KTH Computer Science and Communication

Current Research in Proof Complexity: Problem Set 3

Due: February 6, 2012. Submit as a PDF-file by e-mail to `jakobn at kth dot se` with the subject line `Problem set 3: ⟨your name⟩`. Solutions should be written in LaTeX or some other math-aware typesetting system. Please try to be precise and to the point in your solutions and refrain from vague statements. In addition to what is stated below, the general rules stated on the course webpage always apply.

Hints: For most or all problems, "hints" can be purchased at a cost of 5–10 points. In this way, you can configure yourself whether you want the problems to be more creative and open-ended, where sometimes a lot can depend on finding the right idea, or whether you want them to be more of guided exercises providing a useful work-out on the concepts of proof complexity. If you do not solve a problem, there is no charge for the hint (i.e., it is not deducted from the score on other problems).

Collaboration: Discussions of ideas in groups of two to three people are allowed—and indeed, encouraged—but you should write down your own solution individually and understand all aspects of it fully. For each problem, state at the beginning of your solution with whom you have been collaborating. Everybody collaborating on a certain problem is considered to have purchased a hint for that problem if one of the collaborators has done so.

Reference material: Some of the problems might be "classic" with solutions easily found on the Internet or in research papers. It is not allowed to use such solutions in any way unless explicitly stated otherwise. Anything said during the lectures on in the lecture notes should be fair game, though, unless you are specifically asked to show something that we claimed without proof in class. It is hard to pin down 100% formal rules on what all this means—when in doubt, ask the lecturer.

About the problems: Some of these problems are meant to be quite challenging and you are not necessarily expected to solve all of them. As a general guideline, a total score of around 130 points on this problem set should be enough to get a pass. Any corrections or clarifications will be posted on the course webpage `www.csc.kth.se/~jakobn/teaching/proofcplx11`.

1 (40 p) *Unit propagation*, known in the SAT community as *Boolean constraint propagation (BCP)*, applied on a CNF formula F works as follows: If there is a *unit clause* in F, i.e., a clause of size 1 consisting of a single literal a, set this literal a to true, remove all clauses from F containing a, and shrink all clauses containing $\overline{a}$ by removing this literal. Repeat on the new CNF formula obtained in this way until either contradiction is derived (in the form of an empty clause) or there are no more unit clauses. Note that described in this way, unit propagation can be viewed as an (incomplete) proof system for refuting unsatisfiable CNF formulas.

As explained in the guest lecture on SAT solving, in a CDCL solver one tries to pick decision variables to set in such a way that there is a lot of unit propagation for every decision made. Intuitively, if we can make only a few variable decisions and then refute the rest of the formula by unit propagation, the formula is easy. In this problem, we want to study an extreme case of this situation and derive a correspondence of sorts between theory and practice.

1a Suppose that a CNF formula F is such that it can be refuted by unit propagation only (without any decision variables). Prove that this implies that $Sp_{\mathcal{R}}(F \vdash \bot) = \mathrm{O}(1)$. For full credit, determine the exact refutation clause space complexity in resolution.

1b Suppose that a CNF formula F is such that it can be refuted by a CDCL solver making only a constant number of decisions (and using unit propagation for the non-decision variables). Prove that this implies that $Sp_{\mathcal{R}}(F \vdash \bot) = \mathrm{O}(1)$. (Here the constant hidden in the big-oh notation will depend on the number of decisions, but there is no need to study the concrete numbers.)

Hint: You do not need to worry about clause learning or any fancy techniques for this problem, but can consider a simplified model where the solver branches on the values of variables and a run of the solver corresponds to a decision tree for the search problem of finding an unsatisfied clause for F.

2 (40 p) One important technique in SAT solving is to preprocess the input CNF formula F by removing so-called *blocked clauses*. A clause C in a CNF formula F is *blocked with respect to the literal* a if for every clause D in F that contains $\overline{a}$ there is also another literal $\overline{b}$ in D such that b is a literal in C (recall that we define $\overline{\overline{x}} = x$). Equivalently, C is blocked with respect to a if all resolvents over a involving C yield trivial clauses. (A clause is trivial if it contains both literals x and $\overline{x}$ for some variable x.)

