- Algebraic circuit over some ring R is a circuit which computes polynomial in $R[x_1, \ldots, x_n]$.
- Algebraic circuit has addition and multiplication gates, starting from the input variables $\vec{x}$ and constants from R.
- The size of an algebraic circuit C is the number of nodes in it.
- Constant-free algebraic circuit is an algebraic circuit in which the only constants used are $0, 1, -1$.

Definition. Let $f \in \mathbb{Z}[\overline{x}]$ be a multivariate polynomial over $\mathbb{Z}$. Then $\tau(f)$ is the minimal size of a constant-free algebraic circuit that computes f (that is, a circuit where the only possible constants that may appear on leaves are $1, 0, -1$).

Definition. A sequence (f_n) of integer polynomials belongs to the complexity class VP^0 iff there exists a sequence (C_n) of division-free and constant-free algebraic circuits such that C_n computes f_n and the size and the formal degree of C_n are polynomially bounded in n.

Definition. A sequence $(f_n(X_1, \ldots, X_{u(n)}))$ of integer polynomials belongs to the complexity class VNP^0 iff there exists a sequence $(g_n(X_1, \ldots, X_{v(n)}))$ in VP^0 such that

$$f_n(X_1, \ldots, X_{u(n)}) = \sum_{e \in \{0,1\}^{v(n)-u(n)}} g_n(X_1, \ldots, X_{u(n)}, e_1, \ldots, e_{v(n)-u(n)}).$$

where $u(n)$ and $v(n)$ are polynomially bounded functions of n.

We recall that *permanent* of the matrix $[X_{ij}]$ is defined as

$$Per_n = \sum_{\pi \in S_n} X_{1\pi(1)} \cdots X_{n\pi(n)}$$

Theorem. If $\tau(n!) \neq (\log n)^{O(1)}$ then $\tau(PER_n) \neq n^{O(1)}$. If $\tau(PER_n) \neq n^{O(1)}$ then $(PER_n) \notin VP^0$. If $(PER_n) \notin VP^0$ then $VP^0 \neq VNP^0$.

Exact Algorithms via Monotone Local Search

Paper.

The main result of the paper is that for a large class of problems, an algorithm with running time $c^k N^{\mathcal{O}(1)}$ for any $c > 1$ immediately implies an exact algorithm with running time $(2 - \frac{1}{c})^{n+o(n)}$.

Subset problem: Let U be a universe on n elements, and let $\mathcal{F}$ be a collection of subsets of U. The task is to determine whether $|\mathcal{F}| > 0$.

Examples:

1. Weighted d-SAT

   Input: A propositional formula F in conjunctive normal form where each clause has at most d literals, a weight function $w : var(F) \to \mathbb{Z}$, and integers k and W.
   Question: Is there a set $S \subseteq var(F)$ of size at most k and weight at most W such that F is satisfied by the assignment that sets the variables in S to 1 and all other variables to 0?

2. Feedback Vertex Set

   Input: A graph G and a positive integer k.
   Question: Does there exist a subset $S \subseteq V(G)$ of size at most k such that graph $G - S$ is acyclic?

In the area of exact exponential-time algorithms, the objective is to design algorithms that outperform brute-force for computationally intractable problems. For subset problems, the brute-force algorithm that tries all possible solutions has running time $2^n n^{\mathcal{O}(1)}$. Thus our goal is typically to design an algorithm with running time $c^n n^{\mathcal{O}(1)}$ for $c < 2$, and we try to minimize the constant c.

Def. *Implicit set system* is a function $\Phi : I \to (U_I, \mathcal{F}_I)$, where $I \in \{0,1\}^*$ is referred to as an *instance*, and we denote by $|U_I| = n$ the size of the universe and by $|I| = N$ the size of the instance. We assume that $N \geq n$.

The implicit set system Φ is said to be polynomial time computable if (a) there exists a polynomial time algorithm that given I produces U_I , and (b) there exists a polynomial time algorithm that given I, U_I and a subset S of

U_I determines whether $S \in F_I$.

Φ-Subset Problem.
Input: An instance I
Question: A set $S \in F_I$ if one exists.

Φ-Extension.
Input: An instance I, a set $X \subseteq U_I$, and an integer k
Question: Does there exists a subset $S \subseteq (U_I \setminus X)$ such that $S \cup X \in F_I$ and $|S| \leq k$?

Theorem 1. If there exists an algorithm for Φ-Extension with running time $c^k N^{\mathcal{O}(1)}$ then there exists a randomized algorithm for Φ-Subset with running time $(2 - \frac{1}{c})^n N^{\mathcal{O}(1)}$.

Approximate proof. To prove it, we use *monotone* local search: we randomly sample a set X, which is supposed to be a subset of the answer S, and then run the parameterized algorithm for Φ-Extension in $c^{|S|-|X|} N^{\mathcal{O}(1)}$ time. We run this algorithm for each possible $k' = |S|$ and repeat it $\frac{\binom{n}{|X|}}{\binom{k'}{|X|}}$ times. We choose the size of $|X|$ in such way that the running time is minimized.

Theorem 2. If there exists an algorithm for Φ-Extension with running time $c^k N^{\mathcal{O}(1)}$ then there exists an algorithm for Φ-Subset with running time $(2 - \frac{1}{c})^{n+o(n)} N^{\mathcal{O}(1)}$.

Approximate proof. To prove it, we need to derandomize our algorithm.

