CSC 339

Fall 2016

Programming Assignment 4

**Due: Friday, October 28 at the beginning of class (10:20 AM)**

# Hexadecimal Sudoku Solver and Creator

In this assignment you will use a **SAT solver** to find solutions to **Hexadecimal Sudoku puzzles**. Instead of explicitly searching for a solution (e.g., through DFS), **you will express the rules of the puzzle as logical assertions**. You will encode these rules in CNF and send them to a solver. If the solver finds a solution to the corresponding Boolean satisfiability problem, then you will do a reverse mapping to obtain the corresponding solution to the puzzle.

A Sudoku puzzle is usually considered to be poorly constructed if it has **no solution** or **more than one solution**. Your solver should also be able to report if no solution exists and to confirm that a solution, if it exists, is **unique**.

Finally, as an optional bonus exercise, you can write additional code to use your solver to **generate** Hexadecimal Sudoku puzzles that human beings can play. These may be difficult (and require some guessing), but they should be “well constructed” in the sense that they admit just one solution.

As usual, you will submit your complete code **along with a detailed write-up** explaining the reduction to SAT, how your code works (design choices, data structures, and similar implementation details), and what you learned from the assignment.

## Hexadecimal Sudoku

Here is an example puzzle:

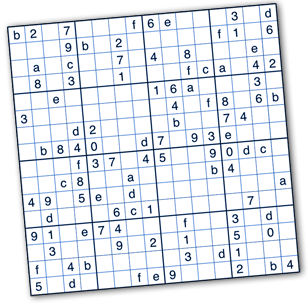


Figure 1. Hexadecimal Sudoku Puzzle (source: <http://krazydad.com/hexsudoku/)>.

The objective is to fill in the remaining spaces in the 16x16 grid, subject to constraints:

* There must be exactly one hexadecimal digit (1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f) in each row.
* There must be exactly one hex digit in each column.
* There must be exactly one hex digit in each 4x4 box (as indicated by the darker boundaries).

Note that “just one” is technical mathematical language: It means “exactly one,” that is, “one and only one.” Also, note that there are **no** constraints on the diagonals. For example, you can see that there are already two 3s on the diagonal that descends from the top left to the lower right corner. There is a variant of Sudoku, so-called Latin Squares Sudoku, that has such a constraint—but that does not apply to the problems we are solving for this assignment.

## What Your Solver Should Do

Input to your solver will be in the form of simple text files corresponding to Hex Sudoku puzzles. Each file will consist of the numerals 0-9, letters a-z, the \_ character (“blank”), and whitespace. Whitespace (except perhaps for some line breaks) can be ignored. There will be a total of 16 non-blank rows, each with 16 non-whitespace characters (hex digits or blanks). For example, the input to the sample problem in Fig. 1 looks like this:

b 2 \_ 7 \_ \_ \_ f 6 e \_ \_ \_ 3 \_ d

\_ \_ \_ 9 b \_ 2 \_ \_ \_ \_ \_ f 1 \_ 6

\_ a \_ c \_ \_ 7 \_ 4 \_ 8 \_ \_ \_ e \_

\_ 8 \_ 3 \_ \_ 1 \_ \_ \_ f c a \_ 4 2

\_ \_ e \_ \_ \_ \_ \_ 1 6 a \_ \_ \_ 3 \_

3 \_ \_ \_ \_ \_ \_ \_ \_ 4 \_ f 8 \_ 6 b

\_ \_ \_ d 2 \_ \_ \_ \_ b \_ \_ 7 4 \_ \_

\_ b 8 4 0 \_ \_ d 7 \_ 9 3 e \_ \_ \_

\_ \_ \_ f 3 7 \_ 4 5 \_ \_ 9 0 d c \_

\_ \_ c 8 \_ \_ a \_ \_ \_ \_ b 4 \_ \_ \_

4 9 \_ 5 e \_ d \_ \_ \_ \_ \_ \_ \_ \_ a

\_ d \_ \_ \_ 6 c 1 \_ \_ \_ \_ \_ 7 \_ \_

9 1 \_ e 7 4 \_ \_ \_ f \_ \_ 3 \_ d \_

\_ 3 \_ \_ \_ 9 \_ 2 \_ 1 \_ \_ 5 \_ 0 \_

f \_ 4 b \_ \_ \_ \_ \_ 3 \_ d 1 \_ \_ \_

5 \_ d \_ \_ \_ f e 9 \_ \_ \_ 2 \_ b 4

The solution, if one exists, should be “pretty-printed” in the same form as the input. For the problem above:

