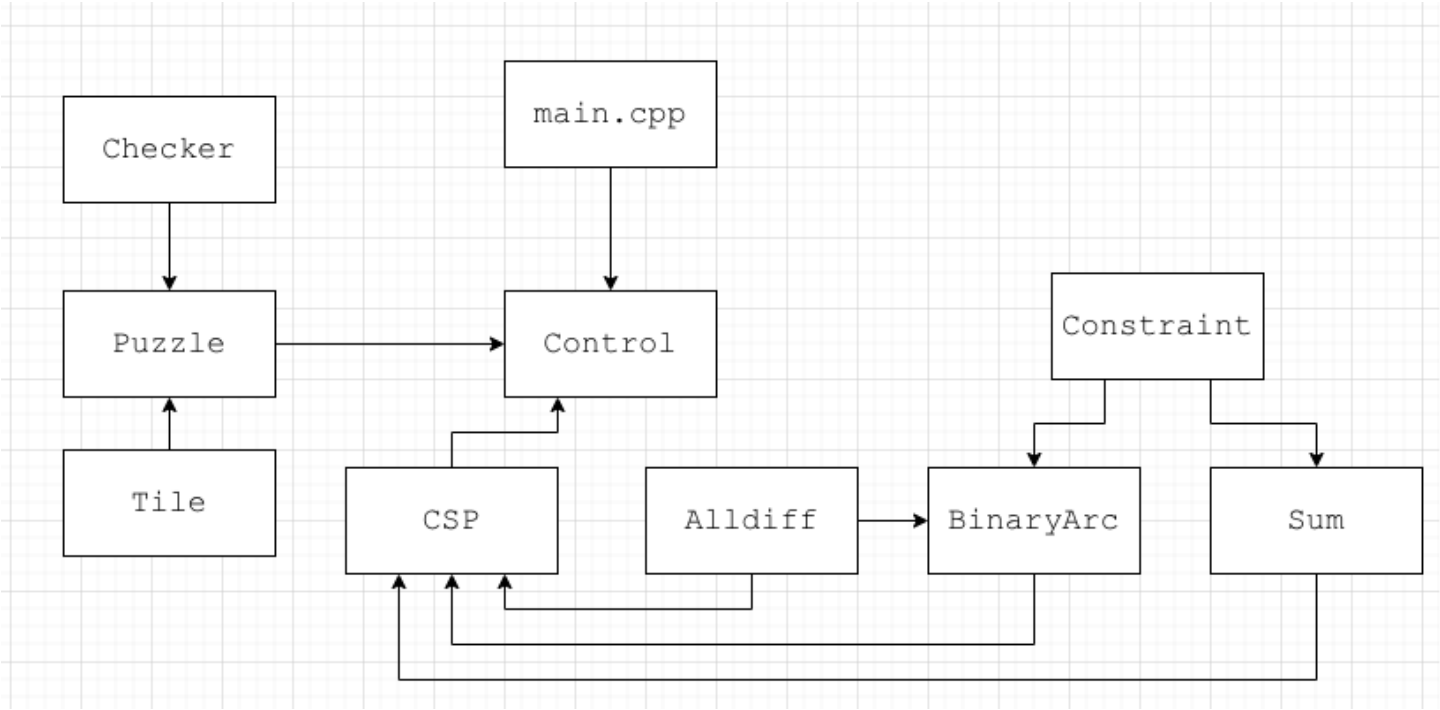
Analysis of a CSP Sudoku Solver

🎀 Claire Liu, CS 420: Artificial Intelligence, Lafayette College, Fall 2022 🎀

👩‍💻Program Design

The Sudoku Solver agent was implemented in C++.
The project code followed the following architecture



- **main.cpp** is the file that drives the experiments
- **Control** is the class that represents the agent. It takes the configuration passed by main.cpp and runs the experiment accordingly. Backtracking search, AC-3, minimum remaining values (MRV) heuristic, least constraining value (LCV) heuristic, and forward checking all live here.
- **Puzzle** represents the sudoku board. It is made up of Tiles.
- **Tiles** represent each cell on a sudoku board. Each tile has a string id, a num (the number, if assigned, 0, if not), and a possible domain.
- **CSP** is the constraint map. There is a key, value pair for every tile. The string id for the tiles are the keys and the values are a vector of Constraints that the key tile is linked to.
- **Constraint** is a virtual interface
    - **BinaryArc** completes the virtual Constraint interface. The Binary Arc represents a not equals constraint on Tiles.
    - **Sum** also can complete the virtual Constraint interface. The Sum represents a sum constraint on Tiles as seen in Killer sudoku
- **Alldiff** represents the classic sudoku constraint of the tiles of the row, cols, and boxes, being unique, respectively. It does not complete the Constraint interface because the Alldiffs are converted to BinaryArcs by the CSP
- **Checker** ensures that the solution proposed by the agent meets the constraints on a standard and overlapping sudoku puzzles, or matches the given solution for the killer sudoku puzzles.

