Constraint Processing Assignment

Ramaravind Kommiya Mothilal
r0605648

- **Magic Square: 2x2**

  Leftmost heuristic is followed for magic square problems.

  a. Submitted variables and constraints:

  When the elements in the grid need not be unique,

  Name: A, domain: 1..4
  Name: B, domain: 1..4
  Name: C, domain: 1..4
  Name: D, domain: 1..4
  Constraint: A+B = C+D, C+D = A+C, A+C = B+D, B+D = A+D, A+D = B+C

  When the elements need to be unique, the constraint becomes,
  Constraint: A+B = C+D, C+D = A+C, A+C = B+D, B+D = A+D, A+D = B+C, A \= B, A \= C, A \= D, B \= C, B \= D, C \= D.

  b. Constraints in natural language:

  Sum of the elements in rows, columns and diagonals is one and the same.

  c. Number of solutions and output:

  Not Unique:

  Number of solutions: 4
  Output:
  Backtracks: 0, Runtime: 0.06s, [A,B,C,D] = [1,1,1,1]
  Backtracks: 0, Runtime: 0.0s, [A,B,C,D] = [2,2,2,2]
  Backtracks: 0, Runtime: 0.0s, [A,B,C,D] = [3,3,3,3]
  Backtracks: 0, Runtime: 0.0s, [A,B,C,D] = [4,4,4,4]

  Unique: Number of solutions: 0

- **Magic Square: 3x3:**

  a. Submitted variables and constraints:

  When the elements in the grid need not be unique,

  Name: A(i), i=0..8, domain: 1..9
  Constraint: A(i) + A(i+1) +A(i+2) = A(j) + A(j+1) +A(j+2), range: (i mod 3 = 0 , i<4) , j=i+3
  Constraint: A(i)+A(i+1)+A(i+2) = A(j)+A(j+3)+A(j+6), range: i=6 , j=0
  Constraint: A(i) + A(i+3) +A(i+6) = A(j) + A(j+3) +A(j+6), range: i<2 , j =i+1
  Constraint: A(i)+A(i+3)+A(i+6) = A(j)+A(j+4)+A(j+8), range: i=2 , j=0
  Constraint: A(i)+A(i+4)+A(i+8) = A(j)+A(j+2)+A(j+4), range: i=0 , j=2

  When the elements in the grid need to be unique,

  Name: A(i), i=0..8, domain: 1..9
  Constraint: A(i) + A(i+1) +A(i+2) = A(j) + A(j+1) +A(j+2), range: (i mod 3 = 0 , i<4) , j=i+3
  Constraint: A(i)+A(i+1)+A(i+2) = A(j)+A(j+3)+A(j+6), range: i=6 , j=0
  Constraint: A(i) + A(i+3) +A(i+6) = A(j) + A(j+3) +A(j+6), range: i<2 , j =i+1
  Constraint: A(i)+A(i+3)+A(i+6) = A(j)+A(j+4)+A(j+8), range: i=2 , j=0
  Constraint: A(i)+A(i+4)+A(i+8) = A(j)+A(j+2)+A(j+4), range: i=0 , j=2
  Constraint: A(i)\=A(j), range: i<9 /\ j<9 /\ i<j

b. Constraints in natural language:
Sum of the elements in rows, columns and diagonals is one and the same.

c. Number of solutions and output:
Not unique: <u>Number of solutions:</u> 129
Unique:<u>Number of solutions:</u> 8

<u>Output:</u>

```
Backtracks: 43, Runtime: 0.05s, [A] = [[2,7,6,9,5,1,4,3,8]]
Backtracks: 16, Runtime: 0.0s, [A] = [[2,9,4,7,5,3,6,1,8]]
Backtracks: 96, Runtime: 0.01s, [A] = [[4,3,8,9,5,1,2,7,6]]
Backtracks: 74, Runtime: 0.0s, [A] = [[4,9,2,3,5,7,8,1,6]]
Backtracks: 133, Runtime: 0.0s, [A] = [[6,1,8,7,5,3,2,9,4]]
Backtracks: 83, Runtime: 0.0s, [A] = [[6,7,2,1,5,9,8,3,4]]
Backtracks: 113, Runtime: 0.0s, [A] = [[8,1,6,3,5,7,4,9,2]]
Backtracks: 14, Runtime: 0.0s, [A] = [[8,3,4,1,5,9,6,7,2]]
```