b 2 5 7 4 a 9 f 6 e 0 1 c 3 8 d

e 4 0 9 b 8 2 c d 5 3 a f 1 7 6

1 a f c 6 d 7 3 4 9 8 2 b 5 e 0

d 8 6 3 5 e 1 0 b 7 f c a 9 4 2

0 7 e 2 8 b 4 9 1 6 a 5 d c 3 f

3 5 9 1 a c e 7 2 4 d f 8 0 6 b

c f a d 2 1 3 6 8 b e 0 7 4 5 9

6 b 8 4 0 f 5 d 7 c 9 3 e 2 a 1

2 e 1 f 3 7 b 4 5 a 6 9 0 d c 8

7 6 c 8 9 2 a 5 0 d 1 b 4 e f 3

4 9 3 5 e 0 d 8 f 2 c 7 6 b 1 a

a d b 0 f 6 c 1 3 8 4 e 9 7 2 5

9 1 2 e 7 4 0 b a f 5 6 3 8 d c

8 3 7 a d 9 6 2 c 1 b 4 5 f 0 e

f 0 4 b c 5 8 a e 3 2 d 1 6 9 7

5 c d 6 1 3 f e 9 0 7 8 2 a b 4

If **no solution** exists, your program should report it as follows:

Sorry, no solution

**If a solution does exist, your program should check for at least one additional solution.** (You can stop after showing that there is another solution. You should NOT try to find them all. Or at the very least, please do not paste lots of solutions into your write-up!!) If one is found, your program should display it. For the problem above, there is only one solution, so the solver should report:

Solution is unique

However, if the puzzle did not include the 7 in row 1, column 3, then the program would report additional solutions. For example:

b 2 5 1 4 a 9 f 6 e 7 0 c 3 8 d

e 4 0 9 b c 2 8 3 5 d a f 1 7 6

6 a f c 5 d 7 3 4 2 8 1 9 b e 0

d 8 7 3 6 e 1 0 b 9 f c a 5 4 2

c f e 0 8 b 4 7 1 6 a 2 d 9 3 5

3 7 9 2 c 1 e a d 4 5 f 8 0 6 b

1 5 6 d 2 f 3 9 8 b 0 e 7 4 a c

a b 8 4 0 5 6 d 7 c 9 3 e f 2 1

2 6 b f 3 7 8 4 5 a 1 9 0 d c e

7 e c 8 9 0 a 5 f d 6 b 4 2 1 3

4 9 1 5 e 2 d b c 0 3 7 6 8 f a

0 d 3 a f 6 c 1 2 8 e 4 b 7 5 9

9 1 2 e 7 4 0 c a f b 5 3 6 d 8

8 3 a 7 d 9 b 2 e 1 4 6 5 c 0 f

f c 4 b a 8 5 6 0 3 2 d 1 e 9 7

5 0 d 6 1 3 f e 9 7 c 8 2 a b 4

(There may be others; I did not check further.)

If you would prefer to output the solution as formatted text—for example, using LaTeX or a GUI—that is also OK. Just be sure to discuss this in your write-up.

## Programming (Reduction to SAT)

Rather than apply one of the direct search methods that we have learned in this class (such as DFS or simulated annealing), your program will instead **reduce** the problem (transform it) to a corresponding Boolean Satisfiability (SAT) problem and then rely on the heuristics used by the SAT solver. You have already written SAT solvers, so you have some sense of how these work “under the hood.” However, for this assignment, we will mostly be treating the SAT solver as a “black box.”

For this part of the assignment, you will need to come up with a suitable reduction that represents the rules of Hexadecimal Sudoku. First, recall that we need Boolean variables, which we will treat as atoms. For example:

is true if and only if there is a 7 in row 1, column 4 of the grid

Consider carefully what your variables will be and how many you will need in all to represent the problem. You will discuss this in your write-up.

Next, come up with sentential logical rules that correspond to the rules of Sudoku. For example, if there is a 7 in row 1, column 4, then there cannot also be an f in row 1, column 4 of the same grid. We could express this in symbolic logic as follows:

You will need similar rules for all columns, rows, the 4x4 boxes, etc.