🛠️ User Manual

To run the program with the current configuration listed in main.cpp, simply run `make` from the project root. There are also 3 make commands that run the three puzzle types with relatively fast configurations

```
make standard      // runs the 10 standard sudoku puzzles
make overlap            // runs the 10 overlap sudoku puzzles
make killer             // runs the 10 killer sudoku puzzles
```

To configure the experiment in main.cpp you can manipulate the following booleans and enum to specify how you want to run the project

```
 bool useAc3 = true;
 bool useMinRemainingValues = true;
 bool useLeastConstrainingValues = true;
 bool useForwardChecking = false;
 puzzleType type = OVERLAP; // either STANDARD, OVERLAP, or KILLER
```

To clean the project, run `make clean`
To add new puzzles, add a txt file to the ./Sudoku directory.
A standard puzzle was encoded like this, where each tile was separated by spaces and each row in the puzzle has its own row too.

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

Tripledoku puzzles were also encoded in a visually similar way to how the puzzle looks, where - was used to indicate buffer space.

```
2 4 1 9 3 5 0 0 7 - - - - - -
0 7 8 6 2 0 9 0 4 - - - - - -
0 6 0 4 7 8 2 1 0 - - - - - -
6 2 0 7 1 4 5 8 9 0 6 0 - - -
9 0 4 5 0 2 0 7 1 0 8 0 - - -
7 0 5 8 9 3 4 2 6 0 0 0 - - -
8 0 2 0 4 0 7 9 0 0 0 2 0 4 0
0 9 6 2 0 0 1 0 0 0 9 0 5 7 2
0 5 0 3 8 9 6 4 2 5 7 1 3 8 9
- - - 6 2 1 9 5 0 3 4 0 0 2 1
```

```
- - - 9 3 8 2 6 4 0 0 7 9 0 8
- - - 4 7 5 0 1 3 9 2 0 4 5 7
- - - - - - 5 7 0 2 6 0 8 0 4
- - - - - - 4 8 0 7 1 5 2 6 3
- - - - - - 0 0 0 4 8 9 7 1 5
```

Killer sudoku puzzles were encoded like the standard puzzle, except the txt file looked like this:

```
// puzzle
// empty line
// puzzle solution
// empty line
// one line for each sum cage
```

The cages were encoded as the tile numbers, 0…80. These were later translated into coordinates for the tileId by the agent.

Data is outputted to a running log file for each puzzle type in the log folder.

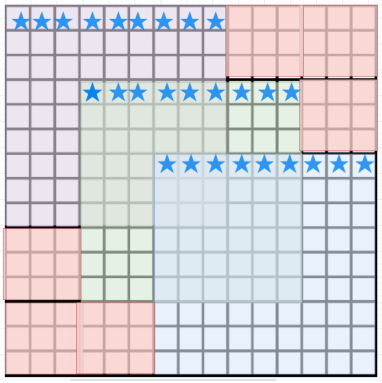
🍙 Puzzle Encoding and Generic Design of CSP Agent

The CSP Agent is generic through the use of the Puzzle class and the Constraint interface.

Standard Sudoku
Standard Sudoku was of course supported. The Puzzle grid was a 9x9 vector and the constraints could be represented as binary arcs. The constraints are the same every time for every puzzle. Tiles that were assigned from the beginning were encoded as that number, and if there was no assignment, then the number was encoded as 0.

TripleDoku - Generic Puzzle Class and Constraint adder
To support TripleDoku, the Puzzle class had to support 15x15 vectors as well. So the size of the Puzzles was dynamic in the Puzzle class. Representing a TripleDoku puzzle as a 15x15 board meant that there was some wasted space (see in the diagram below in orangey red). These dead zones were encoded as -1.

The TripleDoku puzzle was just a larger standard puzzle, but with more Binary not equals constraints. The BinaryArcs were added in a generic way. The Alldiffs class contained 3 generic functions, addBox, addCol, and addRow, where the top and/or left most tiles in the sequence of 9 tiles was passed as an argument. For example, if all the tiles with blue stars were passed to the addCol, then we would get all the alldiff constraints needed for the columns.



Killer Sudoku - Constraint interface

The Puzzle class is able to handle standard and killer puzzles easily because those are simply encoded as a 9x9 vector array.

