Program Design
The program was designed in two pieces. The CSP was designed as a graph. There is a list of variables (just an id), a mapping of variable to domain, and a list of constraints. The constraints hold a list of variables and the sum constraint has an amount to sum to. Each constraint also have a function to generate equivalent arcs, for AC-3. For the purpose of this project, each sum constraint returns an empty list so they are ignored in the preprocessing stage. The various algorithms are implemented very closely to the pseudo code described in the textbook and can operate on any CSP.

The 3 sudoku variations each have a representation and handle input validation. Each sudoku has a function ToCSP() that will return a constraint satisfaction problem that is equivalent to itself. This way all puzzles can be treated the same after this conversion.

User Manual
To use this program, a config.json file is provided. There are details of how to use this file in the README.md, but essentially there are a list of inputs. Certain values are actually lists instead of values, and so for each input, every permutation of one element from each list is ran as a separate CSP. The output is put into log.txt. There is a global parameter that sets the seconds before a given puzzle times out. To compile the program you can use “go build .”. Then run the generated executable. To save some steps, you can build and run in the same command with “go run .”.

Puzzle Encoding
Each puzzle has its own encoding with a function to parse. The normal sudoku is represented as an 81 character string of digits. The overlapping sudoku is represented as a 171 character string of digits. Both of these are the ordering of the tiles from top to bottom, left to right. The killer sudoku is not so simple, as the length is not known. Instead there is a json file with the sudoku board encoding and also a list of cages. Unfortunately this is extremely cumbersome to create from a puzzle and after an hour my encoding didn’t give a correct answer, so there are no working examples of the killer sudoku. The sum constraint is tested however, as the normal sudoku and overlap utilize it over the rows, columns, and boxes as an optimization. Since each of these sums to 45, I found adding this constraint greatly improves performance by giving the heuristics more information to work with.

Agent Performance
For runs of the algorithm using every optimization method, I was not able to find a normal sudoku that couldn’t be solved within around 10ms, and the overlapping sudoku within 100ms. I did include a normal sudoku that was claimed to be the hardest possible sudoku according to a particular forum, and it had no problem computing it.

Sudoku num	1	2	3	4	5	6	7	8	9	10
Runtime (ms)	3	4	4	5	4	5	6	4	6	7
Recursive calls	125	102	120	81	100	135	431	81	335	446

Figure 1: performance of normal sudoku using all optimizations

Figure 1 shows the runtime and number of recursive calls of backtrack for each puzzle. Looking at puzzle 4 and 8, there are only 81 variables so this puzzle was solved perfectly. Contrast this with puzzle 7 and 10, which took 431 and 446 tries respectively. It seems these puzzles forced a guess early on that was incorrect. The runtime of these puzzles seems strongly correlated with the number of recursive calls, but this could change given which optimizations we are choosing.

Overlap num	1	2	3	4	5	6	7	8	9	10
Runtime (ms)	3	4	4	5	4	5	6	4	6	7
Recursive calls	171	171	171	171	171	171	171	171	171	171

Figure 2: performance of overlapping sudoku using all optimizations

Figure 1 and 2 results are stored under archive_log.txt in the project directory.

Addressing figure 2, clearly I need a method to find a more rigorous overlap sudoku, or perhaps overlapping sudoku with unique solutions are trivially easy. Every overlap was able to be solved in 171 moves, which is the number of variables in the CSP it generates. Thus at every step it found the correct value for some variable.

So the solver is very good with the various optimizations, but which ones are most important. I chose normal sudoku for this benchmark as overlapping doesn't seem to introduce anything novel. Of the puzzles, I chose number 5 as it was simple, needing 100 out of minimum 81 to solve, but still difficult.