Let U be a universe of size n and let $0 \leq q \leq p \leq n$. A family $\mathcal{C} \subseteq \binom{U}{q}$ is an (n, p, q)-set-inclusion-family, if for every set $S \in \binom{U}{p}$, there exists a set $Y \in \mathcal{C}$ such that $Y \subseteq S$.

Let $\kappa(n, p, q) = \frac{\binom{n}{q}}{\binom{p}{q}}$. There is a deterministic construction of an (n, p, q)-set-inclusion-family, $\mathcal{C}$, of size at most $\kappa(n, p, q) \cdot 2^{o(n)}$. The running time of the algorithm constructing $\mathcal{C}$ is also upper bounded by $\kappa(n, p, q) \cdot 2^{o(n)}$. Now, instead of $\kappa(n, k', |X|)$ independent repetitions of sampling X, the new algorithm loops over all $Y \in \mathcal{C}$.

Sunflower conjecture: We have some family of sets of size r. Let's call (k, r) sunflower the following thing: k sets, which have common intersection (core), and other parts for every 2 sets (petails) are disjoint. Famous conjecture says that size of family without sunflowers can'e be no more then c_k^r for some constant c_k depending only on k.

Let k be fixed now ($k = 3$, in fact is doesn't matter much).

Old result of Erdos says that size of sunflower-free set is no more then $w^{O(w)}$. In the work is proven that size of sunflower-free set is no more then $(logw)^{O(w)}$. Brief idea:

1)We will make definion of sunflower more flexible. So definition:

Set system F is (α, β) **satisfying** , if $Pr_Y(\exists S \in F, S \subset Y) > 1 - \beta$.

Robust sunflower - Let F be set system, K - common intersection, K not if F. F is (α, β) sunflower, if F_K is α, β satisfying, where F_K is the following - we consider only sets which contain K, and F_K is image of this sets after cutting K.

Why it is useful - $(\frac{1}{r}, \frac{1}{r})$ robust sunflower contains usual r-sunflower. Proof is simple randomness argument.

Now we want to find big robust sunflower. Our idea is consider elements with weights. Let's define **weight profile** - vector $s = (1 \geq s_0 \geq s_1 \geq \ldots \geq s_k)$ of rational numbers. Then we define **weighted set system (WSS)** as set of sets with some weights.

$o(F) = \sum_{F' \subset F} w(F')$. WSS is s-bounded, if $o(F) \geq s_0$, $O(F') \leq s_{|W'|}$ for all smaller sets. Finally, weight profine is α, β **satisfying** , if any s-bounded set system is (α, β) satisfying.

Then if weight profile $(1, k, k^2, \ldots, k^l)$ is (α, β) satisfying for every l less the w, then any w-set system of size k^{-w} is α, β satisfying.

In work is proven that for $k = logw(\frac{loglogwlog(\frac{1}{\beta})}{\alpha})^{O(1)}$ this weight profile is (α, β) satisfying.

Proof is complicated induction with rebuilding weights. Then we use it with $\alpha = \beta = \frac{1}{r}$

Definition 1. *The tensor product $G \times H$ of finite simple graphs G and H has vertex set $V(G) \times V(H)$, and pairs (g,h) and (g',h') are adjacent if and only if $\{g,g'\} \in E(G)$ and $\{h,h'\} \in E(H)$.*

It's obvious that

$$\chi(G \times H) \leq \min\{\chi(G), \chi(H)\}.$$

Main result 1.

$$\exists G, H : \chi(G \times H) < \min\{\chi(G), \chi(H)\}.$$

Some tools for proof:

Definition 2. *Let c be a positive integer, and let Γ be a finite graph that we allow to contain loops; the exponential graph $\mathcal{E}_c(\Gamma)$ has all mappings $V(\Gamma) \to \{1, \ldots, c\}$ as vertices, and two distinct mappings φ, ψ are adjacent if, and only if, the condition $\varphi(x) \neq \psi(y)$ holds whenever $\{x,y\} \in E(\Gamma)$.*

The relevance of $\mathcal{E}_c(\Gamma)$ to the problem is easy to see because the graph $\Gamma \times \mathcal{E}_c(\Gamma)$ has the proper c-coloring $(h, \psi) \to \psi(h)$.

Definition 3. *For a simple graph G, we define the strong product $G \boxtimes K_q$ as the graph with vertex set $V(G) \times \{1, \ldots, q\}$ and edges between (u,i) and (v,j) when, and only when, $\{u,v\} \in E(G)$ or $(u = v) \& (i \neq j)$.*

Lemma 1. *Let G be a finite simple graph with finite girth ≥ 6 Then, for sufficiently large q, one has $\chi(\mathcal{E}_c(G \boxtimes K_q)) > c$ with $c = \lceil 3.1q \rceil$.*

Some other things from report:

Definition 4. *Given positive integers $p \geq q$, a (p,q)-colouring of a graph G is a mapping $c : V(G) \to \{0, 1, \ldots, p-1\}$ such that for any edge (x,x') of G, $q \leq |c(x) - c(x')| \leq p - q$. The circular chromatic number $\chi_c(G)$ of G is the infimum of $\frac{p}{q}$ for which G has a (p,q)-colouring.*

It's obvious that

$$\chi_c(G \times H) \leq \min\{\chi_c(G), \chi_c(H)\}.$$

The question about inequality is open.

