Abstract. This report aims to describe our project of solving KenKen puzzles with various SMT solvers. The basics of our project, a KenKen puzzle and the phases guiding its development are explained in §1. In §2 a brief introduction to SMT-LIB is given, as well as some of the problems encountered with the changeover to the new standard. §3 explains the C++ code that we developed in order to read a markup KenKen puzzle and output a suitable SMT-LIB specification for a solver to read. Examples of code and a markup puzzle are given. In §4 we explain our motivation for developing a GUI, its behavior and some of the pitfalls we encountered there as well. We conclude the report in §5. Our website contains all of the files associated with our project.

1 Introduction: KenKen SMT

In a nutshell, the purpose of our project was to solve KenKen puzzles with an SMT solver, in particular the solver Z3 developed by Microsoft. In case the reader is unfamiliar with what a KenKen puzzle is, the basic idea is this: one is given a square, usually 6x6, which has to be filled in with numbers (1-6), none of which can be the same. So far we basically have a sudoku puzzle, but KenKen ‘ups the ante’ by adding operators in sections called ‘cages’, which are marked up by bold lines. The operators are a further constraint imposed on the prospective puzzler by requiring them to make the numbers inside the cage equal the number to the left of the operator. So if we have a cage of 4 boxes, whose operator is 10+, then all of the numbers in 4 must add up to 10.

We accomplished this by taking a KenKen puzzle which was written in a markup language devised by ourselves and then writing a C++ program which could parse this language. The C++ program would then output a formal specification of that puzzle in the SMT-LIB standard. Afterwards, this specification in SMT-LIB is what a typical SMT solver is capable of reading and producing a solution to, and thus we could ‘ask Z3’ for a solution.

As will be seen, there were some interesting developments and as well as some problems along the way. We were however able to complete our goal and generate puzzles that are solvable by (at least) Z3.

1 Their website is here: http://research.microsoft.com/en-us/um/redmond/projects/z3/
Our project was organized into three phases. They are as follows:

(1) Initial Phase:
- Research SMT-LIB standards and language for implementation
- Understand C++ source code which takes a text file representing a particular sudoku puzzle, and then passes it to an SMT solver called Z3

(2) Intermediate Phase:
- Specify the axioms or rules of a similar logical puzzle-game called KenKen according to SMT-Lib Standards, and
- Write up a suitable source code which is capable of passing this specification to Z3.

(3) Final Phase:
- Polish up initial program with better terminal commands
- Test C++ output with additional solvers
- Add different output types (i.e., linear arithmetic, Z3 native format) for better performance
- Build GUI

The rest of our report has been divided into three different sections (§2-4), where a particularly important aspect of the project will be gone over in more detail. There is also a concluding section. Each of these sections is meant to reflect the nature of our project, as well as an area that a particular group member focused on. It should not be thought however that each section was investigated solely by one particular group member to the exclusion of the others; quite to contrary and true to the spirit of a group project, we all had a hand in developing every part of the project. Table 2 provides an approximate breakdown of the work done by each member.

Moreover, while the structure of the sections does not necessarily map onto the actual temporal progression of our project (as illustrated by the above phases), we feel that they represent the most important and salient areas of
research that were encountered and investigated in our work. Thus they serve as good starting points to investigate aspects of the project, albeit from different perspectives. Together they are sufficient for a complete description of the what, how and why of our project.

Finally, it is our hope that the reader enjoys this report, which is the fruit of our labours these last few months, as much as we did developing the tools and sweat necessary to produce it.

2 SMT-LIB

In this section, we will discuss SMT-LIB; what it is, why a changeover proved to be problematic, and other aspects of the project relating to satisfiability modulo theories (SMT).

2.1 SMT

SMT is a decision problem for determining whether a given expression is satisfiable with respect to some theory. Relativizing the problem in this manner is what gives us the ‘modulo’ in SMT. Theory is used in the technical, logical sense, in that it is a set of assumptions expressed in a first-order logic, and meant to capture a particular mathematical theory or system.

The two main theories that we worked in when generating our KenKen puzzle were quantifier free linear arithmetic and quantifier free bit vectors, written QF\_LIA and QF\_BV respectively. We have provided examples of the differences between these two theories in Appendix A.

