vertex cover which is that the maximum size of a matching is a lower bound for the minimum cardinality vertex cover. We first define VERTEX COVER precisely as follows.

> VERTEX COVER (VC): Given a graph $G = (V, E)$ and a positive integer k, does there exist a subset $V' \subseteq V$ of size at most k such that for every edge $(u, v) \in E$ either $u \in V'$ or $v \in V'$.

VERTEX COVER can be modelled as 2-HS with universe $U = V$ and $\mathcal{C} = \{\{u, v\} \mid (uv) \in E\}$ and hence using Theorem 1 we get a kernel with at most $4k^2$ edges and $8k^2$ vertices. Here we give a kernel with at most $4k$ vertices.

Now we define crown decomposition.

Definition 2. *A crown decomposition of a graph $G = (V, E)$ is a partitioning of V as C, H and R, where C and H are nonempty and the partition satisfies the following properties.*

1. *C is an independent set.*
2. *There are no edges between vertices of C and R, that is $N[C] \cap R = \emptyset$.*
3. *Let E' be the set of edges between vertices of C and H. Then E' contains a matching of size $|H|$, that is the bipartite subgraph $G' = (C \cup H, E')$ has a matching saturating all the vertices of H.*

We need the following lemma by Chor et. al. [7] which makes it possible to find a crown decomposition efficiently.

Lemma 2. *If a graph $G = (V, E)$ has an independent set $I \subseteq V$ such that $|N(I)| < |I|$, then a crown decomposition (C, H, R) of G such that $C \subseteq I$ can be found in time $O(m+n)$, given G and I.*

The crown-decomposition gives us a global method to reduce the instance size. Its importance is evident from the following simple lemma.

Lemma 3. *Let (C, H, R) be a crown decomposition of a graph $G = (V, E)$. Then G has a vertex cover of size k if and only if $G' = G[R]$ has a vertex cover of size $k' = k - |H|$.*

Proof. Suppose G has a vertex cover V' of size k in G. Now, we have a matching of size $|H|$ between C and H that saturates every vertex of H. Thus $|V' \cap (H \cup C)| \geq |H|$, as any vertex cover must pick one vertex form each of the matching edge. Hence the number of vertices in V' covering the edges not incident to $H \cup C$ is at most $k - |H|$, proving one direction of the result.

For the other direction, it is enough to observe that if V'' is a vertex cover of size $k - |H|$ for G' then $V'' \cup H$ is a vertex cover of size k for G. □

Theorem 2. *Vertex Cover has a kernel of size $4k$.*

Proof. Given an input graph $G = (V, E)$ and a positive integer k, we do as follows. We first find a maximal matching M of G. Let $V(M)$ be the set of endpoints of edges in M. Now if $|V(M)| > 2k$, we answer NO and stop as any vertex cover must contain at least one vertex from each of the matching edges and hence has size more than k. Now we distinguish two cases based on the size of $|V - V(M)|$. If $|V - V(M)| \leq 2k$, then we stop as we have obtained a kernel of size at most $4k$. Else $|V - V(M)| > 2k$. In this case we have found an independent set $I = V - V(M)$ such that $|N(I)| \leq |V(M)| < |I|$ and hence we can apply Lemma 2 to obtain a crown decomposition (C, H, R) of G. Given a crown decomposition (C, H, R), we apply Lemma 3 and obtain a smaller instance for a vertex cover with $G' = G[R]$ and parameter $k' = k - |H|$. Now we repeat the above procedure with this reduced instance until either we get a NO answer or we have $|V - V(M)| \leq 2k$ resulting in a kernel of size $4k$. □

The bound obtained on the kernel for Vertex Cover in Theorem 2 can be further improved to $2k$ with much more sophisticated use of crown decomposition. An alternate method to obtain a $2k$ size kernel for Vertex Cover is through a Linear Programming formulation of Vertex Cover. See [10] and [15] for further details on Linear Programming based kernelization of Vertex Cover.

3.4 Clique Cover

Unfortunately, not all known problem kernels are shown to have polynomial size. Here, we present some data reduction results with exponential-size kernels. Clearly, it is a pressing challenge to find out whether these bounds can be improved to polynomial ones.

In this section, we study the (Edge) Clique Cover problem, where the input consists of an undirected graph $G = (V, E)$ and a nonnegative integer k and the question is whether there is a set of at most k cliques in G such that each edge in E has both its endpoints in at least one of the selected cliques.

Given an n-vertex and m-edge graph G, we use $N(v)$ to denote the neighborhood of vertex v in G, namely, $N(v) := \{u \mid \{u, v\} \in E\}$. The *closed* neighborhood of vertex v, denoted by $N[v]$, is equal to $N(v) \cup \{v\}$.