- **<u>Sudoku:</u>**

Unless otherwise stated, these default parameters are followed throughout the problem: (<u>Variable choice heuristic:</u> Leftmost; <u>Value choice heuristic</u>: Step, Up)

1. a. Submitted variables and constraints:

Name: A(i), i=0..80, domain: 1..9
Constraint: A(i) \= A(j), range: (i/9 = j/9) /\ (i mod 9 < j mod 9) /\ (i<j)
Constraint: A(i) \= A(j), range: (i mod 9 = j mod 9) /\ (i/9 < j/9) /\ (i<j)
Constraint: A(i) \= A(j), range: (i/27 = j/27) /\ (i/3 mod 3 = j/3 mod 3) /\ (i<j)

b. Constraints in natural language:

The first constraint confirms that every element in each of the 9 rows is distinct. The second constraint confirms the same condition for each of the 9 columns. The third constraint takes sets of 3 rows in turn and confirms that every variable in each of the three 3x3 squares (that are in the considered set of 3 rows) is distinct.

2. Number of solutions and output for given puzzles:
a. Easy 1:
<u>Number of solutions:</u> 1
<u>Output:</u>

```
Backtracks: 0
Runtime: 0.04s
[A] =
[[4,6,1,5,2,9,8,3,7,7,2,5,8,3,1,6,9,4,3,9,8,6,4,7,1,5,2,8,3,6,2,1,5,4,7,9,9,5,2,7,8,4,3,6,1,1,7,
4,3,9,6,5,2,8,5,4,3,9,7,8,2,1,6,6,1,9,4,5,2,7,8,3,2,8,7,1,6,3,9,4,5]]
```

b. Easy 2:
<u>Number of solutions:</u> 1
<u>Output:</u>

```
Backtracks: 0
Runtime: 0.04s
```

```
[A] =
[[1,3,9,5,8,2,6,7,4,4,8,6,9,7,1,3,5,2,7,2,5,4,6,3,9,1,8,9,4,1,6,5,8,7,2,3,2,5,7,3,4,9,1,8,6,8,6,
3,2,1,7,5,4,9,5,9,4,7,2,6,8,3,1,6,7,8,1,3,4,2,9,5,3,1,2,8,9,5,4,6,7]]
```

c. Easy 3:
Number of solutions: 1
Output:

```
Backtracks: 0
Runtime: 0.04s
[A] =
[[7,4,2,1,8,3,6,9,5,6,1,3,5,4,9,8,7,2,8,9,5,7,6,2,1,4,3,2,5,7,6,1,4,9,3,8,1,8,9,3,5,7,2,6,4,3,6,
4,2,9,8,5,1,7,4,2,6,9,7,5,3,8,1,9,3,8,4,2,1,7,5,6,5,7,1,8,3,6,4,2,9]]
```

d. Difficult 1:
Number of solutions: 1
Output:

```
Backtracks: 20
Runtime: 0.04s
[A] =
[[1,6,5,7,8,3,2,9,4,9,3,2,6,5,4,8,7,1,4,7,8,2,9,1,3,5,6,8,9,7,3,4,6,5,1,2,3,5,1,9,7,2,4,6,8,6,2,
4,8,1,5,9,3,7,7,1,9,4,3,8,6,2,5,5,8,6,1,2,9,7,4,3,2,4,3,5,6,7,1,8,9]]
```

e. Difficult 2:
Number of solutions: 1
Output:

```
Backtracks: 59
Runtime: 0.04s
[A] =
[[4,2,6,5,1,7,8,3,9,9,7,1,4,8,3,2,5,6,5,8,3,9,6,2,4,7,1,7,1,9,2,5,4,3,6,8,6,5,8,3,9,1,7,4,2,3,4,
2,6,7,8,9,1,5,8,9,5,7,4,6,1,2,3,1,3,4,8,2,5,6,9,7,2,6,7,1,3,9,5,8,4]]
```

f. Difficult 3:
Number of solutions: 1
Output:

```
Backtracks: 1
Runtime: 0.04s
[A] =
[[7,2,3,9,5,6,4,1,8,5,8,4,3,1,7,2,6,9,9,1,6,2,4,8,3,7,5,3,5,8,4,6,1,7,9,2,6,4,9,7,8,2,5,3,1,1,7,
2,5,3,9,6,8,4,8,9,7,6,2,4,1,5,3,4,3,1,8,7,5,9,2,6,2,6,5,1,9,3,8,4,7]]
```

