Basic Task

sud2sat.py

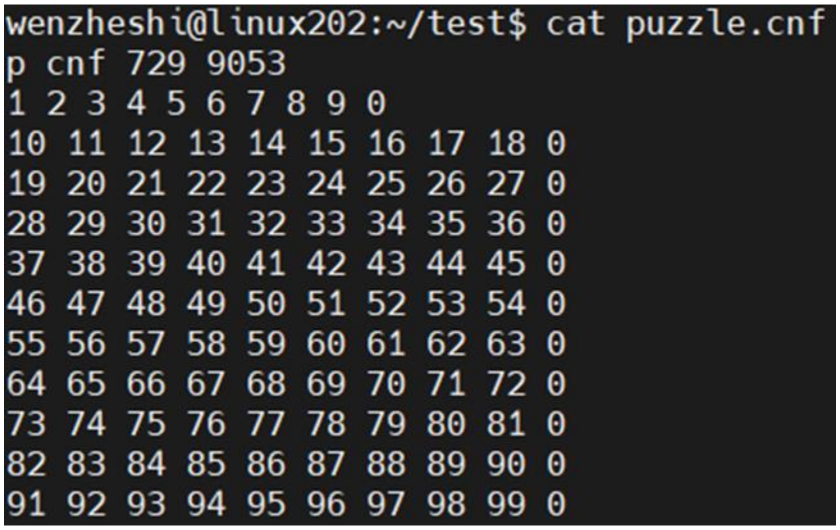
According to the description in the document, the given form of Sudoku can be either a regular matrix or a sequential string of numbers. The empty parts are filled with non-1-9 characters. We need to process the raw data first using the function *dealWithRawData*. This function removes spaces and line breaks from the input string and stores the original Sudoku in a two-dimensional array. The empty parts are uniformly filled with ".". We then wrote a function *getVariables* to assign numbers to all variables. In a 9x9 Sudoku, each cell has 9 variables, totaling 729 variables.

Next, based on the minimal encoding method, the function *minimalEncoding* converts the Sudoku rules into SAT format. The rules are as follows:

Each row represents a different variable with an OR relationship, and different rows represent an AND relationship. The following are needed for the minimal encoding:

- Each cell can only have one true variable.
- Each digit must appear at least once in each row.
- Each digit must appear at least once in each column.
- Each digit must appear at least once in each 3x3 grid.

These rule statements are added to the CNF list. Since the Sudoku itself provides the filled numbers, the variables for these positions are already determined. The corresponding variable for each number is set to true, while others are set to false. Then, they are added to the CNF list using the function *sud2sat*. Finally, the contents of the CNF list are formatted into SAT format. Each line is padded with a 0 at the end and concatenated with a line break. The number of variables and the number of statements are added at the beginning to conform to DIMACS format and the entire string is printed.