We formulate data reduction rules for a generalized version of (Edge) Clique Cover in which already some edges may be marked as "covered". Then, the question is to find a clique cover of size k that covers all uncovered edges. We apply the following data reduction rules [14]:

1. Remove isolated vertices and vertices that are only adjacent to covered edges.
2. If an uncovered edge $\{u, v\}$ is contained in exactly one maximal clique C, that is, if the common neighbors of u and v induce a clique, then add C to the solution, mark its edges as covered, and decrease k by one.
3. If there is an edge $\{u, v\}$ whose endpoints have exactly the same closed neighborhood, that is, $N[u] = N[v]$, then mark all edges incident to u as covered. To reconstruct a solution for the non-reduced instance, add u to every clique containing v.

The correctness of the rules is easy to prove. To show the following problem kernel, only the first and third rule are needed.

Theorem 3 ([14])**.** (Edge) Clique Cover *admits a problem kernel with at most* 2^k *vertices.*

Proof. Consider any graph $G = (V, E)$ with more than 2^k vertices that has a clique cover $C_1, \ldots, C_k$ of size k. We assign to each vertex $v \in V$ a binary vector b_v of length k where bit i, $1 \leq i \leq k$, is set to 1 iff v is contained in clique C_i. Since there are only 2^k possible vectors, there must be $u \neq v \in V$ with $b_u = b_v$. If b_u and b_v are zero, the first rule applies; otherwise, u and v are contained in the same cliques. This means that u and v are connected and share the same neighborhood, and thus the third rule applies. □

4 Lower Bound Machinery

It is easy to see that if a decidable problem admits an $f(k)$ kernel for some function f, then the problem is FPT. Interestingly, the converse also holds, that is, if a problem is FPT then it admits an $f(k)$ kernel for some function f [15]. The proof of this fact is quite simple, and we present it here.

Fact 1 (Folklore, [15])**.** *If a parameterized problem* Π *is* FPT *then* Π *admits an* $f(k)$ *kernel for some function* f*.*

Proof. Suppose there is a decision algorithm for Π running in $f(k)n^c$ time for some function f and constant c. Given an instance (I,k) with $|I|=n$, if $n \geq f(k)$ then we run the decision algorithm on the instance in time $f(k)n^c \leq n^{c+1}$. If the decision algorithm outputs *yes*, the kernelization algorithm outputs a constant size *yes* instance, and if the decision algorithm outputs *no*, the kernelization algorithm outputs a constant size *no* instance. On the other hand, if $n < f(k)$ the kernelization algorithm just outputs (I,k). This yields an $f(k)$ kernel for the problem. □

Fact 1 implies that a problem has a kernel if and only if it is fixed parameter tractable. However, we are interested in kernels that are as small as possible, and a kernel obtained using Fact 1 has size that equals the dependence on k in the running time of the best known FPT algorithm for the problem. The question is - can we do better? The answer is that quite often we can as we saw in the previous section but many times we can not. It is only very recently that a methodology to rule out polynomial kernels has been developed [3, 12]. In this chapter we survey the tecniques that have been developed to show kernelization lower bounds. In this section we suvey some of the recently developed techniques for showing that problems do not admit polynomial kernels.

Consider the Longest Path problem. It is well known that the Longest Path problem can be solved in time $O(c^k n^{O(1)})$ using the well known method of Color-Coding. Is it feasible that it also admits a polynomial kernel? We argue that intuitively this should not be possible. Consider a large set $(G_1,k),(G_2,k),\ldots,(G_t,k)$ of instances to the Longest Path problem. If we make a new graph G by just taking the disjoint union of the graphs $G_1 \ldots G_t$ we see that G contains a path of length k if and only if G_i contains a path of length k for some $i \leq t$. Suppose the Longest Path problem had a polynomial kernel, and we ran the kernelization algorithm on G. Then this algorithm would in polynomial time return a new instance (G',k') such that $|V(G)| = k^{O(1)}$, a number potentially much smaller than t. This means that in some sense, the kernelization algorithm considers the instances $(G_1,k),(G_2,k)\ldots(G_t,k)$ and in *polynomial time* figures out which of the instances are the most likely to contain a path of length k. However, at least intuitively, this seems almost as difficult as solving the instances themselves and since the Longest Path problem is NP-complete, this seems unlikely. We now formalize this intuition.

Definition 3. [Distillation [3]]

- *An OR-distillation algorithm for a language $L \subseteq \Sigma^*$ is an algorithm that receives as input a sequence $x_1,\ldots,x_t$, with $x_i \in \Sigma^*$ for each $1 \leq i \leq t$, uses time polynomial in $\sum_{i=1}^{t}|x_i|$, and outputs $y \in \Sigma^*$ with (a) $y \in L \iff x_i \in L$ for* some *$1 \leq i \leq t$ and (b) $|y|$ is polynomial in $\max_{i \leq t}|x_i|$. A language L is OR-distillable if there is a OR-distillation algorithm for it.*

- *An AND-distillation algorithm for a language $L \subseteq \Sigma^*$ is an algorithm that receives as input a sequence $x_1,\ldots,x_t$, with $x_i \in \Sigma^*$ for each $1 \leq i \leq t$, uses time polynomial in $\sum_{i=1}^{t}|x_i|$, and outputs $y \in \Sigma^*$ with (a) $y \in L \iff x_i \in L$ for* all *$1 \leq i \leq t$ and (b) $|y|$ is polynomial in $\max_{i \leq t}|x_i|$. A language L is AND-distillable if there is an AND-distillation algorithm for it.*

Observe that the notion of distillation is defined for unparameterized problems. Bodlaender et al. [3] conjectured that no NP-complete language can have an OR-distillation or an AND-distillation algorithm.