Observations:

The number of backtracks for ‘easy’ sudokus are 0. This shows that the solver must have found a solution just by employing arc consistency algorithm either only once or in certain iterations until a solution was found. The zero backtracks also shows that there were no inconsistency raised in the process of finding a solution. But for ‘difficult’ sudokus, the backtracks are non-zero. Hence, while solving these problems, the solver must have gone through inconsistencies (such as a variable ending in empty domain) and must have backtracked over the assignments until a solution was found. The more the number of backtracks, the more the inconsistencies. So if the difficulty of a sudoku problem is estimated from the number of backtracks, then the second difficult sudoku problem is more difficult than the first which in turn is more difficult than the third.

As the same constraints are imposed for both easy and difficult sudokus, the total time taken is more or less the same for all sudokus for a number of trials. This also shows the very least amount of time is spent in backtracking. Also note that other factors such as processor built, background processes etc might have accounted for the minute differences in time taken for the sudokus in different trials.

3. Comparison of leftmost and first fail+constraint count:

With first fail + constraint count heuristic, the backtracks differ as shown in the table:

	Difficult 1		Difficult 2		Difficult 3	
	leftmost	first fail+count	leftmost	first fail+count	leftmost	first fail+count
Backtracks	20	13	59	12	1	0
Total time taken	0.04	0.04	0.04	0.05	0.04	0.05

With first fail + constraint count heuristic, those variables which has highest number of non-trivial constraints are first selected and evaluated (before those whose constraints might be trivial like just “true” which always holds), such that the chance of those variables getting successful solution is reduced. As the variables which are likely to fail in the future are removed initially, the branching factor is reduced and/or the failing branches are cut off as soon as possible. It is because of this that the number of backtracks is significantly reduced while employing this heuristic. Whereas the leftmost heuristic considers every variable in order from left to right irrespective of their prospective success/failure rate in the future, thus solving with more backtracks compared to first fail.

4. “Two out of three rule” strategy:

a. Submitted variables and constraints:
Name: A(i), i=0..80, domain: 1..9
Constraint: A(i) \= A(j), range: (i/9 = j/9) /\ (i mod 9 < j mod 9) /\ (i<j)
Constraint: A(i) \= A(j), range: (i mod 9 = j mod 9) /\ (i/9 < j/9) /\ (i<j)
Constraint: A(i) \= A(j), range: (i/27 = j/27) /\ (i/3 mod 3 = j/3 mod 3) /\ (i<j)
Constraint: A(i) = A(j) => ((A(k) = A(i)) \/ (A(k+9) = A(i)) \/ (A(k+18) = A(i))), range: ((i/3 mod 3 = j/3 mod 3) /\ (j/3 mod 3 = k/3 mod 3)) /\ ((i mod 3 \= j mod 3) /\ (j mod 3 \= k mod 3) /\ (i mod 3 \= k mod 3)) /\ ((i/27 = 0) /\ (j/27 = 1) /\ (k/27 = 2) /\ (k/9 = 6))
Constraint: A(i) = A(j) => ((A(k) = A(i)) \/ (A(k+1) = A(i)) \/ (A(k+2) = A(i))), range: ((i/27 = j/27) /\ (j/27 = k/27)) /\ ((i/9 \= j/9) /\ (j/9 \= k/9) /\ (i/9 \= k/9)) /\ ((i/3 mod 3 = 0) /\ (j/3 mod 3 = 1) /\ (k/3 mod 3 = 2) /\ (k mod 3 = 0))

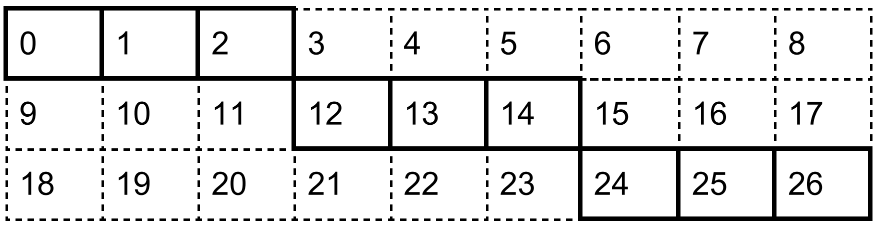
b. Constraints in natural language:
The first three constraints are same as explained in sudoku problem 1.b. Fifth constraint is explained below. The logic for the fourth is similar.