Definition 5. *A fractional colouring of a graph G is a mapping f which assigns to each independent set I of G a real number $f(I) \in [0,1]$ such that for any vertex x, $\sum_{x \in I} f(I) = 1$ The total weight $w(f)$ of a fractional colouring f of G is the sum of $f(I)$ over all the independent sets I of G. The fractional chromatic number $\chi_f(G)$ of G is the minimum total weight of a fractional colouring of G.*

It's obvious that

$$\chi_f(G \times H) \leq \min\{\chi_f(G), \chi_f(H)\}$$

and it's true that

$$\chi_f(G \times H) = \min\{\chi_f(G), \chi_f(H)\}.$$

Sensitivity Conjecture

Theorem 1 (Not the Sensitivity Conjecture itself, but equivalent)**.** *Any set H of $2^{n-1}+1$ vertices of the n-cube contains a vertex with at least $\sqrt{n}$ neighbors in H.*

The proof can be found here: `https://www.cs.stanford.edu/~knuth/papers/huang.pdf`. I suggest you to read the Knuth's version, because I can't make it shorter or simpler. As I remember, there was some hard equation in the end, but can be easily proven if we change it to ... $\leq$ ... inequality. The original equality is also true (we don't need its full strength), but it uses some observations about eigenvalues of the matrix A, which can be found in the original paper.

Connection to Boolean functions

Let $e_i \in \{0,1\}^n$ denote an n-bit Boolean string whose i^{th} bit is 1 and the rest of the bits are 0. Consider $f : \{0,1\}^n \to \{0,1\}$ be a Boolean functions. On an input $x \in \{0,1\}^n$, the i^{th} bit is said to be sensitive for f if $f(x \oplus e_i) \neq f(x)$, i.e., flipping the i^{th} bit results in flipping the output of f. The sensitivity of f on input x, denoted by $s(f,x)$, is the number of bits that are sensitive for f on input x.

Definition 1. *The sensitivity of a Boolean function f, denoted by $s(f)$, is the maximum value of $s(f,x)$ over all choices of x.*

For $B \subseteq \{1,2,...,n\}$ let $e_B \in \{0,1\}^n$ denote the characteristic vector of B. We say that a "block" B is sensitive for f on x if $f(x \oplus e_B) \neq f(x)$. The block sensitivity of f on x, denoted by $bs(f,x)$, is the maximum number of pairwise disjoint sensitive blocks of f on x.

Definition 2. *The block sensitivity of a Boolean function f, denoted by $bs(f)$, is the maximum possible value of $bs(f,x)$ over all choices of x.*

Block sensitivity seems to be somehow unnatural at first glance, but it is polynomially related to many others measures.

Theorem 2 (Sensitivity Conjecture)**.** $bs(f) \leq \mathsf{poly}(s(f))$

Theorems 1 and 2 are known to be equivalent.

Definition 3. *The deterministic decision-tree complexity of a Boolean function f, denoted by $D(f)$, is the depth of a minimum-depth decision tree that computes f.*

Definition 4. *A polynomial $p : \mathbb{R}^n \to \mathbb{R}$ represents f if*

$$\forall x \in \{0,1\}^n \; p(x) = f(x)$$

The degree of a Boolean function f, denoted by $deg(f)$, is the degree of the unique multilinear polynomial that represents f.

	$bs(f)$	$D(f)$	$deg(f)$
$bs(f)$	$1(1)$	$1(1)$	$2(\log_3 6)$
$D(f)$	$3(2)$	$1(1)$	$2(\log_3 6)$
$deg(f)$	$2(\log_4 5)$	$1(1)$	$1(1)$

The table above should be read in the following way:

$$\begin{aligned} bs(f) &= \mathcal{O}((deg(f))^2) \\ bs(f) &= \Omega((deg(f))^{\log_3 6}) \end{aligned}$$

Albina Lialina
albinapt1u@gmail.com

Abstract

We consider the problem of finding a simultaneous root to a system

$$\begin{cases} P_1(\overline{x}) = 0 \\ \dots \\ P_m(\overline{x}) = 0, \end{cases}$$

where P_i are polynomials of degree at most d and coefficients from $\mathbb{F}_2$ and $\overline{x}$ also is from $\mathbb{F}_2$. It is well known that this problem is NP-complete, and we will prove better than brute force bound $O^*(2^{(n-n/(2.7d))})$.

The task of finding the solution reduces to a decision problem. In its turn, the task of deciding whether there is a solution reduces to computing the parity of the number of such solutions. It can be done by isolation technique proposed by Valiant and Vazirani, which inserts $O(n)$ random linear equations into the system. And the new system with good probability has only one solution if the old one had any.

So, we want to compute the parity of the number of solutions. We will solve this problem recursively. The following scheme illustrate the reduction to instance of itself. From now on all arithmetic is over $\mathbb{F}_2$.

1. **Parity as a sum**
   Determining the parity of the number o solutions amounts to computing the sum
   $$I_F = \sum_{x \in \{0,1\}^n} F(x),$$
   where $F(x) = (1 + P_1(x)) \dots (1 + P_m(x))$.

2. **Approximation F by probabilistic polynomials**
   The degree of F could be as big as dm. It is inconvenient for us so we approximate F with independently chosen polynomials $f_1, f_2, \dots f_s$, where every $f_i = (1 + R_1(x)) \dots (1 + R_l(x))$. All R_j are chosen independently for all f_i.