You will then need to transform all of your rules into CNF. This is discussed in the textbook, and we also showed how to do this in class:

Finally, you will need to output all of the rules in DIMACS format—or in some format that your SAT solver can read. This was discussed in the last assignment, so we will not revisit it here.

Note that the “rules of the game” will be the same for **every** Hex Sudoku problem. However, the specific puzzle—the partially filled-in grid—will also need to be encoded in CNF as logical assertions. This will of course be different for each puzzle.

To solve the resulting Boolean satisfiability problem, you may use one of the solvers that you wrote for A3 or a “competition-grade” solver such as MiniSAT (<http://minisat.se>). MiniSAT is installed on the machines in the basement of Olin and is available for most major operating systems. You should be able to call it directly from your program in Python, Java, C++, etc.

Finally, you will need to parse the output from the solver and perform a reverse mapping from the solution to the SAT problem to the solution to the corresponding Hex Sudoku problem. Be sure to consider that the solver may output UNSAT as the answer. Also recall that we showed in class how to find additional SAT solutions.

Note: We are trusting the solver to do the “heavy lifting” for us. Most of the running time will be the time it takes for the SAT solver to search for a solution. Don’t spend too much time trying to write fast, clever code for your parsers or for the reduction! Focus on making sure these are correct and easily understood.

# Write-Up

Along with your code, you will submit a **word-processed write-up** that presents your work to a human reader. In the write-up, you should address the following issues:

* Explain any special terms or notation that you use in the paper. Don’t assume that we can figure out what you mean.
* How did you “frame” the problem? Discuss the semantics of your Boolean variables and the specifics of your reduction.
* Give brief instructions for how to use your code along with some examples of execution for a couple of problems.
* Give a brief overview of how your code works “under the hood.” This is the place to tell what programming language you used, which packages you used, who assisted you in any way with suggestions, and so on. What data structures did you use? Does your code still have an unresolved bug like a memory leak that occasionally causes problems?
* **Give the solutions to all of the provided problems,** if a solution exists. Check whether there are further solutions. If so, give at least one; if not, report that the solution you gave previously is unique.
* Finally, what did you learn from the assignment?

## Challenge Problem (Bonus): Hex Sudoku Puzzle Creator

Make sure the solver is working. Solve all of the problems. Complete the write-up. Edit. Proofread. Once all of this is complete, write additional code so that you can use your solver as the basis for a Hex Sudoku Puzzle Creator. Then, go back and extend your write-up to include a brief discussion of your puzzle creator.

Here is one way that you can go about this. First, generate a solved board (a completely filled-in grid that satisfies all of the rules of Hex Sudoku). You can do this by starting with an empty board and adding entries one at a time, such that the solver reports that a solution exists. When the solution is unique, you stop, fill in the solved grid with the unique solution, and return it.

**procedure** GenerateSudokuSolution

initialize empty

**loop:**

randomly select a blank in the

= **false**

**repeat:**

hex digit selected at random from

= copy of

put hex digit in the of

call solver to see if has a solution

**if** it does **then:**

= **true**

**if** the solution is also unique **then:**

use the solution to fill in the remainder of the

**return** filled-in

**end if**

**end if**

**until**

**end loop**

Then, successively remove entries from your solution to obtain a playable puzzle. Remove an entry only if the solver reports that the resulting solution will be unique. Return the puzzle (the partially filled-in grid that has a unique solution—specifically, the one you generated in the first step) once no further digits can be removed in this way.

**procedure** SudokuPuzzleCreator

GenerateSudokuSolution

the set of all not-empty spaces on

**repeat:**

randomly select a non-blank space from

remove the hex digit from space in

call solver to see if the modified still has a unique solution

if not, undo the change to the

**until**

**return**

Example execution:

Generating solution ...