The range for fifth constraint is split as
➔ ((i/27 = j/27) /\ (j/27 = k/27))
This part consider sets of 3 rows in turn

➔ ((i/9 \= j/9) /\ (j/9 \= k/9) /\ (i/9 \= k/9))
This part confirms that each of the three rows (in the considered set of 3 rows) is different

➔ ((i/3 mod 3 = 0) /\ (j/3 mod 3 = 1) /\ (k/3 mod 3 = 2) /\ (k mod 3 = 0))
This part takes ‘i’ from the first block, ‘j’ from the second block and ‘k’ from the first column of the third block.

And the constraint A(i) = A(j) => ((A(k) = A(i)) \/ (A(k+1) = A(i)) \/ (A(k+2) = A(i))) implies that
If A(i) from first block & row ‘i/9’ equals A(j) from second block & row ‘j/9’, then A(i) must be equal to A(k) or A(k+1) or A(k+2) where ‘k’ is in row ‘k/9’ and in either of the first or second or third column of the third block. The following figure explains the procedure followed:



0	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

Consider the first 3 rows of the grid. According to the range given, when ‘i’ takes value from location 0,1 or 2, ‘j’ takes value from location 12, 13 or 14. Now, ‘k’ actually takes only value 24. But from the constraint condition, if A(i) = A(j), then A(i) must be equal to any one of the following values, namely A(k) or A(k+1) or A(k+2), i.e, location 24 or 25 or 26.

5, 6. How this extension affects the solution?

The output and the number of solutions is the same as found using the naive strategy (hence not stated explicitly again) but the total time taken and the backtracks differ as shown in the following table,

	Easy 1		Easy 2		Easy 3	
	naive	2 out of 3	naive	2 out of 3	naive	2 out of 3
Backtracks	0	0	0	0	0	0
Total time taken	0.04	1.99	0.04	1.96	0.04	1.99

	Difficult 1		Difficult 2		Difficult 3	
	naive	2 out of 3	naive	2 out of 3	naive	2 out of 3
Backtracks	20	6	59	15	1	0
Total time taken	0.04	1.96	0.04	1.97	0.04	2.01

This strategy imposes more constraints that forces variables to take less number of values than the naive strategy. For difficult sudokus, in a way, this reduces the branching factor and/or failing branches very early compared to naive strategy. Consequently, the number of backtracks are reduced.

Given that the total time taken is the sum of time taken to iterate over all the indices in order to impose the constraints and the time taken to satisfy those constraints. As the number of constraints in the ‘2 out of 3 rule’ extension is more and relatively complex than the naive constraint, the complete run time increases significantly (by a factor of around 40) for the former.

Hence, it can be observed that if the difficulty levels are perceived beforehand, then for simple sudokus, it’s better to use naive strategy itself as with fewer and simple constraints, solutions can be found in a shorter time. But for difficult sudokus, the situation is trickier. If the solution has to be computed in lesser time, naive strategy is prefered for the same reason explained above. But if backtracks has to be reduced for reasons like optimizing space complexity etc, then ‘2 out of 3 rule’ extension is prefered.

7. Influence of employed heuristics:

	Easy 1		Easy 2		Easy 3	
	leftmost	first fail+count	leftmost	first fail+count	leftmost	first fail+count
Backtracks	0	0	0	0	0	0
Total time taken	1.99	2.02	1.96	2.01	1.99	2.04

	Difficult 1		Difficult 2		Difficult 3	
	leftmost	first fail+count	leftmost	first fail+count	leftmost	first fail+count
Backtracks	6	2	15	12	0	1
Total time taken	1.96	1.96	1.97	2.01	2.01	1.98

The time taken is almost the same whatever be the heuristic or the difficulty of the problem as the constraints are unchanged.

But for difficult sudokus, influence can be observed with respect to the number of backtracks. As explained in q.4, with first fail + constraint count heuristic, the chance of those variables (which has highest number of non-trivial constraints) getting successful solution is reduced, thereby reducing number of backtracks. But there is an outlier namely for the difficult problem 3. The reason could be that the search might have become too much complicated by the first fail heuristic (or by due to the combination of extended strategy and heuristic) resulting in backtracking which otherwise could have been solved with no backtracks using simple leftmost heuristic.