CSL - 452 Artificial Intelligence

Lab 3

Gaurav Mittal (2012CSB1013), Kaushal Yagnik (2012CSB1039)

1. Sudoku Solver using CSP

The objective was to implement a Sudoku solver as a constraint satisfaction problem. The puzzle is formulated as a constraint satisfaction problem as follows:

- The variable of the problem are each and every 81 cells of the Sudoku puzzle.
- Naturally, the domain of these variables will be 1-9
- Further, each variable can have exactly one value from the domain.
- Also, there are row, column and block constraint of all-diff type, that is, no two values in a row, column or block can be the same.
- The all-diff constraints can be converted into binary constraints by introducing a not-equal constraints between each two variables belonging to a row, column or block.

Based on this, the constraint satisfaction problem is formulated and is solved using different variants:

1. Generic Backtracking search with node and arc-consistency constraints - BS
2. Integrating (1) with minimum remaining values(MRV) heuristic - BS-I
3. Integrating (2) with least constraining value(LCV) heuristic - BS-II
4. Integrating (3) with forward checking - BS-FC
5. Integrating (3) with MAC algorithm - BS-MAC

These performance of these algorithms implemented in Python are compared based on various performance metrics by running on the given 150 sudoku puzzles. The table summarizing the results is given as follows:

Performance Metric	**BS**	**BS-I**	**BS-II**	**BS-FC**	**BS-MAC**
Avg. Search Time (sec)	**14.198**	**25.167**	**27.331**	**7.193**	**7.716**
Max Search Time (sec)	**1,169.259**	**2,039.924**	**1,037.735**	**236.304**	**195.350**
Min Search Time(sec)	0.274	0.130	0.348	0.121	0.411
Avg. Peak Memory Utilization (KB)	20,478.08	19,993.57	20,071.06	20,014.18	20,957.14
Max Peak Memory Utilization (KB)	20,480	19,988	24,116	20,016	25,148
Min Peak Memory Utilization (KB)	20,472	19,988	20,024	20,008	20,812
Avg. Backtracks	**1,739,257.71**	**1,516,685.13**	**2,092,185.74**	**311,518.68**	**11,843.80**
Max Backtracks	**142,620,465**	**116,038,053**	**80,601,424**	**10,038,159**	**329,799**
Min Backtracks	179	298	1070	256	0

Observations and Conclusions

From the table, it is clearly evident that BS-MAC performs much better than any other backtracking search algorithms giving a significantly lower number of backtracks and lower search time. Some other important observations are as follows:

- The average search time for BS-I and BS-II is quite higher than BS. This might be because in BS, arc consistency (AC3) is applied before solving the sudoku which for this particular problem seems to be more efficient in reducing the search time than MRV and

LCV heuristics. Further, memory initialization during intermediate steps may be adding to the search time as well for BS-I and BS-II.

- However, the minimum average search time is achieved using inference heuristics such as forward checking or MAC algorithm.

- The maximum search time is again higher in BS-I than BS suggesting the AC3 proves to be more efficient than MRV for solving a sudoku.

- Also, between BS-FC and BS-MAC, the maximum search time is lower for the latter suggesting that recursively checking for all arcs allows solving the puzzle is much less time than doing just forward checking.

- The minimum search time for all the algorithms is comparable since some sudoku puzzle can be easily solved even doing simple backtracking and application of an unnecessarily sophisticated algorithm simply compensates for the speedup attained using it. Similar conclusion can be drawn for minimum number of backtracks.

- The peak memory usage statistics are again comparable for all the techniques suggesting that the memory usage by a script is more dependent memory buffer used by the language (here, Python) and thus, giving no insight into the varying performance of the different techniques.

- The average number of backtracks shows a smooth decrease as we go from generic backtracking to the application of all sorts of heuristics (MRV + LCV + MAC) with the exception of BS-II. Except for BS-II, it can be concluded from this statistic that using better heuristics helps in significantly improving the performance of solving a constraint satisfaction problem through backtracking.

- The anomalously higher number of average backtracks in case of BS-II can be due to the fact that using LCV seems to select a value which in the long run can result in higher number of backtracks. This suggests that if we choose a value for a variable which poses least amount of constraints on its neighbors, it might initially appear to be approaching a solution but in this end, it fails to do so causing a higher than ever number of backtracks which could have been avoided, if LCV was not applied altogether.