Optimization	None	AC-3	Forward Checking	MRV	LCV
Runtime (s)	120 (Timeout)	0.011	120 (Timeout)	0.04	120 (Timeout)
Recursive Calls	42,686,152	3519	24,909,376	8543	8,916,606

These are interesting results but not surprising. Looking at the correct solution, the first value is a 9 and the partial assignment returned with no optimizations has a 1. Given that it will choose this variable first, the fact that the first root value has not been evaluated completely goes to show how long it would take, 10 to 100 times longer is reasonable. Considering the 3

optimizations that timed out, we can see how they affect performance by analyzing the number of states evaluated in 120 seconds. With the benchmark being 42.7 million, we see forward checking reduces this number to 25 million and LCV even further to 8.9 million. These are both dependant on the number of constraints that affect a given variable, and we can see that forward checking is much more efficient. Although the runtime of both is limited by the number of constraints, forward checking will actually reduce the domain for part of the search, allowing false successor states to be pruned before a recursive call, where LCV only hopes to help pick the best successor state first.

AC-3 has shown in this instance to be the dominant optimization. This is because it removes domain values decreasing the branching factor vastly. Even though the algorithm is a plain DFS, you can see the drastic effect this makes in comparison to no optimizations. This is not surprising, as the branching factor is the dominant runtime constraint on these sorts of problems. The Backtracking with MRV was also able to solve the puzzle in a comparable amount of time that the AC-3 enabled. Although it doesn't reduce the possible branches the algorithm could follow, it will find variables that can only take 1 value first, thus removing an entire layer of the search tree. Considering the worst case with DFS would evaluate 9 values of the initial variable, finding a forced value could remove 8 of 9 branches and therefore could divide the successor state space by 9. This is of course only if there is a forced value, but the math holds true for finding a variable forced to 2 values. Thus it doesn't reduce the domains, but finds the values that are most likely to be correct, reducing the branching factor.

What happens when we mix the optimizations together? For this next experiment I ran every pair of optimizations across puzzle 4 and 5 to test how they interact, and how this interaction changes when the puzzle gets slightly harder.

Optimizations	AC-3 & FC	AC-3 & MRV	AC-3 & LCV	FC & MRV	FC & LCV	MRV & LCV
P4 Runtime	0.013	0.004	0.008	0.000	120 (Timeout)	0.156
P4 Recursive Calls	2,194	101	801	94	18,910,888	8,543
P5 Runtime (s)	120 (Timeout)	0.056	120 (Timeout)	0.001	120 (Timeout)	0.493
P5 Recursive Calls	28,542,667	16,222	15,961,688	132	18652477	27,135

The best was a combination of Forward Checking and MRV. This is surprising because as Forward Checking was not as useful as AC-3 in the previous experiment, but something about the FC and MRV is better than AC-3 and MRV. Interestingly, while the two were comparable for puzzle 4, puzzle 5 is two scales of magnitude larger in terms of recursive calls. This shows that

Forward checking is much better at reducing the branching factor than AC-3 can do, which makes sense as AC-3 is performed once at the start, whereas forward checking reduces the domains throughout. Thus as depth increases, AC-3 can still get exponentially larger where forward checking can prevent this better.

For puzzle 5, the combination of AC-3 and FC timed out and reached 28 million states. Both of these methods utilize domain reduction, yet still leads to exponential increase in explored states. Given a trivial puzzle, domain reduction can lead directly to the solution. However, for a sufficiently complex puzzle reducing the domains will still yield a tree with exponential size compared to its depth. Thus while it could solve simple puzzles like P4, P5 still explodes, as its choice of variable/value ordering is uninformed.

If we are to compare MRV and LCV, the AC-3 & LCV experiment and the AC-3 & MRV experiment illustrates this well. While neither of them timed out for P4, LCV timed out for P5 with 15 million states compared to the 16 thousand that MRV needed to find the solution. This is because while LCV will help you choose paths that are less likely to be inconsistent, therefore causing a backtrack, MRV will actually choose to expand the tree where there are less branches to choose from. Based on this, we might imagine that MRV and LCV could work well together by both choosing to expand where there are few values and choosing the best potential value to try first. Seeing the last column, this conclusion seems to be correct. While it isn't very good compared to the domain reducing methods, it's very good considering it doesn't prune the search space at all, rather just making optimal decisions at every path. Considering many other combinations led to an exponential increase in complexity from P4 to P5, this method only had a 3x increase in states to solve, which is impressive.

This gives a good indication of each optimization method and how they interact. I don't think comparing 3 optimization methods together really tells us much more about the methods, as the interactions would be unclear, so I will conclude here.