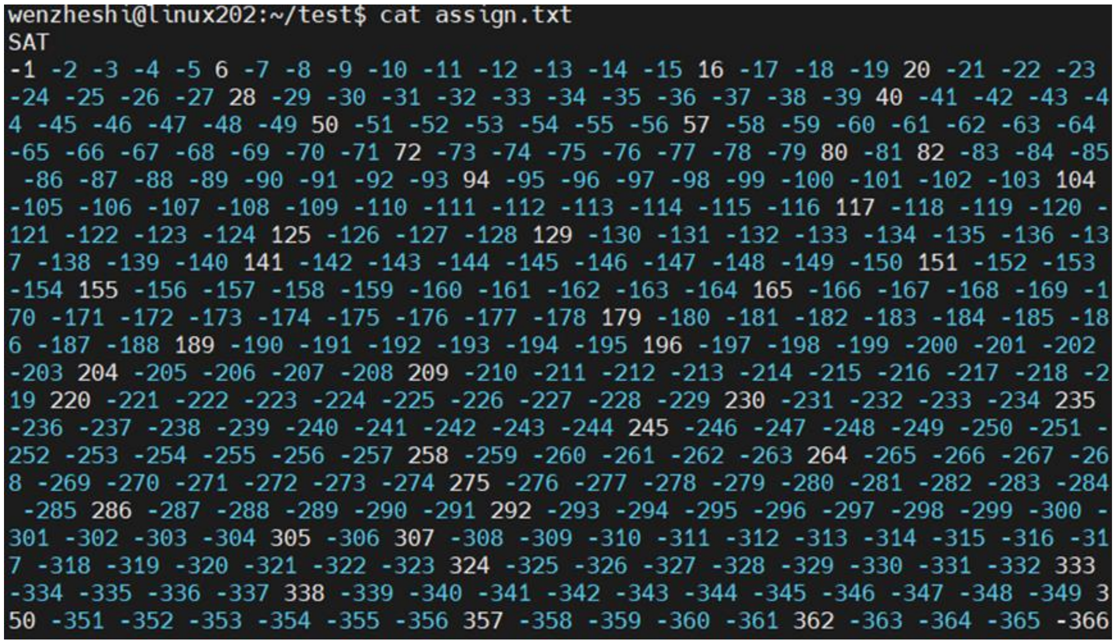
```
wenzheshi@linux202:~/test$ cat puzzle.cnf
p cnf 729 9053
1 2 3 4 5 6 7 8 9 0
10 11 12 13 14 15 16 17 18 0
19 20 21 22 23 24 25 26 27 0
28 29 30 31 32 33 34 35 36 0
37 38 39 40 41 42 43 44 45 0
46 47 48 49 50 51 52 53 54 0
55 56 57 58 59 60 61 62 63 0
64 65 66 67 68 69 70 71 72 0
73 74 75 76 77 78 79 80 81 0
82 83 84 85 86 87 88 89 90 0
91 92 93 94 95 96 97 98 99 0
```

Figure 1: The first few lines of the converted CNF formula

The execution result is shown above with a total of 9053 statements.

sat2sud.py

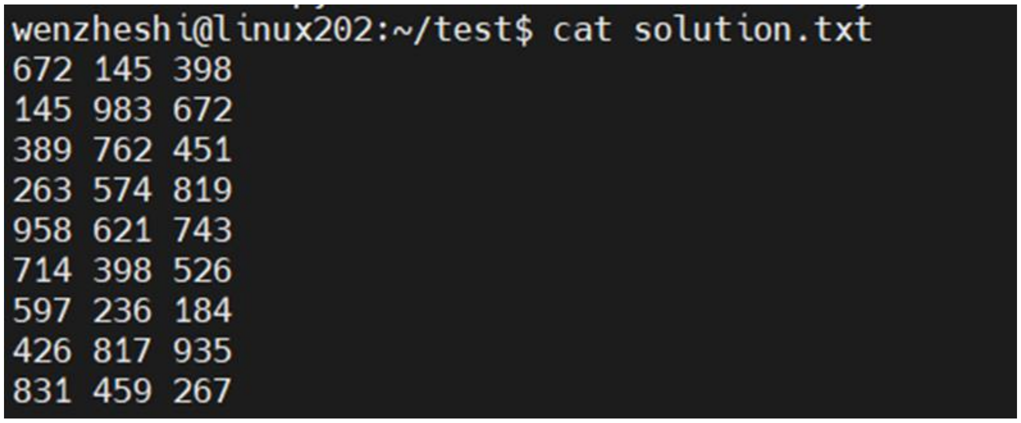
Opening the file “assign.txt” obtained from solving with *minisat*, we can see the coded solution.



```
wenzheshi@linux202:~/test$ cat assign.txt
SAT
-1 -2 -3 -4 -5 6 -7 -8 -9 -10 -11 -12 -13 -14 -15 16 -17 -18 -19 20 -21 -22 -23
-24 -25 -26 -27 28 -29 -30 -31 -32 -33 -34 -35 -36 -37 -38 -39 40 -41 -42 -43 -4
4 -45 -46 -47 -48 -49 50 -51 -52 -53 -54 -55 -56 57 -58 -59 -60 -61 -62 -63 -64
-65 -66 -67 -68 -69 -70 -71 72 -73 -74 -75 -76 -77 -78 -79 80 -81 82 -83 -84 -85
 -86 -87 -88 -89 -90 -91 -92 -93 94 -95 -96 -97 -98 -99 -100 -101 -102 -103 104
-105 -106 -107 -108 -109 -110 -111 -112 -113 -114 -115 -116 117 -118 -119 -120 -
121 -122 -123 -124 125 -126 -127 -128 129 -130 -131 -132 -133 -134 -135 -136 -13
7 -138 -139 -140 141 -142 -143 -144 -145 -146 -147 -148 -149 -150 151 -152 -153
-154 155 -156 -157 -158 -159 -160 -161 -162 -163 -164 165 -166 -167 -168 -169 -1
70 -171 -172 -173 -174 -175 -176 -177 -178 179 -180 -181 -182 -183 -184 -185 -18
6 -187 -188 189 -190 -191 -192 -193 -194 -195 196 -197 -198 -199 -200 -201 -202
-203 204 -205 -206 -207 -208 209 -210 -211 -212 -213 -214 -215 -216 -217 -218 -2
19 220 -221 -222 -223 -224 -225 -226 -227 -228 -229 230 -231 -232 -233 -234 235
-236 -237 -238 -239 -240 -241 -242 -243 -244 245 -246 -247 -248 -249 -250 -251 -
252 -253 -254 -255 -256 -257 258 -259 -260 -261 -262 -263 264 -265 -266 -267 -26
8 -269 -270 -271 -272 -273 -274 275 -276 -277 -278 -279 -280 -281 -282 -283 -284
 -285 286 -287 -288 -289 -290 -291 292 -293 -294 -295 -296 -297 -298 -299 -300 -
301 -302 -303 -304 305 -306 307 -308 -309 -310 -311 -312 -313 -314 -315 -316 -31
7 -318 -319 -320 -321 -322 -323 324 -325 -326 -327 -328 -329 -330 -331 -332 333
-334 -335 -336 -337 338 -339 -340 -341 -342 -343 -344 -345 -346 -347 -348 -349 3
50 -351 -352 -353 -354 -355 -356 357 -358 -359 -360 -361 362 -363 -364 -365 -366
```