3. **Summing in parts**
   To make the algorithm faster we use the following trick. We draw $A \sqcup B = \{1, \dots n\}$. Then for every function f we have
   $$I_f = \sum_{X \in \{1,\dots n\}} f(X) = \sum_{Z \subset B} \sum_{X \subset A} f(X \cup Z) := \sum_{Z \subset B} I_{f|_A^{Z \to B}}.$$
   The last equality is just a notation that we are going to use further.

4. **Approximation of I_F**
   Suppose we have summed each approximation $f_1, f_2, \dots f_s$ in parts. Then to count I_F we use the majority function.
   $$I_Z = MAJ(I_{f_1|_A^{Z \to B}}, I_{f_2|_A^{Z \to B}}, \dots, I_{f_s|_A^{Z \to B}}),$$
   then we show that with a good probability $I_F = \sum_{Z \subset B} I_Z$.

5. **Summing the parts and reduction to Parity**
   Recall that $f_i = (1 + R_1(x)) \dots (1 + R_l(x))$. Let's denote
   $$Q_j = R_j|_A^{Z \to B},$$
   then to sum the parts we need to compute the following
   $$f|_A^{Z \to B}(x) = (1 + Q_1(x)) \dots (1 + Q_l(x)),$$
   which is equivalent to computing the parity of the number of solutions of the following system
   $$\begin{cases} Q_1(\overline{x}) = 0 \\ \dots \\ Q_l(\overline{x}) = 0, \end{cases}.$$
   That is how we got reduction to several smaller but similar problems. And if you choose all the parameters wisely we get the bound that was mentioned above.

Informally, a DFA is some kind of machine with finite memory, that processes the input string strictly from left to right. Having n states is equivalent to having approximately $\log n$ bits of memory.

Formally, a DFA over some *alphabet* Σ is a finite directed graph, with every vertex (or *state*) having exactly $|\Sigma|$ outgoing arcs: one for every symbol from Σ. One of the states is labeled as *initial*, and some subset of them is labeled as *accepting*. Automaton *accepts* some string s, if reading s along the arcs makes it go from the initial vertex to some accepting vertex.

Suppose that you want to construct a small deterministic finite automaton (DFA) that accepts some string s but rejects some other string t. What is the minimal possible number of states in such an automaton? This is, in a sense, one of the simplest possible computational problems.

Theorem 1. *If s and t have different length, than there is an automaton with $O(\log \max(|s|, |t|))$ states that separates them.*

A sketch. Compute the length of the input string modulo all numbers m that are less than $O(\log \max(|s|, |t|))$. For one of them, the remainders will not be equal. $\square$

From now on, we consider only the case $|s| = |t| = n$ and the alphabet Σ is binary. Larger alphabets can be encoded in binary by using $\lceil \log_2 |\Sigma| \rceil$ bits.

Simple counting heuristics suggest that the minimal separating DFA should have $O(\log n)$ states. Moreover, numeric evidence for small n suggest the same upper bound as well. However, all simple constructions fail miserably. For example, there are two string of length n, such that *all* polynomial hashes with modulo being at most $O(\log n)$ coincide on such strings.

It is known that there exists a separating automaton with $O(n^{2/5} \log^{3/5} n)$ states, but the proof is very technical and involves a lot of case-checking. Hence, I recited a more beautiful proof of slighlty weaker result:

Theorem 2. *For different strings s and t of length n over binary alphabet, there is a separating DFA with $O(\sqrt{n})$ states.*

A sketch. Consider the function $f_{m,x}(w)$ that computes the parity of number of 1 on the positions that have remainder x modulo m. This function can be computed with an automaton with $O(m)$ states.

As it turns out, for every two strings s and t, there is always $m = O(\sqrt{n})$ and $x \in [0, m)$, such that $f_{x,m}(s) \neq f_{x,m}(t)$. Hence, there is a separating automaton with $O(\sqrt{n})$ states. The proof is pretty cool, but I will omit it, because Hirsh most probably won't ask anything about it. If you are interested, you can see the original paper (J. M. Robson, "Separating words with machines and groups"). $\square$

The summary for the article of Michal Kunc "The Power of Commuting with Finite Sets of Words".

16 декабря 2019 г.

This summary has been written by Martynova Olga.

1 The main definitions and notations.

The finite set A will be the alphabet in following definitions.

The set A^+ is the set of non-empty finite words over the alphabet A.

The set A^* is the set of all words over the alphabet A; $A^* = A^+ \cup \{\varepsilon\}$, where ε is the empty word.

The languages over the alphabet A are arbitrary subsets of A^*.

The main operations on languages:

- Union. For languages K and L we can consider their union $K \cup L$.
- Concatenation. The concatenation of K and L is $KL = \{uv \mid u \in U, v \in V\}$.
- The Kleene star. This operation is denoted as L^* for language L, and $L^* = \cup_{m \in \mathbb{N}_0} L^m$, where $L^m = LL \ldots L$ (m times), $L^0 = \{\varepsilon\}$.
- Complementation. The complementation of L is the set $A^* \setminus L$.

Regular languages are languages definable by finite automata, or equivalently, by rational expressions. Every regular language can be obtained from finite languages using union, concatenation and the Kleene star.

The language is called star-free if it can be obtained from finite languages using the operations of union, complementation and concatenation.

Let us consider the commutation equation $XL = LX$ for language L.
The language $C(L)$ is the largest language over A, which commutes with L. ($C(L)$ is the largest solution of the equation $XL = LX$, all other solutions are subsets of $C(L)$).
The language $C^+(L)$ is the largest ε-free language (without empty word) which commutes with L.