0 d 2 5 b 3 f 8 7 a 9 6 c 4 1 e

1 e 4 7 a d c 5 3 0 2 b 9 f 6 8

a f 6 8 1 7 9 0 c e 4 5 b 2 d 3

9 c 3 b 6 4 e 2 d 8 1 f a 7 0 5

4 b c a 3 5 2 e 1 f 0 d 7 6 8 9

f 1 7 2 8 b 4 a 5 6 c 9 d e 3 0

e 0 9 d 7 f 6 c 8 2 a 3 1 5 b 4

6 8 5 3 d 0 1 9 e 7 b 4 f a 2 c

8 3 a e f 6 0 4 b 1 7 2 5 c 9 d

b 5 0 1 2 c a d 4 9 f 8 e 3 7 6

7 6 f 9 5 8 3 b 0 c d e 4 1 a 2

2 4 d c 9 e 7 1 6 5 3 a 8 0 f b

c a b 6 e 1 5 3 f d 8 0 2 9 4 7

5 9 8 f 4 a b 6 2 3 e 7 0 d c 1

3 7 1 4 0 2 d f 9 b 5 c 6 8 e a

d 2 e 0 c 9 8 7 a 4 6 1 3 b 5 f

Generating puzzle from solution ...

0 d \_ 5 \_ \_ f 8 7 \_ \_ \_ \_ 4 \_ \_

1 \_ \_ \_ a \_ \_ \_ \_ 0 2 b \_ \_ \_ 8

a \_ 6 8 \_ \_ 9 0 \_ \_ 4 5 b \_ d \_

\_ \_ \_ \_ \_ 4 e \_ \_ \_ \_ \_ \_ \_ \_ \_

\_ \_ \_ \_ \_ \_ \_ \_ \_ f \_ \_ \_ \_ \_ 9

\_ \_ \_ \_ 8 \_ \_ a 5 \_ \_ 9 \_ \_ \_ \_

\_ 0 \_ d 7 \_ 6 c 8 \_ \_ 3 \_ 5 \_ \_

6 \_ \_ 3 d 0 1 \_ \_ \_ \_ \_ f a 2 \_

\_ \_ a e \_ \_ \_ 4 b 1 \_ \_ \_ \_ \_ d

b 5 \_ \_ 2 \_ a \_ \_ \_ f \_ e 3 7 \_

7 \_ \_ \_ \_ 8 \_ \_ \_ \_ d \_ 4 \_ \_ 2

\_ \_ \_ \_ 9 e \_ \_ \_ 5 \_ a \_ 0 f \_

\_ \_ \_ \_ e \_ \_ \_ f \_ 8 \_ \_ 9 \_ 7

\_ 9 \_ f 4 \_ b \_ \_ 3 \_ 7 \_ d \_ 1

\_ 7 1 \_ 0 2 \_ \_ \_ b \_ \_ \_ 8 \_ \_

\_ 2 \_ \_ \_ \_ \_ \_ \_ \_ 6 \_ \_ \_ 5 \_

Puzzle written to file prob\_13.inp

You can undoubtedly think of many ways to improve upon this generation method. In particular, the resulting puzzles are likely to be too difficult for most human Sudoku fans to enjoy. Also note that, for complex reasons, the resulting generation method does not generate Sudoku puzzles *uniformly* at random. In your write-up, discuss any improvements that you made to the generation method. Also give some examples of the output.

# Deliverables:

* Your complete, commented source code [uploaded to Moodle]
* Your write-up as described above [hard copy, submitted in-person if possible]

# Grading Rubric

* Code: 60 points
  + 20 points: You submitted a complete solver implementation that will actually run
  + 20 points: Your solver performs as expected (solves all the puzzles, bug-free, etc.)
  + 10 points: Your solver correctly reports no solution and whether a solution is unique
  + 10 points: Your code is commented and structured so that I can easily follow it
* Write-up: 40 points
  + 10 points: You described your reduction in detail and explained why it must be correct (a formal proof, however, is *not* required)
  + 10 points: I can understand your implementation without looking at your code; you also didn’t overwhelm me with too many technical details in this section
  + 5 points: You told how to use your code and gave some sample output showing that it works
  + 10 points: You proof-read, used correct notation, used suitable headings, etc.
  + 5 points: You described some things you learned from the assignment
* Bonus: Up to 10 points (code and addendum to write-up) [see above]

As noted in the syllabus, late submissions are reduced by 20 points per day or part of a day. The assignment is due at the start of class unless otherwise specified in class or on Moodle.

Note: You may be asked to demonstrate your code to the instructor at a later date. (If so, I will contact you by email within one week of the submission to arrange for this.)