Conjecture 1 (OR-Distillation Conjecture [3]). *No* NP*-complete language L is OR-distillable.*

Conjecture 2 (AND-Distillation Conjecture [3]). *No* NP*-complete language L is AND-distillable.*

One should notice that if any NP-complete language is distillable, then so are all of them. Fortnow and Santhanam [12] were able to connect the OR-Distillation Conjecture to a well-known conjecture in classical complexity. In particular they proved that if the OR-Distillation Conjecture fails, the *polynomial time hierarchy* [17] collapses to the third level, a collapse that is deemed unlikely. No such connection is currently known for the AND-Distillation Conjecture, and for reasons soon to become apparent, a proof of such a connection would have significant impact in Parameterized Complexity. By PH=Σ_p^3 we will denote the complexity-theoretic event that the polynomial time hierarchy collapses to the third level.

Theorem 4 ([12]). *If the OR-Distillation Conjecture fails, then* PH=Σ_p^3.

We are now ready to define the parameterized analogue of distillation algorithms and connect this notion to the Conjectures 1 and 2

Definition 4. [Composition [3]]

- *A composition algorithm (also called OR-composition algorithm) for a parameterized problem $\Pi \subseteq \Sigma^* \times \mathbb{N}$ is an algorithm that receives as input a sequence $((x_1, k), \ldots, (x_t, k))$, with $(x_i, k) \in \Sigma^* \times \mathbb{N}^+$ for each $1 \leq i \leq t$, uses time polynomial in $\sum_{i=1}^{t} |x_i| + k$, and outputs $(y, k') \in \Sigma^* \times \mathbb{N}^+$ with (a) $(y, k') \in \Pi \iff (x_i, k) \in \Pi$ for* some *$1 \leq i \leq t$ and (b) k' is polynomial in k. A parameterized problem is compositional (or* OR-compositional*) if there is a composition algorithm for it.*

- *An AND-composition algorithm for a parameterized problem $\Pi \subseteq \Sigma^* \times \mathbb{N}$ is an algorithm that receives as input a sequence $((x_1, k), \ldots, (x_t, k))$, with $(x_i, k) \in \Sigma^* \times \mathbb{N}^+$ for each $1 \leq i \leq t$, uses time polynomial in $\sum_{i=1}^{t} |x_i| + k$, and outputs $(y, k') \in \Sigma^* \times \mathbb{N}^+$ with (a) $(y, k') \in \Pi \iff (x_i, k) \in \Pi$ for* all *$1 \leq i \leq t$ and (b) k' is polynomial in k. A parameterized problem is AND-compositional if there is an AND-composition algorithm for it.*

Composition and distillation algorithms are very similar. The main difference between the two notions is that the restriction on output size for distillation algorithms is replaced by a restriction on the parameter size for the instance the composition algorithm outputs. We define the notion of the *unparameterized version* of a parameterized problem L. The mapping of parameterized problems to unparameterized problems is done by mapping (x, k) to the string $x\#1^k$, where $\# \notin \Sigma$ denotes the blank letter and 1 is an arbitrary letter in Σ. In this way, the unparameterized version of a parameterized problem Π is the language $\tilde{\Pi} = \{x\#1^k \mid (x, k) \in \Pi\}$. The following theorem yields the desired connection between the two notions.

Theorem 5 ([3, 12]). *Let Π be a compositional parameterized problem whose unparameterized version $\widetilde{\Pi}$ is* NP*-complete. Then, if Π has a polynomial kernel then* PH=Σ_p^3. *Similarly, let Π be an AND-compositional parameterized problem whose unparameterized version $\widetilde{\Pi}$ is* NP*-complete. Then, if Π has a polynomial kernel the AND-Distillation Conjecture fails.*

We can now formalize the discussion from the beginning of this section.

Theorem 6 ([3]). Longest Path *does not admit a polynomial kernel unless* PH=Σ_p^3.

Proof. The unparameterized version of Longest Path is known to be NP-complete [13]. We now give a composition algorithm for the problem. Given a sequence $(G_1, k) \dots (G_t, k)$ of instances we output (G, k) where G is the disjoint union of $G_1 \dots G_t$. Clearly G contains a path of length k if and only if G_i contains a path of length k for some $i \leq t$. By Theorem 5 Longest Path does not have a polynomial kernel unless PH=Σ_p^3. □

An identical proof can be used to show that the Longest Cycle problem does not admit a polynomial kernel unless PH=Σ_p^3. For many problems, it is easy to give AND-composition algorithms. For instance, the "disjoint union" trick yields AND-composition algorithms for the Treewidth, Cutwidth and Pathwidth problems, among many others. Coupled with Theorem 5 this implies that these problems do not admit polynomial kernels unless the AND-Distillation Conjecture fails. However, to this date, there is no strong complexity theoretic evidence known to support the AND-Distillation Conjecture. Therefore it would be interesting to see if such evidence could be provided.