- However, the statistic maximum number of backtracks suggest the applying increasing amount of heuristics helps in reducing the maximum bound on the number of backtracks which could occur.

As a final conclusion, it can be stated that the performance of backtracking in solving a constraint satisfaction problem significantly improves by the application of heuristics such as MRV, LCV, FC and MAC. And, a combination of these gives an even better performance!

2. Sudoku Solver using MiniSAT

In this question, the sudoku puzzle is solved as a Boolean satisfiability problem by formulating boolean sentences in the Conjunctive Normal Form (CNF) as input to MiniSat which gives the solution in case it is satisfiable. The solution of MiniSat is then parsed to get the solved Sudoku puzzle.

The following describes the process of converting constraints of a Sudoku puzzle to Boolean sentences in CNF:

If a variable at position [row,col] contains value val, then it is represented as
100* row + 10*col + val
and it is negated if the value is not present there **-100*row - 10*col - val**

Eg. cell (2,3) contains value 7, it will be represented as **237** and if it doesn't, then it will be **-237**

STATIC CONSTRAINTS (SAME FOR ANY SuDoku)

Singular value for cell constraint:

- To indicate that a cell contains exactly one value in the range 1 to 9. For the cell [2,3]. The clause
  231 232 233 234 235 236 237 238 239 0
  Repeat this for all 81 cells of the SuDoku

- Now, to ensure that one cell does not contain multiple values, we use
  -231 -232 0
  Meaning value of [2,3] can't be 1 and 2 at the same time
  Repeat this for all 81 cells of the SuDoku

Cells in Rows constraint:

- To ensure that each row contains atleast one 1, we use the constraint
  311 321 331 341 351 361 371 381 391 0
  This will ensure that the third row contains atleast one 1
  Repeat for all different values (1-9) for the third row. And similarly repeat for all the rows.

- To ensure that there are no more than one 1 in a row, we take the constraint
  -221 -231 0 [2,2] and [2,3] cannot simultaneously be 1

-221 -241 0   [2,2] and [2,4] cannot simultaneously be 1

This will ensure that [2,2] and [2,3] cannot simultaneously be 1 and so on... Similarly repeat for all different values (1-9) and for all different combinations of cells in a particular row.

Cells in Column constraint:

- At least one value constraint: To ensure that each column contains at least one 1, we use the constraint
  121 221 321 421 521 621 721 821 921 0
  This will ensure that the second column contains at least one 1
  Repeat for all different values (1-9) for the second column. And similarly repeat for all the columns.
- To ensure that there are no more than one 1 in a row, we take the constraint
  -231 -331 0   [2,3] and [3,3] cannot simultaneously be 1
  -231 -431 0   [2,3] and [4,3] cannot simultaneously be 1

  This will ensure that [2,3] and [3,3] cannot simultaneously be 1 and so on... Similarly repeat for all different values (1-9) and for all different combinations of cells in a particular column.

Cells in a Box constraint:

- At least one value constraint: To ensure that each box contains at least one 1, we use the constraint
  111 121 131 211 221 231 311 321 331 0
  This will ensure that the first box column contains at least one 1
  Repeat for all different values (1-9) for the first box. And similarly repeat for all the boxes.

- To ensure that there are no more than one 1 in a box, we take the constraint
  -111 -121 0   [1,1] and [1,2] cannot simultaneously be 1
  -111 -131 0   [1,1] and [1,3] cannot simultaneously be 1

  This will ensure that [1,1] and [1,2] cannot simultaneously be 1 and so on... Similarly repeat for all different values (1-9) and for all different combinations of cells in a particular box.

DYNAMIC CONSTRAINTS (dependent on SuDoku)

- The input values of the SuDoku.
  If the cell [6,9] contains the value 4, it will be represented as

  694 0

  This will ensure that the cell [6,9] has the value 4.
  Repeat this for all the input values of the SuDoku.

METHODOLOGY

- First the static constraints are stored in a particular file named ‘constraints’ so that they can be reused again instead of generating them again and again.
- For a particular SuDoku, the dynamic constraints are created and the static constraints are copied in.
- Then, it is given as input to the MiniSAT solver and the generated output file is parsed to know the value of the variables.
- The answer to the SuDoku is appended to the outputFile. In case the SuDoku is unsolvable, it output ‘UNSAT’ followed by the input sudoku puzzle in place of the solution.