A new constraint had to be added for the Killer sudoku to represent the cages. This was the Sum Constraint. For the CSP Map to support this, an abstract Constraint class was created. It was an interface that the BinaryArc and the Sum classes completed. This allowed Sums and BinaryArcs to be added to the generic Constraint map. Virtual functions implemented all the capabilities that a constraint needed including:

- A proposeAssignment() function that was used to see if an assignment would be consistent
- A revise() function for AC3
- A willChangeDomainsOfOtherTiles() function for ordering the domain values (LCV)
- A removeFrom DomainOfOtherTiles() function for forward checking
- A getTiles() utility

Agent Performance

Per the project description, 10 puzzles of every kind were used to measure the agent's performance. In this section, vanilla means that the backtracking search was used with no improvements like AC-3 preprocessing, minimum remaining value heuristic, least constraining value heuristic, no forward checking.

For each of the puzzle types, we will consider how the agent's performance was in a vanilla scenario, each improvement individually, when all improvements were used except one (these are the complement cases) and when all the improvements are used together.

Performance was measured using the following metrics:

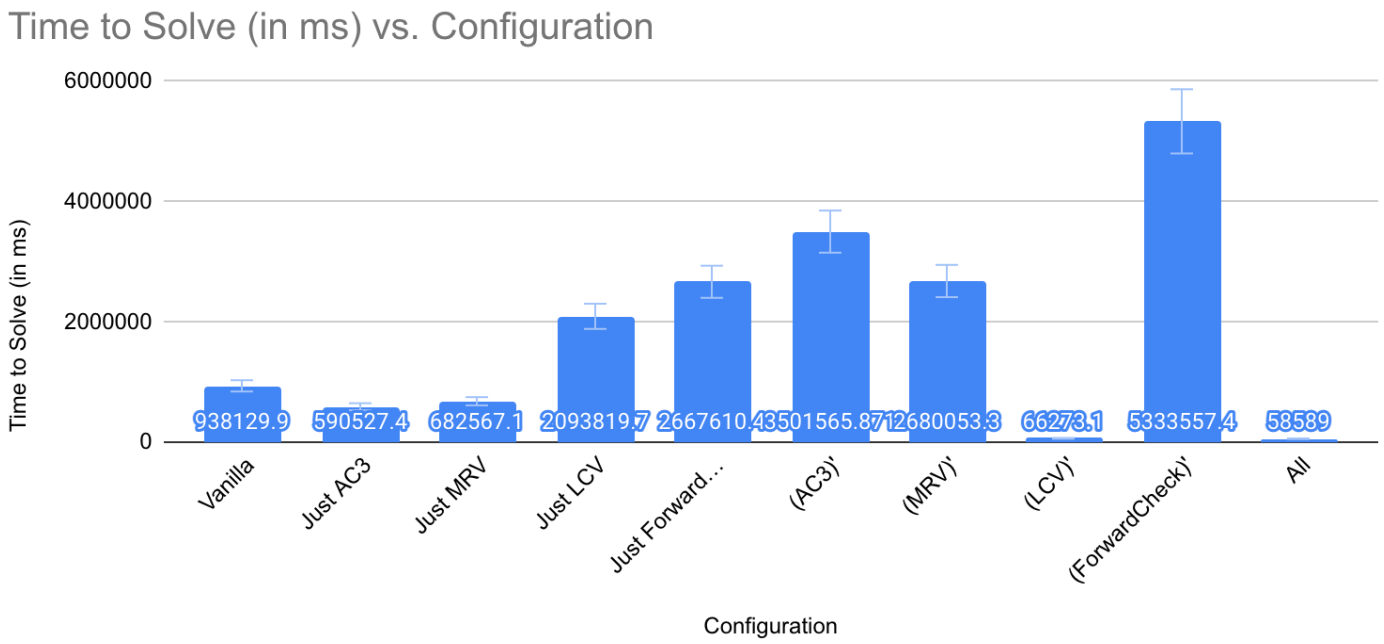
- Just for AC-3: % of tiles unassigned and domain size
- Execution time
- # of times backtrack() was called
- Average depth

Standard Sudoku

AC-3 Analysis

Puzzle #	#Unassigned Tiles Before AC3	#Unassigned Tiles After AC3	% Reduction	Average Domain Size for Unassigned After AC-3
1	50	13	74.00%	1.36364
2	50	18	64.00%	1.62963
3	57	57	0.00%	3.28788
4	53	53	0.00%	3.1129
5	57	57	0.00%	3.37879
6	55	49	10.91%	2.7931
7	56	56	0.00%	3.16923
8	55	55	0.00%	3.09375
9	57	57	0.00%	3.15152
10	52	52	0.00%	3.03279
Average	54.2	46.7	14.89%	2.801323

From the above table, we can see that running AC3 yields significant improvements to the scope of the problem. First, there is an average of 15% reduction in the number of unassigned tiles after running AC3. In puzzle 1, AC3 was able to reduce the number of unassigned tiles by 74%. Other times, AC3 may not be able to reduce the number of tiles at all. Even if it's not able to reduce the number of unassigned tiles, AC-3 still significantly reduces the domain of unassigned tiles. Without AC-3 the domain size at the start for an unassigned tile is 9. However, AC-3 on average was able to reduce it to about 2.8 tiles! Amazing! With a reduction in domain, we significantly decrease the branching factor of our backtrace search, yielding a more focused search, as we shall see.