For some problems, obtaining a composition algorithm directly is a difficult task. Instead, we can give a reduction from a problem that provably has no polynomial kernel unless PH=Σ_p^3 to the problem in question such that a polynomial kernel for the problem considered would give a kernel for the problem we reduced from. We now define the notion of *polynomial parameter transformations*.

Definition 5 ([4]). *Let P and Q be parameterized problems. We say that P is polynomial parameter reducible to Q, written $P \leq_{Ptp} Q$, if there exists a polynomial time computable function $f : \Sigma^* \times \mathbb{N} \to \Sigma^* \times \mathbb{N}$ and a polynomial p, such that for all $(x, k) \in \Sigma^* \times \mathbb{N}$ (a) $(x, k) \in P$ if and only $(x', k') = f(x, k) \in Q$ and (b) $k' \leq p(k)$. The function f is called* polynomial parameter transformation.

Proposition 1 ([4]). *Let P and Q be the parameterized problems and $\tilde{P}$ and $\tilde{Q}$ be the unparameterized versions of P and Q respectively. Suppose that $\tilde{P}$ is* NP*-complete and $\tilde{Q}$ is in* NP*. Furthermore if there is a polynomial parameter transformation from P to Q, then if Q has a polynomial kernel then P also has a polynomial kernel.*

Proposition 1 shows how to use polynomial parameter transformations to show kernelization lower bounds. A notion similar to polynomial parameter transformation was independently used by Fernau et al. [9] albeit without being explicitly defined. We now give an example of how Proposition 1 can be useful for showing that a problem does not admit a polynomial kernel. In particular, we show that the Path Packing problem does not admit a polynomial kernel unless PH=Σ_p^3. In this problem you are given a graph G together with an integer k and asked whether there exists a collection of k mutually vertex-disjoint paths of length k in G. This problem is known to be fixed parameter tractable [2] and is easy to

see that for this problem the "disjoint union" trick discussed earlier does not directly apply. Thus we resort to polynomial parameter transformations.

Theorem 7. PATH PACKING *does not admit a polynomial kernel unless* PH=Σ_p^3.

Proof. We give a polynomial parameter transformation from the LONGEST PATH problem. Given an instance (G, k) to LONGEST PATH we construct a graph G' from G by adding $k-1$ vertex disjoint paths of length k. Now G contains a path of length k if and only if G' contains k paths of length k. This concludes the proof. □

Now we give several non-trivial applications of this framework developed by Bodlaender et al. [3] and Fortnow and Santhanam [12]. In particular we describe a "cookbook" for showing kernelization lower bounds using explicit identification. We then apply this cookbook to show that a wide variety of problems do not admit polynomial kernels. To show that a problem does not admit a polynomial size kernel we go through the following steps.

1. Find a suitable parameterization of the problem considered. Quite often parameterizations that impose extra structure make it easier to give a composition algorithm.
2. Define a suitable colored version of the problem. This is in order to get more control over how solutions to problem instances can look.
3. Show that the colored version of the problem is NP-complete.
4. Give a polynomial parameter transformation from the colored to the uncolored version. This will imply that if the uncolored version has a polynomial kernel then so does the colored version. Hence kernelization lower bounds for the colored version directly transfer to the original problem.
5. Show that the colored version parameterized by k is solvable in time $2^{k^c} \cdot n^{O(1)}$ for a fixed constant c.
6. Finally, show that the colored version is compositional and thus has no polynomial kernel. To do so, proceed as follows.
   (a) If the number of instances in the input to the composition algorithm is at least 2^{k^c} then running the parameterized algorithm on each instance takes time polynomial in input size. This automatically yields a composition algorithm.
   (b) If the number of instances is less than 2^{k^c}, every instance receives an unique identifier. Notice that in order to uniquely code the identifiers (ID) of all instances, k^c bits per instance is sufficient. The IDs are coded either as an integer, or as a subset of a $poly(k)$ sized set.
   (c) Use the coding power provided by colors and IDs to complete the composition algorithm.

In the following sections we show how to apply this approach to show incompressibility and kernelization lower bounds for a variety of problems.