2.2 SMT-LIB and Changeover

The SMT-LIB standard is “is an international effort, supported by several research groups worldwide, with the two-fold goal of producing an extensive on-line library of benchmarks and promoting the adoption of common languages and interfaces for SMT solvers.” [1 pg. 3]
Very recently the SMT-LIB standard changed from v1.2 to v2. While the update made the language much more elegant and added many excellent features it was sufficiently different enough to render most of the existing solvers incapable of parsing anything specified in v2. Furthermore, it seems that most solvers, with the exception of CVC3 and Z3 have yet to be made compliant with the new standard. As a result, it was quite difficult for us to test our KenKen puzzles on different solvers.

Tables 3 and 4 illustrate the differences between version v1.2 and v2 of the SMT-LIB standard.

\begin{verbatim}
(benchmark data
  :extrafuns ((a int) (b int) (c int))
  :formula (and (= (a) (b))
              (= (c) (b))
              (not (= (a) (c))))
)
\end{verbatim}

Table 3: SMT-LIB v1.2

\begin{verbatim}
(declare-fun a () Int)
(declare-fun b () Int)
(declare-fun c () Int)
(assert (and (= a b) (= c b) (not (= a c))))
\end{verbatim}

Table 4: SMT-LIB v2

Nevertheless, because it did not seem sensible for us to work with an older version of the standard, we made sure our KenKen puzzle (specified in either linear arithmetic or the theory of bit vectors) was in v2.

2.3 SMT Solvers

In our project, we worked with two different command-line solvers: Z3, developed by Microsoft, and CVC3, an SMT solver made available by Clark Barrett, one of the co-authors of the SMT standard.

Initially, when we first began work on our C++ code we were only testing our output against Z3. Unfortunately, we discovered that a certain class of expressions were not actually apart of the SMT-LIB v2 standard, but rather were an extension added to Z3’s input language.

For example, Z3 allows one to declare multiple functions on one line like the following:

\begin{verbatim}
(declare-funs ((x Int) (y Int) (z Int)))
\end{verbatim}

The most interesting of which is that every expression of SMT-LIBv2 is a valid s-expression in Common Lisp.
However, this is not technically apart of the SMT-LIB standard, and is not readable by CVC3, for example. In order to express the above in a manner compliant with the standard, one must write:

(declare-fun x () Int)  
(declare-fun y () Int)  
(declare-fun z () Int)

There were similar pitfalls with other extended notations being used in Z3 not compliant with SMT-LIB v2 and required us some re-write our C++ code to account for them.

Another problem we encountered, and for which no solution seems to be immediately forthcoming was an issue with the operator “div”, which is division on the integers. CVC3 does not support this operation, for the following reason:

“Unfortunately, neither CVC3 nor CVC4 support integer division. While it is specified as part of the SMT-LIB v2 standard as part of the integer theory, if you look at the actual logics for which we have benchmarks, none of these use integer division. Integer division is not easy to support because it is a non-linear integer operator which belongs to an undecidable theory. So what we usually do is specify it axiomatically (using quantifiers) and hope for the best. It works for some applications.”

While CVC3 does support real number division (“/”), when tested with a KenKen puzzle that required division in one of the cages and replaced with real number division, CVC3 returned the value “unknown”. Thus replacing integer division with “/” was not an acceptable substitute, and the puzzles that we tested with CVC3 unfortunately had to remain “div” free when working in linear arithmetic.

Of course, this problem does not arise when a puzzle is specified in the theory of bit vectors. As a result, we were able to test puzzles that used division in a cage somewhere with CVC3 by making sure it was specified with bit vectors.

However, specifying division with bit vectors has another problem as the bit vector division does machine division, i.e., it does not take into account remainders.

For example, 8/3 returns 2. This was rectified by adding in the additional constraint that the mod of the operands returned 0. Division, like subtraction, is not commutative and so the permutations had to be made into a disjunction in the assertions of a KenKen puzzle specification.