2a Prove that if we start with a formula F and remove blocked clauses in any order until there are no more blocked clauses, we always get the same formula F'. (In somewhat more fancy terminology, blocked clause elimination is a *confluent* operation.)

2b Prove for the formula F' resulting from blocked clause elimination in F that F' is unsatisfiable if and only if F is unsatisfiable (i.e., the two formulas are *equisatisfiable*).

3 (60 p) Consider the *bitwise pigeonhole principle formula* $BPHP_n^m$ which has axiom clauses $\bigvee_{i=0}^{\ell-1} x[p_1, i]^{1-h_i} \vee \bigvee_{i=0}^{\ell-1} x[p_2, i]^{1-h_i}$ for every two pigeons $p_1 \neq p_2 \in [0, m)$ and every hole $h \in [0, n)$ stating that p_1 and p_2 do not both go to hole h, where $n = 2^\ell$ and $h_{\ell-1} \cdots h_0$ is the binary encoding of h. In class, we proved an $\Omega(n)$ lower bound on monomial space in PCR for these formulas, and we now want to take a closer look at this proof.

Recall that we defined a *commitment* to be a 2-clause of the form $x[p_1, i_1]^{b_1} \vee x[p_2, i_2]^{b_2}$, where $p_1 \neq p_2$, and a *commitment set* $\mathcal{A}$ to be a 2-CNF formula consisting of commitments where all pigeons are distinct. We used these commitment sets to capture information about PCR-configurations. More precisely, for a PCR-derivation $\pi = \{\mathbb{P}_0 = \emptyset, \mathbb{P}_1, \ldots, \mathbb{P}_\tau\}$ in small space we constructed commitment sets $\mathcal{A}_0 = \emptyset, \mathcal{A}_1, \ldots, \mathcal{A}_\tau$ such that $\mathcal{A}_i$ implied $\mathbb{P}_i$ in a certain sense, and this proved that no small-space derivation could refute $BPHP_n^m$.

3a In a key lemma, we proved that any commitment set $\mathcal{A}$ is in fact "super-satisfiable" in the sense that *both literals* in all commitments $x[p_1, i_1]^{b_1} \vee x[p_2, i_2]^{b_2}$ can be satisfied simultaneously. This raises the obvious question whether we could not just have single literals as commitments instead of 2-clauses.

Therefore, let us define a commitment to be a unit clause $x[p_1, i_1]^{b_1}$ and let a commitment set be a 1-CNF formula consisting of commitments where all pigeons are distinct. Does the proof we did in class still go through with this modification? Explain how to adapt the proof to make it work, or point out where it fails.

3b Suppose that we drop the requirement that $p_1 \neq p_2$ for a commitment $x[p_1, i_1]^{b_1} \vee x[p_2, i_2]^{b_2}$, but still insist that any two distinct commitments in the same commitment set $\mathcal{A}$ cannot mention the same pigeon. Does the proof we did in class still go through with this modification? Explain how to adapt the proof to make it work, or point out where it fails.

3c Suppose we change the definition of commitment to an exclusive or $x[p_1, i_1]^{b_1} \oplus x[p_2, i_2]^{b_2}$ requiring that one of the literals but not both should be true (and where we still have $p_1 \neq p_2$), and let a *commitment set* $\mathcal{A}$ be a "2-XOR formula" consisting of commitments where all pigeons are distinct. Does the lower bound proof still work with this definition of commitment sets? Explain how to adapt it or point out where it breaks down.

4 (30 p) Define a new sequential proof system *implicational monomial calculus (IMC)* as follows. Every line in a proof is a Boolean function $f(m_1, \ldots, m_d)$ where the arguments $m_1, \ldots, m_d$ are monomials over variables and negated variables, just as in PCR. Clauses are translated to monomials as in PCR. For inference steps, any function $f(m_1, \ldots, m_d)$ that follows semantically from the monomial functions $f_1(m_{1,1}, \ldots, m_{1,d_1}), \ldots, f_s(m_{s,1}, \ldots, m_{s,d_s})$ currently in memory can be derived in one step.

Let the size of a proof be the total number of monomials occuring in the proof (counted with repetitions) and let $S_{\mathit{IMC}}(F \vdash \bot)$ be the minimal size of refuting the formula F in implicational monomial calculus. Let the (monomial) space of a configuration be the total number of monomials (counted with repetitions) in all functions $f(m_1, \ldots, m_d)$ currently in memory, and let $Sp_{\mathit{IMC}}(F \vdash \bot)$ denote the minimal space of refuting the formula F.