4.1 Steiner Tree, Variants of Vertex Cover, and Bounded Rank Set Cover

The problems Steiner Tree, Connected Vertex Cover (ConVC), Capacitated Vertex Cover (CapVC), and Bounded Rank Set Cover are defined as follows. In Steiner Tree we are given a graph a graph $G = (T \cup N, E)$ and an integer k and asked for a vertex set $N' \subseteq N$ of size at most k such that $G[T \cup N']$ is connected. In ConVC we are given a graph $G = (V, E)$ and an integer k and asked for a vertex cover of size at most k that induces a connected subgraph in G. A *vertex cover* is a set $C \subseteq V$ such that each edge in E has at least one endpoint in C. The problem CapVC takes as input a graph $G = (V, E)$, a capacity function $cap : V \to \mathbb{N}^+$ and an integer k, and the task is to find a vertex cover C and a mapping from E to C in such a way that at most $cap(v)$ edges are mapped to every vertex $v \in C$. Finally, an instance of Bounded Rank Set Cover consists of a set family $\mathcal{F}$ over a universe U where every set $S \in \mathcal{F}$ has size at most d, and a positive integer k. The task is to find a subfamily $\mathcal{F}' \subseteq \mathcal{F}$ of size at most k such that $\cup_{S \in \mathcal{F}'} S = U$. All four problems are known to be NP-complete (e.g., see [13] and the proof of Theorem 8); in this section, we show that the problems do not admit polynomial kernels for the parameter $(|T|, k)$ (in the case of Steiner Tree), k (in the case of ConVC and CapVC), and (d, k) (in the case of Bounded Rank Set Cover), respectively. To this end, we first use the framework presented at the beginning of this chapter to prove that another problem, which is called RBDS, does not have a polynomial kernel. Then, by giving polynomial parameter transformations from RBDS to the above problems, we show the non-existence of polynomial kernels for these problems.

In Red-Blue Dominating Set (RBDS) we are given a bipartite graph $G = (T \cup N, E)$ and an integer k and asked whether there exists a vertex set $N' \subseteq N$ of size at most k such that every vertex in T has at least one neighbor in N'. We show that RBDS parameterized by $(|T|, k)$ does not have a polynomial kernel. In the literature, the sets T and N are called "blue vertices" and "red vertices", respectively. In this paper we will call the vertices "terminals" and "nonterminals" in order to avoid confusion with the colored version of the problem that we are going to introduce. RBDS is equivalent to Set Cover and Hitting Set and is, therefore, NP-complete [13].

In the colored version of RBDS, denoted by Colored Red-Blue Dominating Set (Col-RBDS), the vertices of N are colored with colors chosen from $\{1, \ldots, k\}$, that is, we are additionally given a function $col : N \to \{1, \ldots, k\}$, and N' is required to contain exactly one vertex of each color. We will now follow the framework described at the beginning of this chapter.

Lemma 4. (1) *The unparameterized version of* Col-RBDS *is* NP*-complete.* (2) *There is a polynomial parameter transformation from* Col-RBDS *to* RBDS. (3) Col-RBDS *is solvable in* $2^{|T|+k} \cdot |T \cup N|^{O(1)}$ *time.*

Proof. (1) It is easy to see that Col-RBDS is in NP. To prove its NP-hardness, we reduce the NP-complete problem RBDS to Col-RBDS: Given an instance $(G = (T \cup N, E), k)$ of RBDS, we construct an instance $(G' = (T \cup N', E'), k, col)$ of Col-RBDS where the vertex set N' consists of k copies $v^1, \ldots, v^k$ of every vertex $v \in V$, one copy of each color. That is, $N' = \bigcup_{a \in \{1,\ldots,k\}} \{v^a \mid v \in N\}$, and the color of every vertex $v^a \in N_a$ is $col(v^a) = a$. The edge

set E' is given by

$$E' = \bigcup_{a \in \{1,\ldots,k\}} \{\{u, v^a\} \mid u \in T \wedge a \in \{1,\ldots,k\} \wedge \{u,v\} \in E\}.$$

Now, there is a set $S \subset N$ of size k dominating all vertices in T in G if and only if in G', there is a set $S' \subset N'$ of size k containing one vertex of each color.

(2) Given an instance $(G = (T \cup N, E), k, col)$ of Col-RBDS, we construct an instance $(G' = (T' \cup N, E'), k)$ of RBDS. The set T' consists of all vertices from T plus k additional vertices $z_1, \ldots, z_k$. The edge set E' consists of all edges from E plus the edges

$$\{\{z_a, v\} \mid a \in \{1,\ldots,k\} \ \wedge \ v \in N \ \wedge \ col(v) = a\}.$$

Now, there is a set $S' \subset N'$ of size k dominating all vertices in T' in G' if and only if in G, there is a set $S \subset N$ of size k containing one vertex of each color.

(3) To solve Col-RBDS in the claimed running time, we first use the reduction given in (2) from Col-RBDS to RBDS. The number $|T'|$ of terminals in the constructed instance of RBDS is $|T|+k$. Next, we transform the RBDS instance (G', k) into an instance $(\mathcal{F}, U, k)$ of Set Cover where the elements in U one-to-one correspond to the vertices in T' and the sets in $\mathcal{F}$ one-to-one correspond to the vertices in N. Since Set Cover can be solved in $O(2^{|U|} \cdot |U| \cdot |\mathcal{F}|)$ time [11, Lemma 2], statement (3) follows.