Z3 on the other hand, does support the operator “div”, but how exactly it accomplishes that feat in light of the above quotation is still unclear to us. It is possible that some workaround was devised in underground labs at Microsoft, or perhaps it is importing a different theory’s “division” operator instead. The latter option is perhaps more likely…

Lastly, as was expected, CVC3 was vastly outperformed by Microsoft’s Z3 — whether in linear arithmetic or bit vectors. Unfortunately, it was hard to

3 Personal communication from Prof. Clark Barrett.
determine the exact performance of CVC3 because the command line option that prints internal statistics was not functioning at the time of this writing.

In addition to this, CVC3’s command line option to print a SAT model was not working either, whereas Z3’s was, with one caveat. In order for Z3 to print a model, one needs to include the line (get-info model) in the specification, which is not SMT-LIB v2 compliant, but rather apart of Z3’s extended language.

Output from Z3’s “-st” command-line option is given in Appendix [4].

3 C++ Code

This section will address some of the issues involved in creating our C++ code, which took a KenKen puzzle and transformed it into something an SMT solver could work with.

3.1 Initial Foray

The first phase of project was a Z3 SMT implementation done in C++. Some of our reasons for doing so are as follows. First, the Z3 website[5] already included C++ code for a Sudoku solver. This solver took an input file that represented a Sudoku puzzle and transformed it into a format that could be inputted into the online Z3 engine. We were thus able to study the code and understand how they did it — no sense in re-inventing the wheel if not necessary. Table 5 is an example of the online code’s markup language for a sudoku puzzle.

<table>
<thead>
<tr>
<th>Table 5: Online Markup Language for Sudoku</th>
</tr>
</thead>
<tbody>
<tr>
<td>..2..1.6.</td>
</tr>
<tr>
<td>..7..4..</td>
</tr>
<tr>
<td>5.....9..</td>
</tr>
<tr>
<td>.1.3.....</td>
</tr>
<tr>
<td>8...5..4</td>
</tr>
<tr>
<td>....6.2.</td>
</tr>
<tr>
<td>..6.....7</td>
</tr>
<tr>
<td>...8..3..</td>
</tr>
<tr>
<td>.4.9..2..</td>
</tr>
</tbody>
</table>

Unfortunately, a couple of problems with the online sudoku solver were made immediately evident. First was that the code would not even compile on a Unix machine. This turned out to be a combination of problems related to the code being written specifically for windows. The second problem was that the code that was outputted for the Z3 engine wouldn’t run on the online Z3 engine. Moreover, in the Sudoku solver code the author comments that state that he

does not fully understand what one section of the code does and that it was included because another Sudoku solver also included the code.

However, even though we struggled with the online code, our efforts did yield some useful results. All of the problems listed above gave us our first hints into the major differences between SMT-LIB v1.2 and SMT-LIB v2. We also thereby learned that the online Z3 was not backwards compatible with the old standard.

Moreover, the online solver also allowed a user to specify which SMT theory the solver would use: Z3 native, linear arithmetic, and bit vectors. The author had also posted benchmarks of how each of the theories performed. These ideas were implemented into the first version of our C++ KenKen Solver.

3.2 1st Version

The first version of the KenKen solver worked in many ways just like the online Sudoku solver. And just like the online solver, we needed to create a custom mark-up language to represent a KenKen puzzle. Our C++ code would parse the KenKen puzzle and then output a file that could be cut and pasted into the online Z3 engine. Table 6 is a snippet of our markup language, which shows how we represented KenKen cages.

<table>
<thead>
<tr>
<th>size</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>begin cage</td>
<td>5+</td>
</tr>
<tr>
<td>r1c1</td>
<td>r1c2</td>
</tr>
<tr>
<td>end cage</td>
<td></td>
</tr>
<tr>
<td>begin cage</td>
<td>1=</td>
</tr>
<tr>
<td>r1c3</td>
<td></td>
</tr>
<tr>
<td>end cage</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: KenKen Markup Language

Having completed the basics, we then expanded the KenKen solver to include features such as choosing the input and output file as well as the theory to be used. At this point we were mostly working on how to correctly represent the different SMT theories.