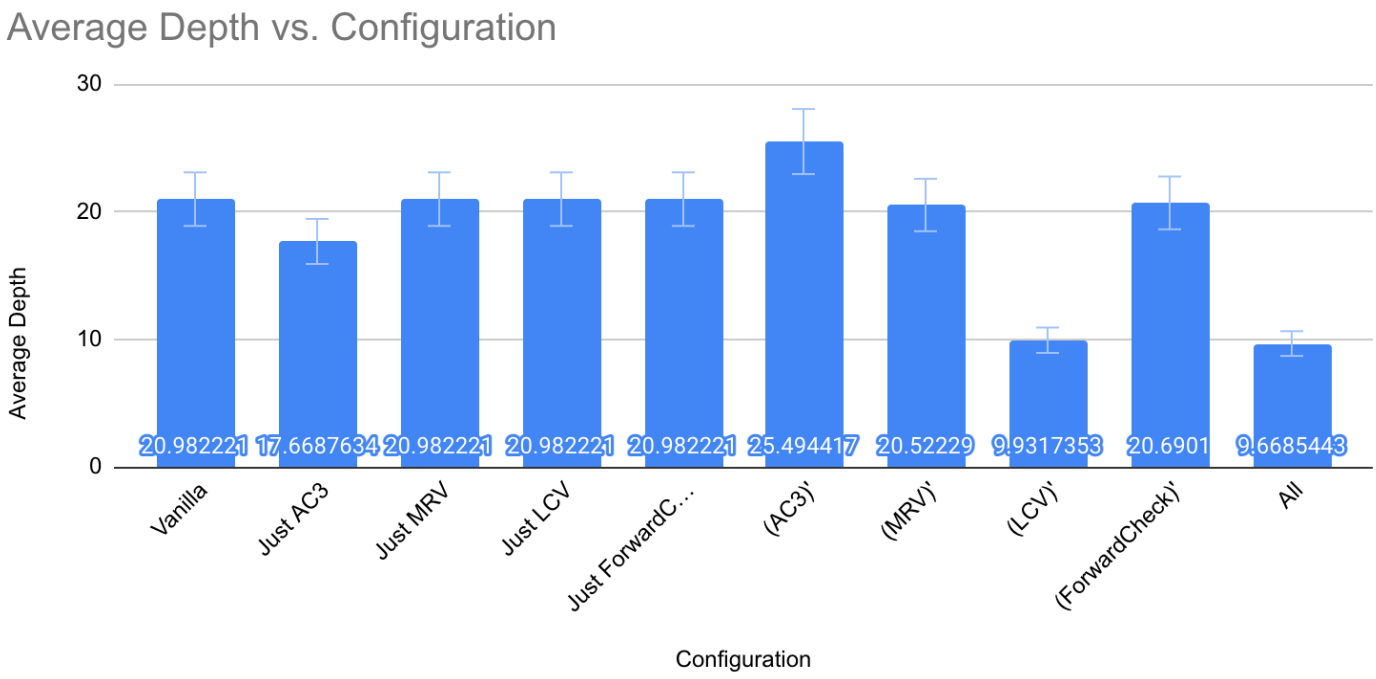


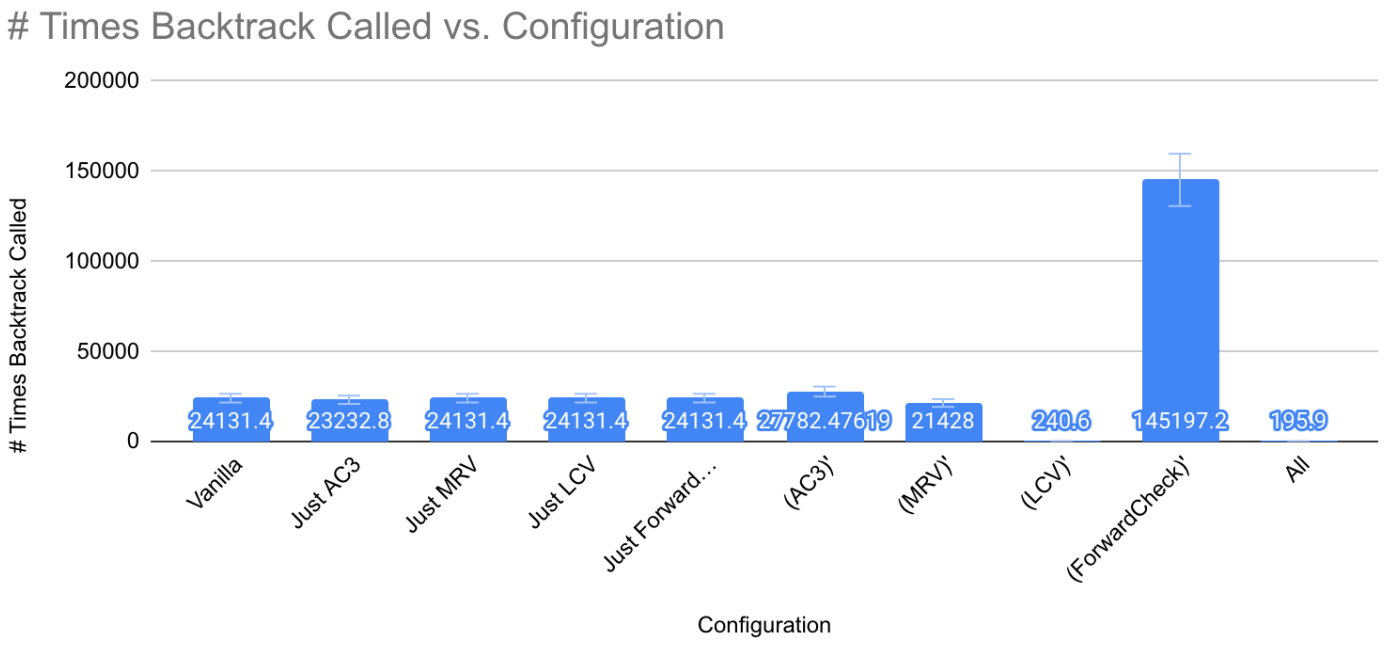
In the above graph, we can see that using all of the improvements together significantly reduces the runtime.

Using just AC3 and just MRV also reduces the runtime significantly. This makes sense because AC-3's execution time is not counted in the time to solve, only backtrack() is. Ordering the variables to consider by MRV rather than just considering them linearly also led to reduction in run time, showing that the MRV heuristic may be effective in directions the search's scope. In the (AC3)' and (MRV)' case we see that the improvements from using (AC3) and (MRV) are great relative to the gain we get from (LCV).

Interestingly, using just LCV and just forward checking lead to increased runtime. This could be from the overhead from each of these improvements. LCV must calculate the number of domains that the assignment of a value could change. Forward checking also includes a lot of changing domains and then restoring domains, which could lead to higher runtime.

Also, we notice that when we use everything but forward check, things start to come off the rails a bit. We will explore reasons why the runtime spikes shortly.





Average depth and the number of times backtrack is called is tied closely together. Using all the improvements together reduces the number of times backtrack is called by a factor of 10, but it still must search to the solution depth, so it makes sense that the improvement would appear less there. Using just MRV, just LCV, and just forward checking doesn't appear to yield improvements to the number of times backtrack is called. However, comparing the complements with the All case, we see that not using MRV leads to a higher number of calls to backtrack() and a higher average depth, so MRV does improve the search. In the case of LCV, we can see that there are small improvements when we do use LCV. Most interestingly, we see the case of not using Forward check, the number of calls to backtrack dramatically increases, but the average depth slightly decreases. This means that we are just doing a wider search of the search space since we aren't reducing domains as we go (what we do in forward checking). Therefore, we can see that forward checking is really important to directing the search to the correct area. Without it, the heuristics can lead the agent astray (comparing it with the vanilla version).

