CS3243 Group 25 Project 2: Sudoku Solver using Backtracking Search

Choong Jin Yao (A0173247A), Dominic Frank Quek (A0173261L), Yang Shuqi (A0177383U), Yang Yiqing (A0161424N)

No Institute Given

1 Introduction

Constraint Satisfaction Problem (CSP) search algorithm uses the structure of states and a general purpose heuristic to solve problems. We are given the task of implementing a Sudoku solver using backtracking algorithm with relevant heuristics.

2 Problem Specification

The puzzle is modeled as a list of 81 cells where each cell is represented by an (x, y) coordinate where $1 \leq$ `x,y` ≤ 9. Every cell is assigned a value `v` where $0 \leq$ `v` ≤ 9. An empty cell is where $v = 0$. The variables are the coordinates of empty cells in the given input puzzle. A list `unassigned_var_list` contains the variables that are yet to be assigned a value. The constraints are that for any of the 9 rows, 9 columns and 9 diagonals, no value can appear more than once. To enforce this constraint, at the start, we initialise 27 lists containing the legal values for each row, column and diagonal after removing the pre-assigned values in the given input puzzle. Then, we keep track of the domain of each variable where the domain contains the legal values to be assigned to that variable. The domains are updated to be consistent with the 27 lists.

Afterwards, we begin the backtracking search algorithm. When a value is assigned to a variable, the variable is removed from `unassigned_var_list` and its domain becomes empty. After every assignment, we use heuristics to remove illegal values from the domains of neighbouring variables. Hence, the constraints are guaranteed not to be violated throughout.

3 Algorithm variants

We implemented a backtracking solver that incorporates these variants:

1) Variable ordering: choosing the most constrained variable with Minimum Remaining Values (MRV) heuristic with/without tie-breaking with Degree heuristic

2) Value ordering: Least Constraining Value (LCV) heuristic

3) Inference: Forward Checking or Arc Consistency-3(AC3) .

After implementation, we evaluate and select the best variant from the timinig gathered from the heuristics on the Sudoku problem.

3.1 Backtracking

Backtracking is a recursive search algorithm that allows for reversion of value assignments when a certain complete assignment turns out to be unsuccessful. When that occurs, we restore the domains prior to the assignment to the variable. It recurses if the assigned variable is consistent with its neighbouring constraints. The backtracking algorithm forms the backbone of the entire search as seen in the function `backtracking_search`. Within the backtracking algorithm, heuristics are implemented in 3 aspects: Variable Ordering, Value Ordering and Inference.

3.2 Variable Ordering

We prioritized the variable for expansion that is the most constrained. For this, we experimented with 2 heuristics: Minimum Remaining Value (MRV) and Degree.

Most Constrained Variable: Minimum Remaining Value Minimum Remaining Value chooses the variable with the least number of legal values. This is basically the variable with the smallest domain size. Note that when the domain size of an unassigned variable is 0, the assignment is impossible and the algorithm will backtrack. MRV is implemented in the function `minimum_remaining_values`.

Most Constraining Variable: Degree Degree heuristic acts as a tie-breaker among the Most Constrained Variable using MRV heuristic. Among variables that have least and equal domain sizes, the variable with the most unassigned neighbours is prioritized. By doing this, it potentially further reduces the search space as it leads to a maximal reduction in domains for other variables. Degree heuristic is implemented in the function `degree`.

3.3 Value Ordering: Least Constraining Value

Least Constraining Value heuristic prioritizes the domain value of a variable that results in the minimum reduction in the domains of neighbouring variables. In our implementation, we count the number of conflicts between the value and the values of its neighbours. A conflict simply happens when the value of the target variable is contained in the domain of a neighbouring variable. The value with the least number of conflicts is then prioritized. LCV is implemented in the function `least_constraining_value` that sorts the variable domain using the `count_conflicts` function as the key.

3.4 Inference

Inference is effective in reducing the domain of variables hence significantly reducing search space. Whenever a value is assigned to a variable, we can infer new domain reductions on its neighbouring variables. We have experimented with 2 variants: Forward Checking (FC) and Arc Consistency (AC-3).

Forward Checking Forward Checking ensures that a value assignment is consistent with the domains of all its neighbouring variables. Fundamentally, it ensures that the assigned variable is arc-consistent with the other variables. FC prunes the the domain values of variables that are in conflict with the value assignment. Failure is detected when any of the reduced domains is empty as a result. This is implemented in the function `forward_checking`.

AC-3 Whereas FC ensures arc consistency between a certain variable and the other variables, AC-3 ensures arc consistency between any 2 variables after every value assignment. This requires an initilisation of binary constraints where we create a tuple for each pair of neighbour variables as seen in the function `init_binary_constraints`. After every assignment, we loop through the list of binary constraints, reducing the domains whenever necessary. Whenever a domain is reduced using the `revise` function, the binary constraints of the neighbours are added to the queue. Evidently, this can be computationally expensive. This is implemented in the function `ac_3` and the subfunction `revise`.