Figure 2: the solved output of *minisat* given the converted CNF formula

Starting from the second line, the numbers represent the variable’s ID. A positive number indicates that the variable is true, otherwise it is false. Obtaining all the positive numbers allows us to obtain the solved Sudoku. The process of obtaining this is as follows. Starting from the second line, read all the strings. Split them by spaces, convert them to integers, and then iterate through the list. Find all numbers greater than 0. Taking their modulo 9 will convert them back to their original digits. Since multiples of 9 will have a modulo of 0, when the modulo result is 0 the original digit should be 9. Then, format and print the Sudoku in its original form, as shown in the figure below.

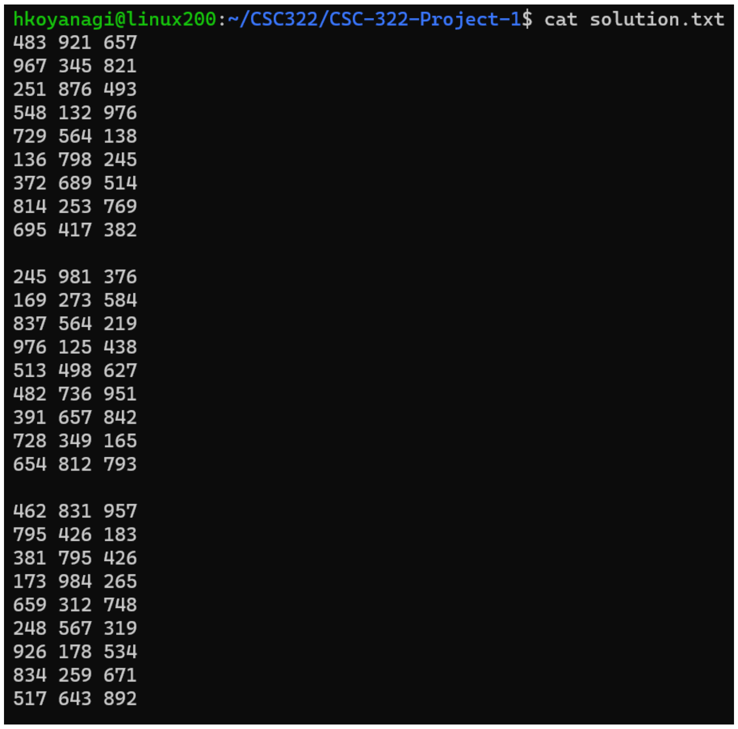


```
wenzheshi@linux202:~/test$ cat solution.txt
672 145 398
145 983 672
389 762 451
263 574 819
958 621 743
714 398 526
597 236 184
426 817 935
831 459 267
```