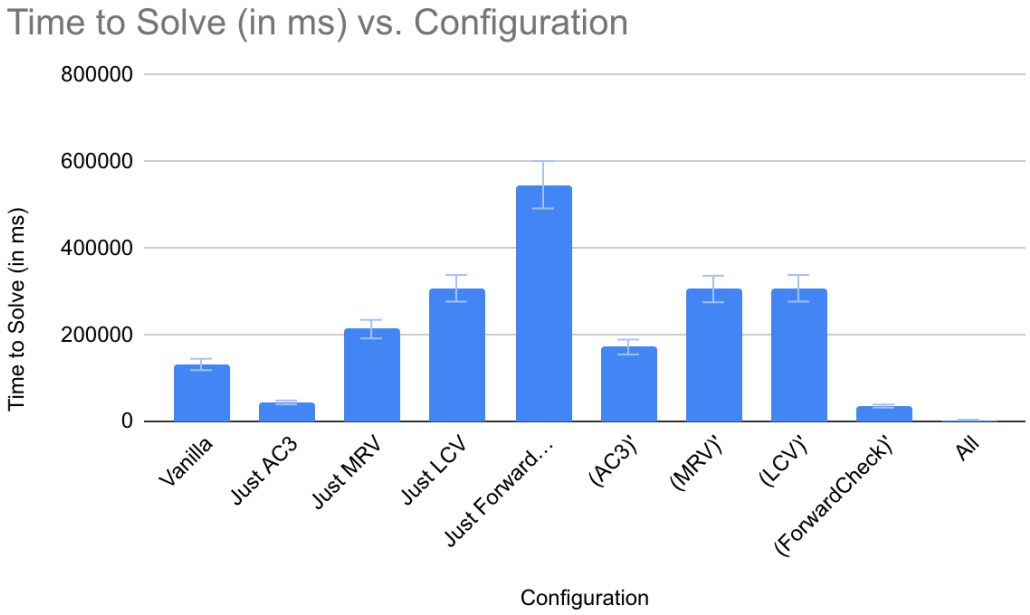
TripleDoku

AC-3 Analysis

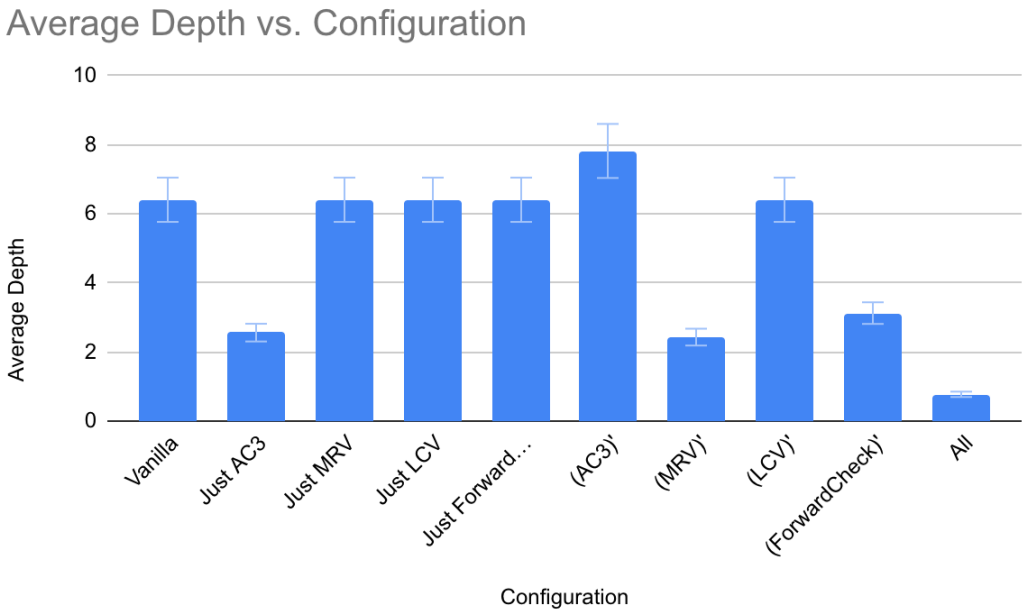
Puzzle #	# Unassigned	# Unassigned after AC3	% Reduction	Average Domain Size After AC3
1	45	0	100.00%	0
2	52	2	96.15%	2
3	46	0	100.00%	0
4	48	0	100.00%	0
5	75	4	94.67%	2
6	74	8	89.19%	2.125

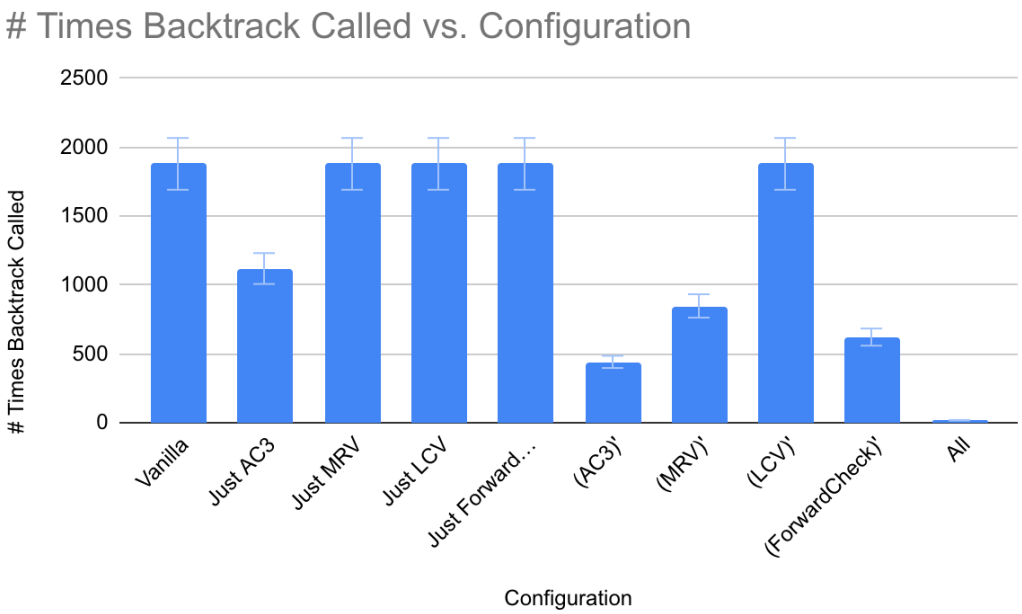
7	73	2	97.26%	2.5
8	72	2	97.22%	2
9	96	29	69.79%	2.68966
10	105	81	22.86%	3.12346
average	68.6	12.8	86.71%	1.643812