2 The main results.

Theorem 1. *There exists a star-free language L such that:*

1. *The largest language commuting with L is not recursively enumerable.*
2. *The difference between the largest language commuting with L and the largest ε-free language commuting with L is not recursively enumerable.*

The following theorem is the main result of the paper.

Theorem 2. *There exists a finite language L such that the largest language commuting with L and the largest ε-free language commuting with L are not recursively enumerable.*

These theorems are important results in theory of language equations, the proofs of these theorems are very difficult and technical.

Parsing Expression Grammars

PEG = (Σ (alphabet), N (nonterminals), R (rules, one per nonterminal), S (starting symbol)).

Parsing expression tries to recognise a substring straight ahead, left-to-right (i.e., a prefix of the remaining string) and remove it from the string (unless stated otherwise). Language consists of strings which are entirely recognised by the entire expression.

Parsing expressions are comprised of Σ, N, ε, FAIL, and their closure under the following operations:

- fg – tries to recognise f, then tries g on the remainder
- f/g – tries to recognise f, then tries g on the same string if the first attempt FAILs
- $\&f$ – tries to recognise f, then forgets if anything was removed (but remembers FAIL)
- $!f$ – tries to recognise f, then forgets if anything was removed, FAILing iff the internal expression doesn't.

A symbol from the alphabet is removed if it matches and FAILs if it does not; a nonterminal symbol is replaced with the corresponding rule, then recognition continues.

Can be recognised in linear time by going right-to-left instead of left-to-right.

Scaffolding automata

SA = (Σ (alphabet), Γ (internal alphabet), d (degree), k (depth), Q (states), q_0 (initial state), F (accepting states), δ (transition function)).

Builds a d-regular (with d outcoming edges being distinctly labelled with numbers from 1 to d, some possibly absent) directed graph as it recognises a string. The initial graph consists of a single vertex with no label. Transition depends on the internal state, the labels in the k-neighbourhood of the latest vertex, and the freshly read symbol; it produces a new state, a label for the new vertex, and a list of d vertices from the neighbourhood (some possibly absent) to point the new edges into. Accepts if the state is accepting.

Central theorem: a language is recognised by a PEG iff its reverse is recognised by a SA.

Proof: simulation in both directions.

Theorem 2: for every computable function *f* there is such a computable function *g* that the language $\{f(x)\#^{g(x)}x\}_x$ is recognised by a PEG.

Proof (was not finished in time): use the scaffolding automaton to simulate a Turing machine, using the extra # symbols to write the computation states on the internal vertex labels in a back-and-forth fashion (copy the last in reverse order, then flip again while processing the step).

Theorem 3: There is no pumping lemma for PEGs (that is, a totally computable function *A* that for every PEG *G* and every large enough ($|x| > n_G$) string *x* from *L(G)* the output $y = A(G, x)$ is in *L(G)* and $|y| > |x|$).

Proof (was not finished in time): use Theorem 2 and a bit of recursive trickery to construct an automaton that always recognises a bigger next string than a given function predicts.

Separation Logic language is incomplete

The Hoare logic is a common tool for software verification. Its statements are written as triples like $\{P\}instr\{Q\}$. Such notation states that if P holds then after execution of instruction $instr$ Q will hold.

Unfortunately such approach complicates reasoning about complex programs. For example, one can easily prove that a single instruction doesn't overwrite all available memory but it's hard to prove the same for a large program.

We note though that a program can be usually split into functions that modify separate memory parts and therefore don't interfere with each other. It's easier to prove that if a property holds for such a function then it also holds for a whole program.

This idea underlies the separation logic. This is a Hoare logic extension which allows to express the fact that the memory is split between different functions. For example, the statement ”the memory consists of exactly two locations with addresses x and y with values 1 and 2 correspondingly”can be rephrased as ”the memory can be split into two disjoint parts, each consisting of a single location x (or y) and value 1 (or 2)”. The latter statement can be written in the separation logic language as $(x \mapsto 1) * (y \mapsto 2)$.

The satisfiability of this language's formula depends on memory state. Notation $h \models \phi$ denotes that if memory is described with function $h : Addresses \to Values$ then the formula ϕ holds. It can be seen that the tautological formulas of FOL are similar to this language's valid statements — those that hold with any memory state.

For separation logic the validity problem is undecidable. It follows from the non-recursively-enumerability of this logic's language [1].

The non-r.e. is proved by reduction from other non-r.e. language. It is the set of such formulas of FOL language with only one binary predicate that hold in every finite structure. The non-r.e. of this language has been shown by Trakhtenbrot.

For simpler versions of this language the validity problem is decidable though. For example, the validity for language without quantifiers and $\mapsto$ predicate is NP-complete.

[1]by Post theorem

1 Definitions

1.1 Tseitin formulas

Let $G(V, E)$ be a graph. Let $c : V \to \{0, 1\}$ be a **charge function**. A Tseitin formula $\mathrm{T}(G, c)$ depends on the propositional variables x_e for $e \in E$. For each vertex $v \in V$ we define the parity condition of v as

$$P_v := \left(\sum_{e \text{ is incident to } v} x_e \equiv c(v) \bmod 2 \right).$$

The **Tseitin formula** $\mathrm{T}(G, c)$ is the conjunction of parity conditions of all the vertices: $\bigwedge_{v \in V} P_v$. Tseitin formulas is represented in CNF as follows: we represent P_v in CNF in the canonical way for all $v \in V$.