After implementing our first C++ KenKen solver it became evident that we did not have a firm understanding of how the different SMT theories should be represented. This is also where we discovered the reason why the online sudoku solver hadn’t been working. The online solver had been written in SMT-LIB 1 and Z3 had since been updated to an extended version of SMT-LIB v2.
As a result of these discoveries, it became evident that we would have to redo the parts of our C++ code that generated SMT-LIB compliant output. This required further research to be done on the standard, and here it was particularly useful to be able to use an independent solver (in our case, CVC3) with which to gauge the standard.

3.3 2nd Version

For the second version of our C++ code we decided to abandon the extended language that Z3 supported, even though this language was much more convenient and easier to use. This was done for three reasons. First, the Z3 extended language was not SMT-LIB v2 compliant, which prevented us from benchmarking (or even just running) the output on different SMT engines. Second, it wasn’t clear whether the extended language represented a particular SMT theory, or whether was just a simpler way to write specifications for the Z3 engine, perhaps independent of the standard. We are still unsure of the exact status of the extended language in this respect. Thirdly, we wanted our code to be unlike the online sudoku code in that future solvers could make use of it since it would be written in the latest standard — and to avoid potential future graduate students from becoming frustrated at our lack of foresight when they tried to learn from our code.

Thus version 2 of the C++ code now included the ability to solve a puzzle using either linear arithmetic or the theory of bit vectors. Furthermore, the code that it outputted was SMT-LIB v2 compliant, which allowed us to run the code on Z3 and CVC3.

3.4 Conclusion

To conclude, our C++ KenKen ‘solver’ now supports bit vectors, linear arithmetic and outputs SMT-LIB v2 compliant KenKen puzzle specifications that can run and be solved by different SMT engines (currently only tested with Z3 and CVC3). Unfortunately, as mentioned in §2 other solvers are few and far between that work with the latest SMT-LIB standard as the transition between v1.2 and 2 was fairly catastrophic to the working pool of SMT solvers.

However, we suspect that the day is not far away when all solvers will be created equal. Therefore, by sticking to the latest standard our code will be ready and prepared for that bright future.

4 C# GUI

Although the mark up language for representing KenKen puzzles was fairly simple, we determined that a graphical interface would provide a more intuitive way to both create puzzles and view the solutions. This application could also incorporate the previously written C++ code to allow output of SMT-LIB v2 files that would provide solutions to the puzzles.
Another motivation to make a GUI was to experiment with the managed dll for Z3 that is provided with the solver. Because of this, it was decided that the GUI would be programmed in the C# programming language and would use Z3’s API to internally solve the puzzle.

4.1 Implementation

Because of the nature of the C# programming language, object oriented programming played a big part in the implementation. A KenKen cell class was created that contained a graphical button, a name, an operator, an argument and several Boolean flags. The KenKen puzzle cages were simply represented as a list of KenKen cells and all the cages were represented as a list of cages. All of the KenKen cells were stored in a two dimensional array whose indexes corresponded to where they were on the puzzle. While this required some “up front” work, it made every other aspect of implementing the GUI much simpler.

Graphically the KenKen cells were represented with the Windows Form Button objects. Cells that are in a cage are the same color and their argument and operator are displayed on the top, right-most cell (see the Appendix C for screen shots). Users can use the top menu to save, quit, and solve the puzzle along with setting various options. The bottom two buttons allow the user to create cages and clear the puzzle. In short, the interface was made to be as easy and intuitive as possible.

4.2 Managed API

Microsoft’s Z3 engine comes with a managed dll which provides an API for internally using Z3 inside of .net applications. It was hoped that the API would be able to simply parse an SMT-LIB v2 file and solve it. If that were the case, then the GUI application could simply create an SMT-LIB v2 file with the C++ code and then have Z3 parse it and solve it internally. Unfortunately, while the API does support parsing files, it has not yet been updated to support the SMT-LIB v2 syntax. Because of this, the logic of the C++ code had to be completely re-implemented in C# using the API’s term and context objects. The example posted on Microsoft’s webpage provides a good illustration of how to use these objects correctly. This provided the oppportunity to work more in depth with the Z3 API and was a good learning experience.

In short, the GUI application provides an intuitive and quick way of creating KenKen puzzles that can be solved with SMT solvers. This was also able to make debugging easier and we uncovered several subtle bugs with our approach. Because of the API’s inability to parse SMT-LIB v2 files, we were able to explore the actual objects provided with Microsoft’s managed dll and utilize them so solve a puzzle internally.