In the above table, we can see that AC-3 is amazingly effective at TripleDoku problems. This is because there is a high number of tiles that are already assigned in the TripleDoku problems. The average reduction rate is 86%, which means that AC3 significantly reduces the search space for the backtracking algorithm. At times, no further search is needed, or very few tiles need to be searched. The domain sizes for those tiles are also very small, with about 2 values to try.



In the above graph, we can see that the run times for the vanilla problems are actually quite good. When just MRV, just LCV, and Just forward checking are used we actually see higher runtimes. This would be because of the overhead computation for those functions. The overhead computation cost is also seen in the complement cases. Interestingly we see a reduction in the time to solve for using everything but forward check. This is actually the reverse of what we saw for the Standard puzzles. A likely reason is the number of tiles that we had assigned to begin with. In the case of the Standard puzzles, many of the tiles were not assigned at the start. However, in the TripleDoku case, the majority of tiles have been assigned. This means that the overhead cost of forward checking overshadowed the number of tiles we had to assign.





We can see that the average depth for these puzzles was extremely low. In any case where we used AC-3, the average depth was close to 1 since we didn't have to use the backtracking search in some of the puzzles, pulling the average down. We see similar results in the backtracking case.

Because we had few assignments to work with in this puzzle type, it's possible that results are a bit skewed to the individual puzzles. However, from the graph, we can see that using just MRV, Just LCV, and just forward checking did not yield many improvements on the number of calls to backtrack, similar to the Standard puzzles. We do see that not using LCV in the (LCV)' case compared to the All case resulted in a higher number of calls to backtrack. This indicates that LCV was pretty effective in improving the search. All also shows improvements compared to (AC3)', (MRV)', and (ForwardCheck)', so it shows that all of these improvements did help the backtracking searching to a certain extent.