Figure 3: the resulting solved puzzle

Project Euler Test

To test our *sud2sat* and *sat2sud* programs we were given a list of 50 unsolved sudoku puzzles from the Project Euler website (found at https://projecteuler.net/project/resources/p096_sudoku.txt). To run and analyze these tests efficiently, we wrote a Bash Script that iterated through each puzzle. Each puzzle was run through *sud2sat,* the solved with *minisat,* converted into a readable solution with *sat2sud*. As the script solved each puzzle, each solution was concatenated to the end of a text file as seen below.



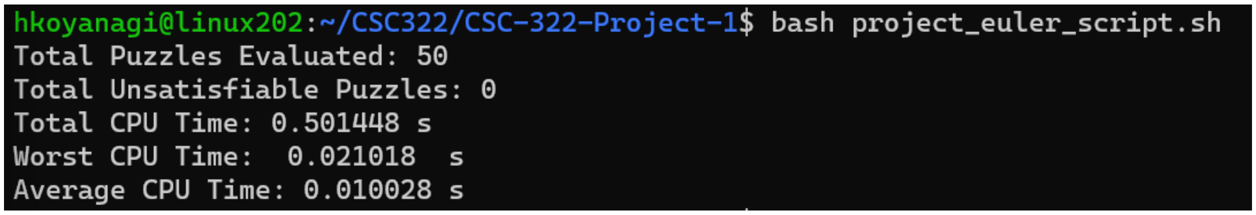
```
hkoyanagi@linux200:~/CSC322/CSC-322-Project-1$ cat solution.txt
483 921 657
967 345 821
251 876 493
548 132 976
729 564 138
136 798 245
372 689 514
814 253 769
695 417 382

245 981 376
169 273 584
837 564 219
976 125 438
513 498 627
482 736 951
391 657 842
728 349 165
654 812 793

462 831 957
795 426 183
381 795 426
173 984 265
659 312 748
248 567 319
926 178 534
834 259 671
517 643 892
```

Figure 4: the first three solved puzzles from the Project Euler text file

As well, for each iteration information was taken from the statistic file that resulted from the *minisat* evaluation. The stats we found were most indicative of the program efficiency included total CPU time and average CPU time as well as number of total puzzles evaluated and number of unsatisfiable puzzles for reference. These statistics were calculated by our script and shown in STDOUT.



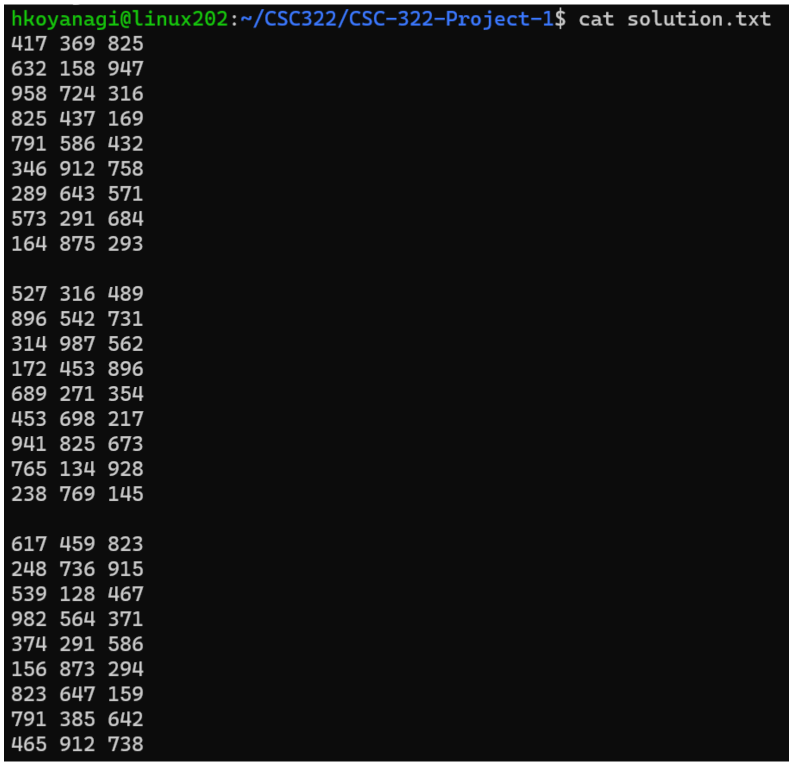
```
hkoyanagi@linux202:~/CSC322/CSC-322-Project-1$ bash project_euler_script.sh
Total Puzzles Evaluated: 50
Total Unsatisfiable Puzzles: 0
Total CPU Time: 0.501448 s
Worst CPU Time:  0.021018  s
Average CPU Time: 0.010028 s
```