4 Experimental Setup

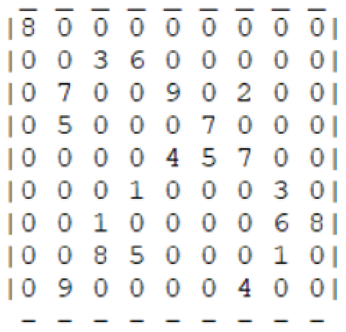
For the experimental setup, we run different variants of the aforementioned heuristics. We measure the perfomance of each variant by measuring the running time. Measuring the runtime allows us to compare the efficiency of the variants in terms of time. The runtime of the variant covers all the work done and computations by the involved heuristics.

We find it inaccurate and difficult to measure and compare the search space of different variants via computing the number of traversed nodes. It is impossible to establish a fair definition of what a node represents. This is due to the different implementations of various heuristics. For instance, for AC-3, on top of the number of value assignments made before arriving at a solution, it also requires checking through every binary constraint everytime an assignment occurs. For FC, it requires traversing the domains of remaining unassigned variables. We can simply define it to be the number of assignments made (equivalently, the number of calls to the `recursive_backtrack` function. However, that is clearly too trivial and inaccurate as the additional work done outside by the heurstic is not accounted for.

Additionally, we test the variants on the world's hardest Sudoku puzzle that we've found online (Taken from https://www.conceptispuzzles.com/index.aspx?uri=info/article/424).

5 Results and Discussion

The results for runtime performance is given in Table 1, number of nodes explored is given in Table 2. Note that for runtimes that are longer than 300 seconds, we put a '-' symbol.



```
 _ _ _ _ _ _ _ _ _
|8 0 0 0 0 0 0 0 0|
|0 0 3 6 0 0 0 0 0|
|0 7 0 0 9 0 2 0 0|
|0 5 0 0 0 7 0 0 0|
|0 0 0 0 4 5 7 0 0|
|0 0 0 1 0 0 0 3 0|
|0 0 1 0 0 0 0 6 8|
|0 0 8 5 0 0 0 1 0|
|0 9 0 0 0 0 4 0 0|
 _ _ _ _ _ _ _ _ _
```

Fig. 1. World's Hardest Sudoku

As can be observed from Table 1, FC+MRV+LCV and AC3+MRV+LCV have the fastest overall run times for the Sudoku puzzle. Between these 2 variants, FC+MRV+LCV is faster for easier puzzles, whereas AC3+MRV+LCV is faster for harder puzzles. This is because AC3 leads to a significant reduction in search space, especially for harder puzzles with enormous search space. For simpler puzzles, however, ie. input1 and hardest, the reduction in runtime due to smaller search space in AC3 could be outweighed by the reduction in runtime due to less node traversal and computation in FC. This is due to the amount of computation needed per assignment as explained above. AC3 has to ensure arc consistency between every 2 variables, whereas FC simply ensures consistency between a certain variable and other variables. This is further confirmed when you compare AC3 and FC as seen in the last 2 columns of Table 1.

Implementing Degree heuristic, denoted as D in the table, slows down the search instead of speeding it up. This is evident in column 1(with D) and column 3(without D), as well as in column 2(with D) and column 4(without D). This is because the increase in time spent on computing the degree of every variable outweighs the reduction in time due to smaller search space. In fact, the purpose of the Degree heuristic is merely for tie-breaking for MRV. The extra work of computing the degree is not worth it.

It can also be noted that MRV and LCV play a significant role in the decrease in runtime. If we were to take both of them away, the runtime of the search will be significantly increased.

6 Further Evaluation between FC+MRV+LCV and AC3+MRV+LCV

If we must decide between these 2 variants, we will choose AC3+MRV+LCV. Our reasoning is that even though this variant may be slower for simpler puzzles, the difference in time is really small ($\leq 0.1s$). For complex puzzles, however, the time difference is much larger, up to 2 times faster(compare last 2 columns

Run time (seconds)	FC+MRV+LCV+D	AC3+MRV+LCV+D	FC+MRV+LCV	AC3+MRV+LCV
Sudoku (input1)	0.912	0.826	0.347	0.251
Sudoku (input2)	0.082	0.085	0.003	0.016
Sudoku (input3)	0.023	0.009	0.052	0.002
Sudoku (input4)	0.018	0.012	0.019	0.003
Sudoku (hardest)	0.901	0.825	0.349	0.253

Fig. 2. Experiments performed to measure runtime with various heuristics on Sunfire

Run time (seconds)	FC+MRV	FC+LCV	AC3+MRV	AC3+LCV	AC3	FC
Sudoku (input1)	0.912	-	0.616	-	120.42	236.654
Sudoku (input2)	0.013	0.095	0.022	0.033	0.141	0.520
Sudoku (input3)	0.023	0.002	0.146	0.023	0.023	0.110
Sudoku (input4)	0.011	0.003	0.802	0.011	0.018	0.350
Sudoku (hardest)	0.926	-	0.614	-	120.390	236.671

Fig. 3. Experiments performed to measure runtime with various heuristics on Sunfire

in Table 1). Using the stronger but more computationally expensive AC3 is preferred to FC.

7 Conclusion

Implementing heuristics are a very useful aspect of search algorithms. Backtracking search alone may take a long time to solve a Sudoku puzzle, but with the help of effective heuristics, the enormous search space can be quickly pruned to reduce the search time significantly.