□

Lemma 5. Col-RBDS *parameterized by* $(|T|, k)$ *is compositional.*

Proof. Given a sequence

$$(G_1 = (T_1 \cup N_1, E_1), k, col_1), \ldots, (G_t = (T_t \cup N_t, E_t), k, col_t)$$

of Col-RBDS instances with $|T_1| = |T_2| = \ldots = |T_t| = p$, we show how to construct a Col-RBDS instance $(G = (T \cup N, E), k, col)$ as described in Definition 4.

For $i \in \{1,\ldots,t\}$, let $T_i := \{u_1^i, \ldots, u_p^i\}$ and $N_i := \{v_1^i, \ldots, v_{q_i}^i\}$. We start with adding p vertices $u_1, \ldots, u_p$ to the set T of terminals to be constructed. (We will add more vertices to T later.) Next, we add to the set N of nonterminals all vertices from the vertex sets $N_1, \ldots, N_t$, preserving the colors of the vertices. That is, we set $N = \bigcup_{i \in \{1,\ldots,t\}} N_i$, and for every vertex $v_j^i \in N$ we define $col(v_j^i) = col_i(v_j^i)$. Now, we add the edge set $\bigcup_{i \in \{1,\ldots,t\}} \left\{\{u_{j_1}, v_{j_2}^i\} \mid \{u_{j_1}^i, v_{j_2}^i\} \in E_i\right\}$ to G (see Figure 1). The graph G and the coloring col constructed so far have the following property: If at least one of the Col-RBDS instances $(G_1, k, col_1), \ldots, (G_t, k, col_t)$ is a *yes*-instance, then (G, k, col) is also a *yes*-instance because if for any $i \in \{1,\ldots,t\}$ a size-k subset from N_i dominates all vertices in T_i, then the same vertex set selected from N also dominates all vertices in T. However, (G, k, col) may even be a *yes*-instance in the case where all instances $(G_1, k, col_1), \ldots, (G_t, k, col_t)$ are *no*-instances, because in G one can select vertices into the solution that originate from different instances of the input sequence.

To ensure the correctness of the composition, we add more vertices and edges to G. We define for every graph G_i of the input sequence a unique identifier $\mathrm{ID}(G_i)$, which consists

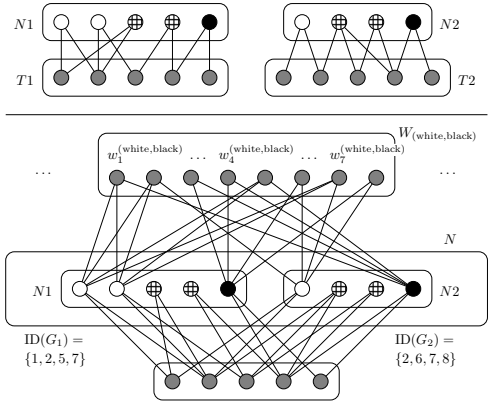


Figure 1: Example for the composition algorithm for Col-RBDS. The upper part of the figure shows an input sequence consisting of two instances with $k = 3$ (there are three colors: white, checkered, and black). The lower part of the figure shows the output of the composition algorithm. For the sake of clarity, only the vertex set $W_{(\text{white,black})}$ is displayed, whereas five other vertex sets $W_{(a,b)}$ with $a, b \in \{\text{white}, \text{checkered}, \text{black}\}$ are omitted. Since $k = 3$ and $p = 5$, each ID should consist of eight numbers, and $W_{(\text{white,black})}$ should contain 16 vertices. For the sake of clarity, the displayed IDs consist of only four numbers each, and $W_{(\text{white,black})}$ contains only eight vertices.

of a size-$(p + k)$ subset of $\{1, \ldots, 2(p + k)\}$ Since $\binom{2(p+k)}{p+k} \geq 2^{p+k}$ and since we can assume that the input sequence does not contain more than 2^{p+k} instances, it is always possible to assign unique identifiers to all instances of the input sequence. (Note that if there are more than 2^{p+k} instances, then we can solve all these instances in $\sum_{i=1}^{t} 2^{p+k} \cdot (p + q_i)^{O(1)} \leq t \cdot \sum_{i=1}^{t} (p + q_i)^{O(1)}$ time, which yields a composition algorithm.) For each color pair $(a, b) \in \{1, \ldots, k\} \times \{1, \ldots, k\}$ with $a \neq b$, we add a vertex set $W_{(a,b)} = \{w_1^{(a,b)}, \ldots, w_{2(p+k)}^{(a,b)}\}$ to T, and we add to E the edge set

$$\bigcup_{i \in \{1,\ldots,t\}, j_1 \in \{1,\ldots,q_i\}} \left\{ \{v_{j_1}^i, w_{j_2}^{(a,b)}\} \mid a = col(v_{j_1}^i) \wedge b \in \{1, \ldots, k\} \setminus \{a\} \wedge j_2 \in \text{ID}(G_i) \right\} \cup$$

$$\bigcup_{i \in \{1,\ldots,t\}, j_1 \in \{1,\ldots,q_i\}} \left\{ \{v_{j_1}^i, w_{j_2}^{(a,b)}\} \mid b = col(v_{j_1}^i) \wedge a \in \{1, \ldots, k\} \setminus \{b\} \wedge j_2 \notin \text{ID}(G_i) \right\}$$

(see Figure 1).