5 Final Remarks

Throughout the course of this report we have seen various aspects of our project highlighted — from research into the SMT-LIB standard and the use of different solvers, to our implementation of that standard in our C++ code, to a GUI for creating puzzles and having them solved with the internal Z3 APIs.

That being said, the code base that we developed is easily extensible, and by way of conclusion, a brief discussion of some potential future additions will be given. Unfortunately, due to time constraints we were not able to pursue them, but nevertheless we did give thought to their possible implementation.

5.1 Solving for Unknown Operators

The matter was brought up concerning solving for unknown operators in a KenKen puzzle. This might look something like \( ?20 \), where 20 is the result of an unspecified operation. While this technically never occurs in a KenKen puzzle, it would be an interesting problem for an SMT solver to work with, and is an example of an potential addition to our code.

Z3 offers the ability to solve for unknown functions, for example in the following:

```
(declare-funs ((x Int) (y Int) (z Int)))
(assert (>= (* 2 x) (+ y z)))
(declare-funs ((f Int Int) (g Int Int Int)))
(assert (< (f x) (g x x)))
(assert (> (f y) (g x x)))
(check-sat)
(get-info model)
(quit)
```

A large amount of constraints would have to be applied to the ‘unknown’ functions here specified, since a KenKen puzzle can only have 4 possible operators. In other words, the function that Z3 returns might not correspond at all to one of those 4 operators, and thus a substantial amount of assertion/constraints would have to be implemented to correctly model the acceptable operators.

Another workaround for solving for unknown operators could involve disjoining the possible operators in a KenKen puzzle specification and having Z3 generate a model where one of those operators is selected, which would then yield a satisfiable solution.

Nevertheless, the problem does not seem in principle intractable, but again, due to time constraints we were unable to fully incorporate this addition to our project.

5.2 Modulo Different Theories

Another interesting addition would have been expressing a KenKen puzzle in different theories of the SMT-LIB, and comparing the solution times between
them. Seeing that the solution times between linear arithmetic and the theory of bit vectors was surprisingly different, and that this was one of the more interesting discoveries in our project, we wondered how this might translate to different specifications in different theories.

For example, as Greg Szubzda pointed out in an email, linear arithmetic might be too expressive for our purposes, and thus accounts for the increased solution time. While it seems unlikely that a theory might be faster than solving in bit vectors, specifications in other theories between the solution time for linear arithmetic and bit vectors might be found.

It would be therefore be interesting to note the differences between these theories, and perhaps account for the differences in solution time.

To sum up the lessons from our project, we learned that SMT is a versatile and powerful tool in the computer sciences, and were glad for the opportunity to investigate and explore these matters in an exciting and engaging setting.

References


A Example of SMT-LIB Theory Differences

Table 7 and 8 are examples of SMT-LIB v2 compliant specifications of a fragment of the same KenKen puzzle (outputted by our C++ code of course). As we later learned, the solving time for a problem in one theory may be drastically reduced (or increased) by changing to a different theory.
(assert (or (= 5 (- r2c1 r2c2 )) (= 5 (- r2c2 r2c1 ))))
(assume (or (= 1 (- r1c3 r2c3 )) (= 1 (- r2c3 r1c3 ))))
(assume (= 6 (* r1c4 r2c4)))
(assume (or (= 2 (div r1c5 r2c5 )) (= 2 (div r2c5 r1c5 ))))
(assume (= 2 r1c6))
(assume (= 17 (+ r2c6 (+ r3c6 (+ r3c5 r4c5)))))
(assume (= 10 (+ r3c1 (+ r3c2 r4c2))))
(assume (or (= 5 (- r3c3 r4c3 )) (= 5 (- r4c3 r3c3 ))))
(assume (or (= 2 (- r3c4 r4c4 )) (= 2 (- r4c4 r3c4 ))))
(assume (= 40 (* r4c1 (* r5c1 r5c2))))
(assume (= 24 (* r4c6 (* r5c6 r6c6))))
(assume (or (= 2 (div r6c1 r6c2 )) (= 2 (div r6c2 r6c1 ))))
(assume (= 6 (* r5c3 r6c3)))
(assume (= 6 (+ r5c4 r6c4)))
(assume (or (= 1 (- r5c5 r6c5 )) (= 1 (- r6c5 r5c5 ))))