Lemma 1. *A Tseitin formula $\mathrm{T}(G, c)$ is satisfiable if and only if for every connected component $C(U, E_U)$ of the graph G, the condition $\sum_{u \in U} c(u) \equiv 0 \bmod 2$ holds.*

1.2 NBP and 1-NBP

A **nondeterministic branching program** (NBP) represents a Boolean function $f : \{0, 1\}^n \to \{0, 1\}$. A nondeterministic branching program for a function $f(x_1, x_2, \dots, x_n)$ is a directed acyclic graph. It has three types of nodes: guessing nodes (OR nodes), nodes labeled with a variable (we call them just labeled nodes) and two sinks; the unique source is either a guessing node or a labeled node.

- Sinks are labeled with 0 and 1.
- Each of labeled nodes labeled with a variable from $\{x_1, x_2, \dots, x_n\}$ and has exactly two outgoing edges: the first edge is labeled with 0, the second is labeled with 1.
- Each of guessing nodes has two outgoing unlabeled edges.

The value of every node is defined recursively. Each node v of a branching program computes a function $f_v : \{0, 1\}^n \to \{0, 1\}$.

- For a $k \in \{0, 1\}$, the sink s labeled with k, computes the function $f_s \equiv k$.
- Assume that a node v is labeled with x_i, the outgoing edge from v labeled with 0 ends in a node v_0 and the outgoing edge labeled with 1 ends in a node v_1. Then $f_v(x_1, \dots, x_n)$ equals $f_{v_1}(x_1, \dots, x_n)$ if $x_i = 1$ and equals $f_{v_0}(x_1, \dots, x_n)$ if $x_i = 0$.
- Assume that a node v is a guessing node and v_0 and v_1 are two direct successors of v. Then $f_v(x_1, \dots, x_n)$ equals $f_{v_1}(x_1, \dots, x_n) \vee f_{v_0}(x_1, \dots, x_n)$.

We say that the branching program computes the function computed in its source.

The size of a nondeterministic branching program is the number of nodes in it.

A nondeterministic branching program is **read-once** (1-NBP) if every path in it contains at most one occurrence of each variable.

1.3 BP and 1-BP

A **branching program** (BP) is a NBP such that there are no guessing nodes.

A **read-once** branching program (1-BP) is a 1-NBP such that there no guessing nodes.

1.4 OBDD

Let π be a permutation of the set $\{1, \dots, n\}$ (an order).

A **π-ordered nondeterministic binary decision diagram** ($\pi-$NOBDD) is a 1-NBP such that on every path from the source to a sink variable $x_{\pi(i)}$ can not appear before $x_{\pi(j)}$ if $i > j$. A **nondeterministic ordered binary decision diagram** (NOBDD) is a π-ordered nondeterministic binary decision diagram for some π.

A **π-ordered binary decision diagram** ($\pi-$OBDD) is a $\pi-$NOBDD such that there are no guessing nodes. A **ordered binary decision diagram** (OBDD) is a π-ordered binary decision diagram for some π.

2 Minimal read-once branching program for satisfiable Tseitin formulas is OBDD

Our goal is to prove that if one wants to compute the value of satisfiable Tseitin formula, then NBP is not stronger (in terms of size) than OBDD.

We use the notation $\#G$ for the number of connected components of G.

Lemma 2. *If a Tseitin formula* $\mathrm{T}(G, c)$ *is satisfiable, then it has* $2^{|E|-|V|+\#G}$ *satisfiable assignments.*

Lemma 3. *For every* $f \in \mathcal{F}_J(G, c)$ *the size of the set* $\{\alpha \in A_{G,c} \mid f = \mathrm{T}(G, c)|_{\alpha_J}\}$ *equals* $2^{|E|-2|V|+\#G_{E\setminus J}+\#G_J}$.

For a graph $G(V, E)$ and a set of edges $J \subseteq E$ we introduce the notation $\mathrm{comp}_J(G) = |V| - \#G_{E\setminus J} - \#G_J + \#G$.

Lemma 4. *The size of the set* $\mathcal{F}_J(G, c)$ *is equal to* $2^{\mathrm{comp}_J(G)}$.

Proof. $2^{\mathrm{comp}_J(G)} = 2^{|E|-|V|+\#G}/2^{|E|-2|V|+\#G_{E\setminus J}+\#G_J}$. □

Lemma 5. *Let D be a* 1-NBP *computing a satisfiable* $\mathrm{T}(G, c)$. *Let s be a node of D such that there is an accepting path passing through s. Then the following holds:*

1) every two paths from the source to s assign values to the same set of variables $\{x_e \mid e \in J\}$, *where* $J \subseteq E$;

2) the maximal number of accepting paths passing through s corresponding to different satisfying assignments of $\mathrm{T}(G, c)$ *is at most* $2^{|E|-2|V|+\#G_{E\setminus J}+\#G_J}$.

Let D be a nondeterministic OBDD using the order of variables $x_{\pi(1)}, x_{\pi(2)}, \dots, x_{\pi(n)}$. For every $i \in [n]$, the i-th level of D is the set of nodes labeled with variable $x_{\pi(i)}$.