Note that the construction conforms to the definition of a composition algorithm; in particular, k remains unchanged and the size of T is polynomial in p, k because $|T| = p + k(k - 1) \cdot 2(p + k)$. To prove the correctness of the construction, we show that (G, k, col) has a solution $N' \subseteq N$ if and only if at least one instance (G_i, k, col_i) from the input sequence has a solution $N_i' \subseteq N_i$.

In one direction, if $N_i' \subseteq N_i$ is a solution for (G_i, k, col_i), then the same vertex set chosen

from N forms a solution for (G, k, col). To see this, first note that the vertices from T are dominated by the chosen vertices. Moreover, for every color pair $(a, b) \in \{1, \ldots, k\} \times \{1, \ldots, k\}$ with $a \neq b$, each vertex from $W_{(a,b)}$ is either connected to all vertices v from N_i with $col(v) = a$ or to all vertices v from N_i with $col(v) = b$. Since N'_i contains one vertex of each color class from N_i, each vertex in $W_{(a,b)}$ is dominated by a vertex from N chosen into the solution.

In the other direction, to show that any solution $N' \subseteq N$ for (G, k, col) is a solution for at least one instance (G_i, k, col_i), we prove that N' cannot contain vertices originating from different instances of the input sequence. To this end, note that each two vertices in N' must have different colors. Assume, for the sake of a contradiction, that N' contains a vertex $v_{j_1}^{i_1}$ with $col(v_{j_1}^{i_1}) = a$ originating from the instance (G_{i_1}, k, col_{i_1}) and a vertex $v_{j_2}^{i_2}$ with $col(v_{j_2}^{i_2}) = b$ originating from a different instance (G_{i_2}, k, col_{i_2}). Due to the construction of the IDs, we have $\mathrm{ID}(G_{i_1}) \setminus \mathrm{ID}(G_{i_2}) \neq \emptyset$ and $\mathrm{ID}(G_{i_2}) \setminus \mathrm{ID}(G_{i_1}) \neq \emptyset$. This implies that there are vertices in $W_{(a,b)}$ (namely, all vertices $w_j^{(a,b)}$ with $j \in \mathrm{ID}(G_{i_2}) \setminus \mathrm{ID}(G_{i_1})$) and vertices in $W_{(b,a)}$ (namely, all vertices $w_j^{(b,a)}$ with $j \in \mathrm{ID}(G_{i_1}) \setminus \mathrm{ID}(G_{i_2})$) that are neither adjacent to $v_{j_1}^{i_1}$ nor to $v_{j_2}^{i_2}$. Therefore, N' does not dominate all vertices from T, which is a contradiction to the fact that N' is a solution for (G, k, col). □

Theorem 8. ([8]) Red-Blue Dominating Set *and* Steiner Tree*, both parameterized by* $(|T|, k)$*,* Connected Vertex Cover *and* Capacitated Vertex Cover*, both parameterized by* k*, and* Bounded Rank Set Cover*, parameterized by* (k, d)*, do not admit polynomial kernels unless* $PH = \Sigma_p^3$*.*

Proof. For RBDS the statement of the theorem follows directly by Theorem 5 together with Lemmata 4 and 5.

To show that the statement is true for the other four problems, we give polynomial parameter transformations from RBDS to each of them—due to Proposition 1, this suffices to prove the statement. Let $(G = (T \cup N, E), k)$ be an instance of RBDS. To transform it into an instance $(G' = (T' \cup N, E'), k)$ of Steiner Tree, define $T' = T \cup \{\tilde{u}\}$ where $\tilde{u}$ is a new vertex and let $E' = E \cup \{\{\tilde{u}, v_i\} \mid v_i \in N\}$. It is easy to see that every solution for Steiner Tree on (G', k) one-to-one corresponds to a solution for RBDS on (G, k).

To transform (G, k) into an instance $(G'' = (V'', E''), k'')$ of ConVC, first construct the graph $G' = (T' \cup N, E')$ as described above. The graph G'' is then obtained from G' by attaching a leaf to every vertex in T'. Now, G'' has a connected vertex cover of size $k'' = |T'| + k = |T| + 1 + k$ iff G' has a steiner tree containing k vertices from N iff all vertices from T can be dominated in G by k vertices from N.