4a Prove that IMC can simulate PCR-proofs without any blow-up in size or space.

4b Can you use the ideas from the PCR-space lower bound for $BPHP_n^m$ to prove a similar lower bound for IMC-space? Explain how to do this or point out where and why the analogous approach fails.

5 (60 p) In this problem, we consider the functional pigeonhole principle formula $FPHP_n^{n+1}$ discussed in lecture 9 and the standard 3-CNF version (as defined in lecture 5) $\widetilde{FPHP}_n^{n+1}$ of this formula.

Recall that $FPHP_n^{n+1}$ is the formula PHP_n^{n+1} with added axioms $\overline{x}_{i,j_1} \vee \overline{x}_{i,j_2}$ for all pigeons i and all holes $j_1 \neq j_2$ specifying that a pigeon can only go into one hole. Recall also that for a CNF formula $F = C_1 \wedge \cdots \wedge C_m$, for each C_j with $W(C_j) \leq 3$ we let $\widetilde{C}_j = C_j$, and for each C_j with $W(C_j) > 3$, say $C_j = a_1 \vee a_2 \vee \cdots \vee a_w$, we let $\widetilde{C}_j$ be the set of clauses

$$\left\{\overline{y}_{j,0}\,,\; y_{j,0} \vee a_1 \vee \overline{y}_{j,1}\,,\; y_{j,1} \vee a_2 \vee \overline{y}_{j,2}\,,\; \ldots\,,\; y_{j,w-1} \vee a_w \vee \overline{y}_{j,w}\,,\; y_{j,w}\right\} \tag{1}$$

where $y_{j,i}$ are new variables that are unique to $\widetilde{C}_j$ and do not appear anywhere else. With this notation, we define $\widetilde{F}$ to be the conjunction of all the clauses in $\widetilde{C}_j$ for $j = 1, \ldots, m$.

5a Prove that for any CNF formula F it holds that $Sp_{\mathcal{R}}\big(\widetilde{F} \vdash \bot\big) \leq Sp_{\mathcal{R}}(F \vdash \bot) + \mathrm{O}(1)$, i.e., that going to the 3-CNF version of a formula can never increase the clause space (at least not by more than a very small constant term).

5b Prove that for the functional pigeonhole principle, the opposite direction also holds in the sense that $Sp_{\mathcal{R}}\big(FPHP_n^{n+1} \vdash \bot\big) \leq Sp_{\mathcal{R}}\big(\widetilde{FPHP}_n^{n+1} \vdash \bot\big) + \mathrm{O}(1)$.

Hint: Regarding the auxiliary variables in Equation (1), it is natural to think of the positive literal $y_{j,i}$ as a shorthand for $\bigvee_{\ell \leq i} a_\ell$, since if $y_{j,i}$ is set to true then any satisfying assignment must also satisfy the clause $\bigvee_{\ell \leq i} a_\ell$. In the same way, the negative literal $\overline{y}_{j,i}$ corresponds to $\bigvee_{\ell > i} a_\ell$. Thus, a naive approach for proving the inequality above would be to substitute these clauses for $y_{j,i}$ and $\overline{y}_{j,i}$, respectively, in any resolution refutation of $\widetilde{FPHP}_n^{n+1}$ to get a refutation in essentially the same clause space for $FPHP_n^{n+1}$. This does not work in general for any CNF formula (why?), but for $FPHP_n^{n+1}$ this idea can actually be implemented with a bit of care. (For bonus points, explain exactly which property of $FPHP_n^{n+1}$ makes this go through. Would the same idea work also for PHP_n^{n+1}?)

5c Is it also true for PCR that we can relate $Sp_{\mathcal{PCR}}\big(FPHP_n^{n+1} \vdash \bot\big)$ and $Sp_{\mathcal{PCR}}\big(\widetilde{FPHP}_n^{n+1} \vdash \bot\big)$ in a way analogous to Problem 5b? Explain how to prove a similar statement, or explain why this does not work. (In the latter case we are not necessarily asking about a proof for the opposite statement, but just an explanation of why the approach sketched above fails.)