Lemma 6. *Let* $\mathrm{T}(G, c)$ *be a satisfiable Tseitin formula based on a graph* $G(V, E)$. *Let π be a permutation of* $[|E|]$. *For every i from 0 to $|E|$ we denote by J_i the set* $\{e_{\pi(1)}, e_{\pi(2)}, \dots, e_{\pi(i)}\}$ *of the first i edges according to permutation π.*

1) Let D be a nondeterministic π-ordered OBDD *computing* $\mathrm{T}(G, c)$. *Then for every i from* $[|E|]$, *the i-th level of D contains at least* $2^{\mathrm{comp}_{J_{i-1}}(G)}$ *nodes.*

*2) If D is a minimal π-*OBDD *computing* $\mathrm{T}(G, c)$, *then the i-th level of D has exactly* $2^{\mathrm{comp}_{J_i}(G)}$ *nodes and for every node s from the i-th level of D there are exactly* $2^{|E|-2|V|+\#G_{E\setminus J_{i-1}}+\#G_{J_{i-1}}}$ *accepting paths going through s.*

Lemma 7. *The size of any* 1-NBP *computing a satisfiable* $\mathrm{T}(G, c)$ *is at least the minimal size of* OBDD *computing* $\mathrm{T}(G, c)$.

Sketch. Consider a minimal read-once nondeterministic branching program D computing $\mathrm{T}(G, c)$. Consider a set of accepting paths P.

For every node v of D we denote by $q(v)$ the number of paths from P passing through v. For every $p \in P$, we consider the value $\gamma(p) = \sum_{v\in p} \frac{1}{q(v)}$, where summation goes over all vertices of p.

Notice that $\sum_{p\in P} \gamma(p) = |D| - 1$, since every node v except the 0-sink was calculated $q(v)$ times with weight $\frac{1}{q(v)}$. Hence, $|D| - 1 \geqslant |P| \min\limits_{p\in P} \gamma(p)$. Let $\min\limits_{p\in P} \gamma(p)$ be achieved on a path $p^* \in P$. Let π be a permutation corresponding to the order of the edges in p^* in the direction from the source to the 1-sink.

Let D' be a minimal $\pi-$OBDD computing $T(G, c)$, let P' be set of accepting paths of D', and let p' be the path in D' corresponding to the path p^* in D. For any node v of D' we denote by $q'(v)$ the number of accepting paths passing through v. For any path p in D' we define $\gamma'(p) = \sum_{v \in p} \frac{1}{q'(v)}$.

By previous lemmas, $\gamma'(p)$ does not depend on p, and $q(v) \geqslant q'(v')$ for each pair of corresponding vertices v and v' on the paths p' and p^*, so $\gamma(p') \leqslant \gamma(p^*)$ and

$$|D'| - 1 = \sum_{p \in P'} \gamma'(p) = |P'|\gamma(p') \leqslant |P|\gamma(p^*) \leqslant \sum_{p \in P} \gamma(p) = |D| - 1,$$

and, thus, $|D| \geqslant |D'|$. $\square$

An exact exponential algorithm for trees to find tropical connected set

Eugene Vagin

[07.11.19]

1 Intro

Paper: **Exact exponential algorithms to find tropical connected sets of minimum size**

We consider the problem of finding minimal tropical connected sets on colored trees. The main result is a branching algorithm which computes a minimum tropical set on trees in time $\mathcal{O}^*(1.2721^n)$

2 Definitions

- $T = (V, E)$: tree, V - set of vertices, E - set of edges
- $\forall\, X \subseteq V \;\; G[X]$: subgraph induced by X
- $N(v)$: all neighbours of vertex v
- $N[v] = N(v) \cup \{v\}$
- $S \subseteq V$: **connected**, iff $G[S]$ - connected
- $C = \{c(v) : v \in V\}$: set of colors for G
- (T, c) : colored graph
- $c(S) = \{c(v) : v \in S\}$: set of colors of S
- S **is tropical**, if $c(S) = C$

3 Branching

Branching belongs to main tools to design exact exponential algorithms
Idea: solving instance of problem by recursively branching into instances of subproblems of smaller sizes via branching and reduction rules
Let $T(n)$ - upper bound for the running time of algorithm, when applied to any instance of the size n.
Branching rule: the rule which splits problem to subproblems (e.g. $n \to n - t_1, n - t_2, .., n - t_b$).
Branching vector $= (t_1, .., t_b) \sim \tau(t_1, .., t_b)$. Assumed that $T(n) \leq \sum_1^b T(n - t_i)$
Branching number: the largest $\lambda \in \mathbb{R}$ of equation $\lambda^n - \sum_1^b \lambda^{n-t_i} = 0$.
If there are different branching vectors suitable, λ_{max} is choosen
Finally, the upper bound of running time of algorithm is $\mathcal{O}^*((\lambda_{max})^n)$

4 Instances and subproblems

Again: we want to find minimal tropical connected set (**TCS**) on colored tree.
Let instance of subproblem is 3-partition of V: (S, F, D)

- S: selected vertices, which belong to any solution
- F: free vertices, later its will be moved to S or D
- D: discarded vertices, $\forall v \in D \;\; v \notin \{any\, solution\}$

Our task is to find TCS S^* of (T, c) (solution of (S, F, D)) with next properties:

- $S \subseteq S^*$
- $S^* \cap D = \emptyset$
- S^* is minumum size

Satisfied properties:

1. root r of T belongs to S
2. S and $S \cup F$ is a connected sets of T
3. $\forall v \in D \;\; T(v) \in D$