Figure 5: an example of the statistics from finding the solutions to 50 puzzles

Task 1

For Task 1, we were to modify our programs to handle a text file (found at http://magictour.free.fr/top95) of 95 "hard" unsolved puzzles with differing format from the Project Euler file. The way in which we implemented our basic task, however, made it so our base code did not need to be updated. This was as our *sud2sat* code used the *dealWithRawData* function to remove whitespace and replace any other characters that were not numbers in the range 1-9 with "." before writing into a two-dimensional array. Doing so allowed for the universal conversion of both linear and grid-type puzzles with differing empty space holders to be converted into the needed CNF formulas.

We wrote another script to test our code against the 95 puzzles with the same structure as that for the Project Euler script with a few changes to account for the different formats. Some of the resulting statistics can be found below.

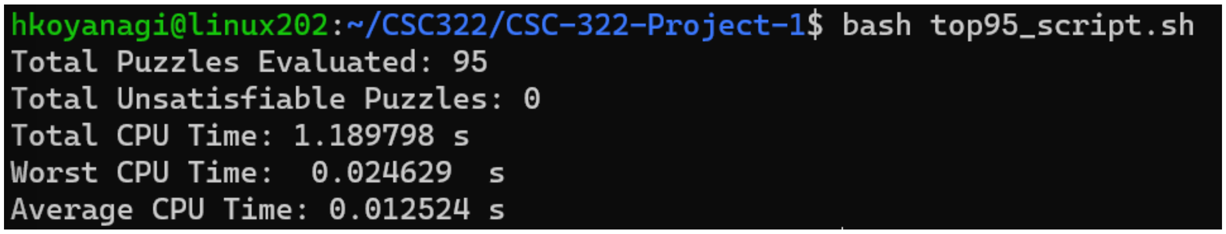


```
hkoyanagi@linux202:~/CSC322/CSC-322-Project-1$ cat solution.txt
417 369 825
632 158 947
958 724 316
825 437 169
791 586 432
346 912 758
289 643 571
573 291 684
164 875 293

527 316 489
896 542 731
314 987 562
172 453 896
689 271 354
453 698 217
941 825 673
765 134 928
238 769 145

617 459 823
248 736 915
539 128 467
982 564 371
374 291 586
156 873 294
823 647 159
791 385 642
465 912 738
```

Figure 6: the first three solved puzzles from the 95 puzzles text file



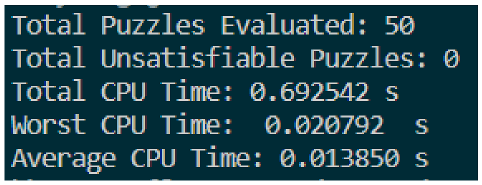
```
hkoyanagi@linux202:~/CSC322/CSC-322-Project-1$ bash top95_script.sh
Total Puzzles Evaluated: 95
Total Unsatisfiable Puzzles: 0
Total CPU Time: 1.189798 s
Worst CPU Time:  0.024629  s
Average CPU Time: 0.012524 s
```

Figure 7: an example of the statistics from finding the solutions to 95 puzzles

Given no changes had to be made to our code between the Basic Task and Task 1, the negligible difference between the average CPU time in processing the 50 puzzles versus the 95 puzzles (these being 0.011963s versus 0.012164s, respectively) is as expected. The worst-case CPU time however was consistently greater in the case of the 95 puzzles as opposed to the 50. This could be due to the increased workload of the CPU due to the apparently “harder” 95 puzzles, but more testing would be needed to determine this.