6 (60 p) *Kakuro*, or *Cross Sums*, is a crossword puzzle but with numbers instead of words. The empty cells, or squares, in the grid should be filled in so that each run of cells adds up to the total in the *clue square* above or to the left. Only numbers 1–9 should be used, and a number can never be used more than once per run (but can reoccur in the same row or column in another, separate run). Kakuro puzzle grids can be of any (rectangular) size.

A clue square, which we write as `d\a` below, can have a *down clue* or an *across clue*, or both. For an across clue `a`, the numbers in the blank cells to the right of the clue should sum to `a`. As already referred to above, these blank cells are said to constitute a *run*. For a down clue `d`, the cells forming the run that should sum to `d` are positioned below the clue square. For more details see, e.g., `en.wikipedia.org/wiki/Kakuro`.

The purpose of this problem is to investigate if and how SAT solvers can be used to solve Kakuro puzzles. For this problem, do not submit any code, but instead describe how it works. Place the actual code in a directory in the AFS file system where `jakobn` has reading and listing permission `rl` (as shown by `fs la .`). Note that permission `l` is needed for the whole path leading to the directory. Make sure your code works in the KTH CSC Ubuntu Linux environment. Include a Makefile in the directory, or a shellscript `make` that will compile your code. If there are problems with any of the above, contact the lecturer to agree on some other technical solution. For the SAT solving you should use MiniSAT. Some helpful practical information about MiniSAT and about the standard *DIMACS* format used by MiniSAT and other SAT solvers can be found on the webpage `www.csc.kth.se/~jakobn/teaching/proofcplx11/minisat.php`.

6a Describe a way to encode any given Kakuro instance as a CNF formula in such a way that the formula is satisfiable if and only if the Kakuro puzzle has a solution, and so that a solution to the puzzle can be read off from any satisfying assignment to the formula. The encoding should be explicit and have reasonable size and complexity.

In case you happen to consider several different options, describe what these are and discuss what you think are possible pros and cons. (All such variants should be correct, of course. Also, this is optional in the sense that only one correct encoding is needed for full credit.)

6b Write a program that generates your CNF encoding from problem 6a for a a given Kakuro instance and outputs it in DIMACS format. If given no arguments, the program should read a Kakuro instance from standard input and write the CNF formula on standard output.

The format of the Kakuro input file should be as in the following example:

```
8 x 6
 -\-  17\-  28\-   -\-  42\-  22\-
 -\9    -     -   31\14   -     -
 -\20   -     -     -     -     -
 -\-   -\30   -     -     -     -
 -\-  22\24   -     -     -    -\-
 -\25   -     -     -     -   11\-
 -\20   -     -     -     -     -
 -\14   -     -    -\17   -     -
```

That is, the first line `<int> x <int>` specifies the dimensions of the Kakuro puzzle grid (rows x colums). Then the puzzle grid follows, line by line. Clue squares are formatted like `d\a`, where `d` is a down clue and `a` is an across clue. A dash instead of a number `d` or `a` means that there is no clue. Empty cells `-` should be filled with numbers 1–9, and `-\-` denotes a cell that should not be filled in. There is at least one blank between each cell, but the number of blanks does not matter (we have just used some pretty-printing above to get columns aligned). Each line is terminated by a newline character.

Write another program that reads a satisfying assignment as produced by MiniSAT and pretty-prints a solved puzzle on the format analogous to that described above (so that the problem and the solution can be easily compared). The program should take (at least) two arguments, namely first the problem instance and then the satisfying assignment. If no further command line arguments are given, the program should write the solved puzzle on standard output.

How good is MiniSAT at solving Kakuro, and how much time does it take when the solver is successful? Can you find a solvable or unsolvable Kakuro instance that MiniSAT cannot handle? How large does the instance have to be (note that here it is required that the hardness should not be the result of an obviously inefficient CNF encoding, but should in some intuitive sense be intrinsic to the instance). What happens for an overconstrained instance (e.g., if you look at a solution and change the original problem in a way that is in conflict with this solution)? In case you were considering different encoding options above, do they seem to make any difference in practice? (You do not need to answer this final question in order to get full credit.)

To get you started, the directory `www.csc.kth.se/~jakobn/teaching/proofcplx11/files` contains four files `kakuro1.txt` to `kakuro4.txt` in the format described above that you can experiment with.