Table 7: QF_LIA

(assume (= (_ bv5 4) (bvadd r1c1 r1c2)))
(assume (or (= (_ bv5 4) (bvsub r2c1 r2c2 )) (= (_ bv5 4) (bvsub r2c2 r2c1 ))))
(assume (or (= (_ bv1 4) (bvsub r1c3 r2c3 )) (= (_ bv1 4) (bvsub r2c3 r1c3 ))))
(assume (= (_ bv6 4) (bvmul r1c4 r2c4)))
(assume (or (and (= (_ bv2 4) (bvdiv r1c5 r2c5 )) (= (_ bv0 4) (bvurem r1c5 r2c5))
             (and (= (_ bv2 4) (bvdiv r2c5 r1c5 )) (= (_ bv0 4) (bvurem r2c5 r1c5)))))
(assume (= (_ bv2 4) r1c6))
(assume (= (_ bv17 4) (bvadd r2c6 (bvadd r3c6 (bvadd r3c5 r4c5))))
(assume (= (_ bv10 4) (bvadd r3c1 (bvadd r3c2 r4c2))))
(assume (or (= (_ bv5 4) (bvsub r3c3 r4c3 )) (= (_ bv5 4) (bvsub r4c3 r3c3 ))))
(assume (or (= (_ bv2 4) (bvsub r3c4 r4c4 )) (= (_ bv2 4) (bvsub r4c4 r3c4 ))))
(assume (= (_ bv40 4) (bvmul r4c1 (bvmul r5c1 r5c2))))
(assume (= (_ bv24 4) (bvmul r4c6 (bvmul r5c6 r6c6))))
(assume (or (and (= (_ bv2 4) (bvdiv r6c1 r6c2 )) (= (_ bv0 4) (bvurem r6c1 r6c2))
             (and (= (_ bv2 4) (bvdiv r6c2 r6c1 )) (= (_ bv0 4) (bvurem r6c2 r6c1)))))
(assume (= (_ bv6 4) (bvmul r5c3 r6c3)))
(assume (= (_ bv6 4) (bvadd r5c4 r6c4)))
(assume (or (= (_ bv1 4) (bvsub r5c5 r6c5 )) (= (_ bv1 4) (bvsub r6c5 r5c5 ))))

Table 8: QF_BV

B Z3 Command-Line Performance

This is the output generated by using Z3’s command line option “-st”. Z3 ran on full versions of the 6x6 puzzles seen in Appendix A. Notice the time difference for solutions between Table 9 and Table 10 (the theory of bit vectors and linear arithmetic, respectively) — greater than a factor of 100!
<table>
<thead>
<tr>
<th></th>
<th>sat</th>
<th>QF_BV</th>
</tr>
</thead>
<tbody>
<tr>
<td>num. conflicts</td>
<td>193</td>
<td>657</td>
</tr>
<tr>
<td>num. decisions</td>
<td>849</td>
<td>2108</td>
</tr>
<tr>
<td>num. propagations</td>
<td>14262 (binary: 2400)</td>
<td>24140 (binary: 13520)</td>
</tr>
<tr>
<td>num. restarts</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>num. final checks</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>num. min. lits</td>
<td>272</td>
<td>488</td>
</tr>
<tr>
<td>num. added eqs.</td>
<td>3019</td>
<td>20</td>
</tr>
<tr>
<td>num. mk bool var</td>
<td>2097</td>
<td>393</td>
</tr>
<tr>
<td>num. del bool var</td>
<td>297</td>
<td>113</td>
</tr>
<tr>
<td>num. mk enode</td>
<td>438</td>
<td>1907</td>
</tr>
<tr>
<td>num. del enode</td>
<td>66</td>
<td>54</td>
</tr>
<tr>
<td>num. mk clause</td>
<td>755</td>
<td>520</td>
</tr>
<tr>
<td>num. del clause</td>
<td>700</td>
<td>700</td>
</tr>
<tr>
<td>num. mk bin clause</td>
<td>17499</td>
<td>520</td>
</tr>
<tr>
<td>num. mk lits</td>
<td>148</td>
<td>13919</td>
</tr>
<tr>
<td>num. bv conflicts</td>
<td>16</td>
<td>575</td>
</tr>
<tr>
<td>num. bv diseq (d)</td>
<td>451</td>
<td>20</td>
</tr>
<tr>
<td>num. bv bit2core</td>
<td>499</td>
<td>20</td>
</tr>
<tr>
<td>num. bv-&gt;core eq</td>
<td>743</td>
<td>20</td>
</tr>
<tr>
<td>max. heap size</td>
<td>1.65556 Mbytes</td>
<td>1.93407 Mbytes</td>
</tr>
<tr>
<td>time</td>
<td>0.02 secs</td>
<td>0.28 secs</td>
</tr>
</tbody>
</table>