We will also use next notations:

- **T'** $:= G[S \cup F]$ - subgraph induced by $S \cup F$
- **C'** $:= C \setminus c(S)$ - colors which is not yet appear in S

Also we will consider size of F as size of subproblem instance

5 Description of algorithm

The main idea: we try to build the minimal TCS rooted at each vertex of graph:

```
for each $v \in V$ do
  TCS-Tree(T=T(v), (S, F, D) = ({v}, V \$\{v\}, \emptyset$))
```

Also two routines used:

- **ADD(v)**: moves all nodes between r and v from F to S and updates C'.
- **RMV(v)**: moves all subtree of v from F to D

The main ideas of reduction rules:

- $C' = \emptyset \Rightarrow S$ is a solution
- $F = \emptyset \Rightarrow$ solution not found (exit) (note that we assume that all previous items was checked)
- vertices which colors $\notin C'$ can be eliminated (moved from F to D). Also vertices which color is unique in S should be added (moved from F to S)
- if for leaf vertex v exist vertex u with same color which is "cheaper"to add, we add it (e.g. $dist(S, u) = 1$)

There are 3 major rules, each rule have subrules:

- **B1**: $\exists v \in F : dist(v, S) \geq 4$
- **B2**: $\exists v \in F : dist(v, S) = 3$
- **B3**: $\exists v \in F : dist(v, S) = 2$

Subrules looks similar, so I'll consider only one subrule: **B1.(a)**.

if $\exists$ internal $v', v'' \in F \setminus \{v\} : c(v) = c(v') = c(v'')$ **then** $(\{ADD(v), RMV(\{v', v''\})\} \mid\mid RMV(v))$
When $ADD(v)$ called, we should move $path(v, S)$ from F to S. Note that $path(v, S) \geq 4$ vertices.
For calling $RMV(\{v', v''\})$ exists two cases:

- v' is ancestor of v'' (or vice-versa): in this case at least **3** vertices should be moved from F (because v' is internal and has at least two children: v'' and child of v'')
- v' is not ancestor of v'' (and vice-versa): in this case at least **4** vertices should be moved from F (because v' and v'' has at least 1 child)

When $RMV(v)$ is called, we can move from F only one vertex (because v is leaf).
So, finally, instance with size n is branched to $(n-7, n-1)$ or $(n-8, n-1)$. Branching vector is $\tau(7, 1) \leq 1.2555$ (we consider the worst case).

Similarly can be considered other subrules. The worst obtained branching vector is $\tau(4, 2) \leq 1.2721$.
So, the running time of this algorithm is $\mathcal{O}^*(1.2721^n)$

Quantified Boolean Formula

December 17, 2019

1 quantified boolean formula

main proof: Satisfiability of 2-QB-3CNF with n variables and poly(n) clauses can be solved in time$2^{n-\Omega(n^{1/2})}$

To proof this, we need to use a lemma proofed in the other paper.

lemma:there is a deterministic algorithm A solving satisfiability of CNFs with m clauses and n variables in time O($poly(m)2^{n-n/(1+log(m))}$)

We will run a recursive algorithm R on the given 3-CNFs formula an n variables and two quantifier blocks.

n=the number of variables in the original formula.

u= the number of universal variables,e= the number of existential variables such that $n = u + e$

Here we divide the algorithm R into 4 cases, and fix a parameter ϵ later.

a) the trivial cases.

If there is a clauses containing only universal literals or an empty clauses, return 0, if there are no clauses left ti satisfy,return 1.

if there are no universal literals in the formula, the instance is a 3-SAT. If the 3-SAT is satisfiable, return 1, else return 0.

b) if $e > \sqrt{n}$

Try all $2n - e$ assignments to the universals and using lemma solve the remaining 3-CNF in 1.4^e time. this case takes $O^*(2^u 1.4^e)$

c) if $e \leq \sqrt{n}$

then suppose there exists a clause with two universal and 1 existe.ntial variables. Let it be $(u_i \vee u_j \vee e_k)$. Perform the next three recalls:

- set $(u_i = 1)$ and call the algorithm R on the formula, named **type A**
- set $(u_i = 0)$ $(u_j = 1)$.call R **type B**
- set $(u_i = 0)$,$(u_j = 0)$,$(e_k = 1)$ call R. **type C**

In this case, we first need to bound the number of leaves N of the recursion tree. we classify the leaves according how many times type C occur. denote i the number of type C occur, and $f(i)$ denote the number of leaves for which i calls of type C occur on the path from root to leaf, and $0 \leq i \leq \sqrt{n}$.

$f(0) = O(\phi^u) = O(\phi^n)$, $f(i) \leq C_n^i O(\phi^u)$, as there are C_n^i ways of choosing the i levels of the recursion tree at which calls of type C are made.

thus the number of leaves can be bounded by$N \leq \sum_{i=1}^{i=\sqrt{n}} f(i) \leq 2^{n-\Omega(n)}$

then we partition the leaves into deep leaves and shallow leaves, and divide the shallow leaves into light leaves and heavy leaves, sum the cost of each type of leaves we can obtained the cumulative cost which is $O(2^{n-\Omega(n)})$.

d) if all clauses contain at most one universal literal

- if $u < \epsilon n$, then solve the QBF using brute force search
- else, make a DNF with 2^e conjuncts, which has $O(2^e u)$ clauses, using lemma can solved in $O^*(2^{u-\Omega(u/e)})$ time