Next, we describe how to transform (G, k) into an instance $(G''' = (V''', E'''), cap, k''')$ of CapVC. First, for each vertex $u_i \in T$, add a clique to G''' that contains four vertices $u_i^0, u_i^1, u_i^2, u_i^3$. Second, for each vertex $v_i \in N$, add a vertex v_i''' to G'''. Finally, for each edge $\{u_i, v_j\} \in E$ with $u_i \in T$ and $v_j \in N$, add the edge $\{u_i^0, v_j'''\}$ to G'''. The capacities of the vertices are defined as follows: For each vertex $u_i \in T$, the vertices $u_i^1, u_i^2, u_i^3 \in V'''$ have capacity 1 and the vertex $u_i^0 \in V'''$ has capacity $\deg_{G'''}(u_i^0) - 1$. Each vertex v_i''' has capacity $\deg_{G'''}(v_i''')$. Clearly, in order to cover the edges of the size-4 cliques inserted for the vertices of T, every capacitated vertex cover for G''' must contain all vertices $u_i^0, u_i^1, u_i^2, u_i^3$. Moreover, since the capacity of each vertex u_i^0 is too small to cover all edges incident to u_i^0,

at least one neighbor v_j''' of u_i^0 must be selected into every capacitated vertex cover for G'''. Therefore, it is not hard to see that G''' has a capacitated vertex cover of size $k''' = 4 \cdot |T| + k$ iff all vertices from T can be dominated in G by k vertices from N.

Finally, to transform (G, k) into an instance $(\mathcal{F}, U, k)$ of Bounded Rank Set Cover, add one element e_i to U for every vertex $u_i \in T$. For every vertex $v_j \in N$, add one set $\{e_i \mid \{u_i, v_j\} \in E\}$ to $\mathcal{F}$. The correctness of the construction is obvious, and since $|U| = |T|$, every set in $\mathcal{F}$ contains at most $d = |T|$ elements. □

References

[1] Jochen Alber, Michael R. Fellows, and Rolf Niedermeier. Polynomial-time data reduction for dominating set. *J. ACM*, 51(3):363–384, 2004.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Color-coding. *J. Assoc. Comput. Mach.*, 42(4):844–856, 1995.

[3] Hans L. Bodlaender, Rodney G. Downey, Michael R. Fellows, and Danny Hermelin. On problems without polynomial kernels. In *Proc. 35th ICALP*, volume 5125 of *LNCS*, pages 563–574. Springer, 2008.

[4] Hans L. Bodlaender, Stéphan Thomassé, and Anders Yeo. Analysis of data reduction: Transformations give evidence for non-existence of polynomial kernels. Technical Report UU-CS-2008-030, Department of Information and Computing Sciences, Utrecht University, 2008.

[5] Jianer Chen, Henning Fernau, Iyad A. Kanj, and Ge Xia. Parametric duality and kernelization: Lower bounds and upper bounds on kernel size. In *Proc. 22nd STACS*, volume 3404 of *LNCS*, pages 269–280. Springer, 2005.

[6] Jianer Chen, Iyad A. Kanj, and Weijia Jia. Vertex Cover: Further observations and further improvements. *J. Algorithms*, 41(2):280–301, 2001.

[7] Benny Chor, Mike Fellows, and David W. Juedes. Linear kernels in linear time, or how to save k colors in o(n^2) steps. In *Proc. 30th WG*, volume 3353 of *LNCS*, pages 257–269. Springer, 2004.

[8] Michael Dom, Daniel Lokshtanov, and Saket Saurabh. Incompressibility through Colors and IDs. In *Proc. 36th ICALP*, To appear.

[9] Henning Fernau, Fedor V. Fomin, Daniel Lokshtanov, Daniel Raible, Saket Saurabh, and Yngve Villanger. Kernel(s) for problems with no kernel: On out-trees with many leaves. In *Proc. 26th STACS*, pages 421–432.

[10] J. Flum and M. Grohe. *Parameterized Complexity Theory (Texts in Theoretical Computer Science. An EATCS Series)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[11] Fedor V. Fomin, Dieter Kratsch, and Gerhard J. Woeginger. Exact (exponential) algorithms for the dominating set problem. In *Proc. 30th WG*, volume 3353 of *LNCS*, pages 245–256. Springer, 2004.

[12] Lance Fortnow and Rahul Santhanam. Infeasibility of instance compression and succinct PCPs for NP. In *Proc. 40th STOC*, pages 133–142. ACM Press, 2008.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[14] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Data reduction, exact, and heuristic algorithms for Clique Cover. In *Proc. 8th ACM-SIAM ALENEX*, pages 86–94. ACM-SIAM, 2006. Long version to appear in *The ACM Journal of Experimental Algorithmics*.

[15] Rolf Niedermeier. *An Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[16] W. V. Quine. The problem of simplifying truth functions. *Amer. Math. Monthly*, 59:521–531, 1952.

[17] Larry J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[18] Stéphan Thomassé. A quadratic kernel for feedback vertex set. In *Proc. 20th SODA*, pages 115–119. ACM/SIAM, 2009.