Table 9: Statistics for QF_BV

<table>
<thead>
<tr>
<th></th>
<th>sat</th>
<th>QF_LIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>num. conflicts</td>
<td>657</td>
<td>575</td>
</tr>
<tr>
<td>num. decisions</td>
<td>2108</td>
<td>2630</td>
</tr>
<tr>
<td>num. propagations</td>
<td>24140 (binary: 13520)</td>
<td>163494</td>
</tr>
<tr>
<td>num. restarts</td>
<td>6</td>
<td>4630</td>
</tr>
<tr>
<td>num. final checks</td>
<td>4</td>
<td>13315</td>
</tr>
<tr>
<td>num. min. lits</td>
<td>488</td>
<td>12460</td>
</tr>
<tr>
<td>num. added eqs.</td>
<td>20</td>
<td>13315</td>
</tr>
<tr>
<td>num. mk bool var</td>
<td>571</td>
<td>1255</td>
</tr>
<tr>
<td>num. del bool var</td>
<td>16</td>
<td>96</td>
</tr>
<tr>
<td>num. mk enode</td>
<td>393</td>
<td>121</td>
</tr>
<tr>
<td>num. del enode</td>
<td>113</td>
<td>121</td>
</tr>
<tr>
<td>num. mk clause</td>
<td>1907</td>
<td>9</td>
</tr>
<tr>
<td>num. del clause</td>
<td>54</td>
<td>eq. adapter: 19</td>
</tr>
<tr>
<td>num. mk bin clause</td>
<td>520</td>
<td>9</td>
</tr>
<tr>
<td>num. mk lits</td>
<td>13919</td>
<td>eq. adapter: 19</td>
</tr>
<tr>
<td>num. ta conflicts</td>
<td>575</td>
<td>1255</td>
</tr>
<tr>
<td>num. add rows</td>
<td>163494</td>
<td>96</td>
</tr>
<tr>
<td>num. pivots</td>
<td>4630</td>
<td>1255</td>
</tr>
<tr>
<td>num. assert lower</td>
<td>13315</td>
<td>96</td>
</tr>
<tr>
<td>num. assert upper</td>
<td>12460</td>
<td>1255</td>
</tr>
<tr>
<td>num. assert diseq</td>
<td>15</td>
<td>96</td>
</tr>
<tr>
<td>num. bound prop</td>
<td>1255</td>
<td>121</td>
</tr>
<tr>
<td>num. max-min</td>
<td>96</td>
<td>121</td>
</tr>
<tr>
<td>num. pseudo nl.</td>
<td>121</td>
<td>9</td>
</tr>
<tr>
<td>num. nl. bounds</td>
<td>9</td>
<td>eq. adapter: 19</td>
</tr>
<tr>
<td>max. heap size</td>
<td>1.93407 Mbytes</td>
<td>0.28 secs</td>
</tr>
<tr>
<td>time</td>
<td>0.28 secs</td>
<td>0.28 secs</td>
</tr>
</tbody>
</table>

Table 10: Statistics for QF_LIA
C GUI Screenshots

Examples of our GUI with a puzzle and the solution using the theory of bit vectors.

Fig. 1: GUI KenKen Puzzle
Fig. 2: GUI KenKen Solution