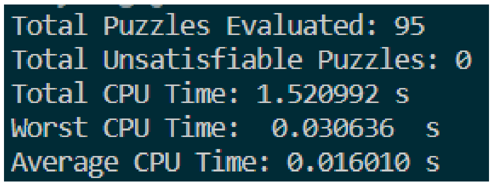
Task 2

For Task 2, we were to modify our code to adhere to efficient encoding. To do this, we added the clause “at most one number in each cell” to the CNF formulas. There were 729 resulting variables and 11969 resulting clauses. The statistics for the multi puzzle tests were as follows.



```
Total Puzzles Evaluated: 50
Total Unsatisfiable Puzzles: 0
Total CPU Time: 0.692542 s
Worst CPU Time:  0.020792  s
Average CPU Time: 0.013850 s
```

Figure 8: an example of the statistics from finding the solutions to 50 puzzles



```
Total Puzzles Evaluated: 95
Total Unsatisfiable Puzzles: 0
Total CPU Time: 1.520992 s
Worst CPU Time:  0.030636  s
Average CPU Time: 0.016010 s
```

Figure 9: an example of the statistics from finding the solutions to 95 puzzles

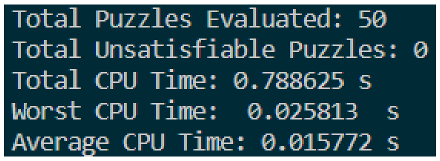
From Figures 8 and 9 above, we can see that there was a slight increase in both worst and average CPU time between the 50 puzzle test and the 95 puzzle test, similarly to that of the basic task. In comparison to the basic task however, the worst CPU time and the average CPU time were both increased aside from a slightly lower worst CPU time with regards to the 50 puzzle test.

Task 3

For Task 2, we were to modify our code to adhere to extended encoding. To do this, we added the following clauses to the CNF formulas:

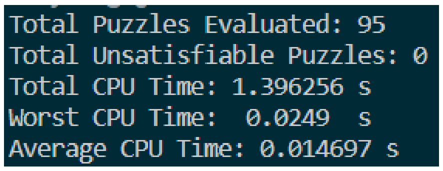
- There is at most one number in each cell
- Every number appears at least once in each row
- Every number appears at least once in each column
- Every number appears at least once in each subgrid

There were 729 resulting variables and 11969 resulting clauses. The statistics for the multi puzzle tests were as follows.



```
Total Puzzles Evaluated: 50
Total Unsatisfiable Puzzles: 0
Total CPU Time: 0.788625 s
Worst CPU Time:  0.025813  s
Average CPU Time: 0.015772 s
```

Figure 10: an example of the statistics from finding the solutions to 50 puzzles



```
Total Puzzles Evaluated: 95
Total Unsatisfiable Puzzles: 0
Total CPU Time: 1.396256 s
Worst CPU Time:  0.0249  s
Average CPU Time: 0.014697 s
```

Figure 11: an example of the statistics from finding the solutions to 95 puzzles

In comparing the results of the 50 puzzle test to that of task 2 (Figure 10 versus Figure 8), we can see that the worst CPU time and the average CPU time had a had a similar increase to that between the basic task and task 2. The 95 puzzle test, however, had lower worst CPU time and average CPU time than that of task 2 (Figure 9) but still greater than that of the basic task (Figure 7). This gives evidence that the extended coding could be more efficient for higher numbers of puzzles being solved.

Conclusion

In comparing the efficiency of the minimal encoding versus the efficient encoding versus the extended encoding, the minimal encoding performed the best. However, the increased efficiency of the extended encoding in task 3 suggests that the extended encoding could be more efficient for higher workloads. As well, our code was itself not as efficient as it could be as it recalculated all the CNF formulas for each iteration where hard-coded formulas may have been more efficient. With the efficient and extended encodings, the amount of CNF formulas to be calculated increased. This could have stunted some findings for the lower numbers of puzzles evaluated if the CPU time lost evaluating more CNF formulas was greater than the time gained from having more efficient encodings.