Killer Sudoku

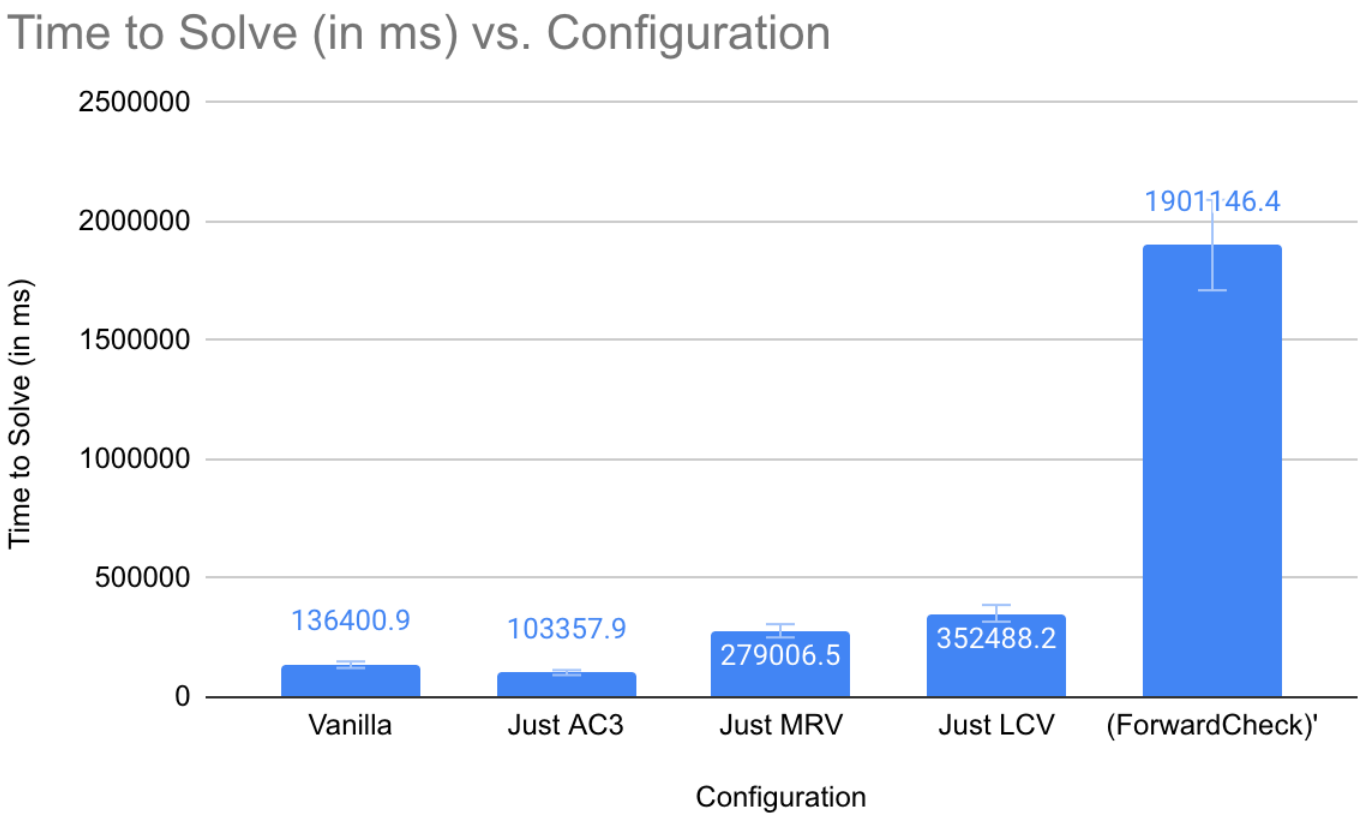
In Killer Sudoku, I ran into some issues with implementing forward checking and then ran out of time to debug all the changes 😢 (I think it may have been something to do with undoing changes with the sum constraints and binary arcs conflicting?)

Because of this, we only consider the cases of vanilla, just ac3, just Mrv, and (forward check)'.

AC-3 Analysis

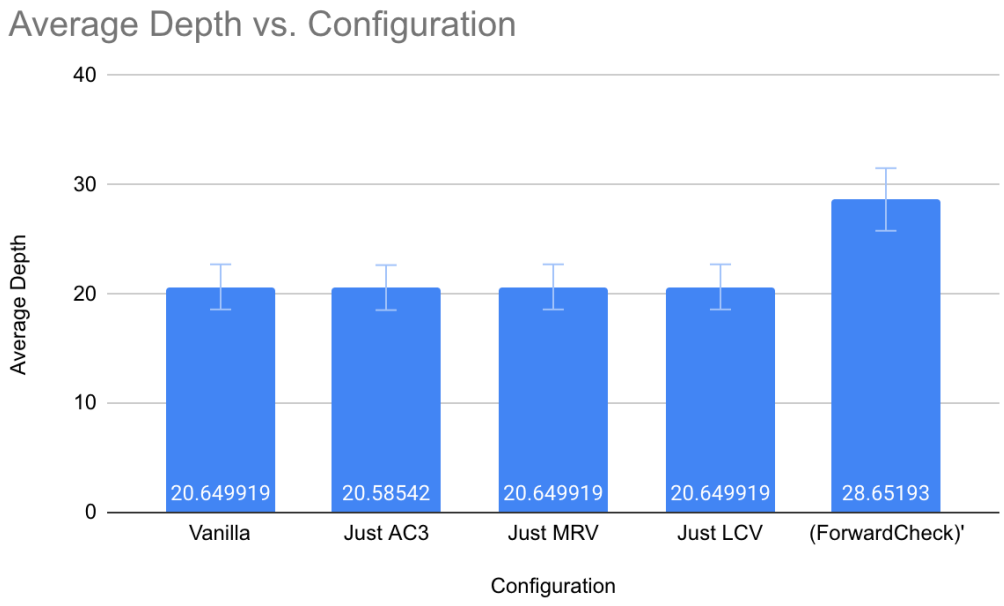
Puzzle #	# Unassigned	# Unassigned after AC3	% Reduction	Average Domain Size After AC3
1	71	65	8.45%	5.36923
2	71	68	4.23%	6.25
3	71	66	7.04%	5.92424
4	71	64	9.86%	5.14062
5	71	63	11.27%	4.98413
6	71	65	8.45%	5.58462
7	71	65	8.45%	5.32308
8	71	69	2.82%	6.23188
9	71	65	8.45%	5.4
10	71	65	8.45%	5.33846
average	71	65.5	7.75%	5.554626

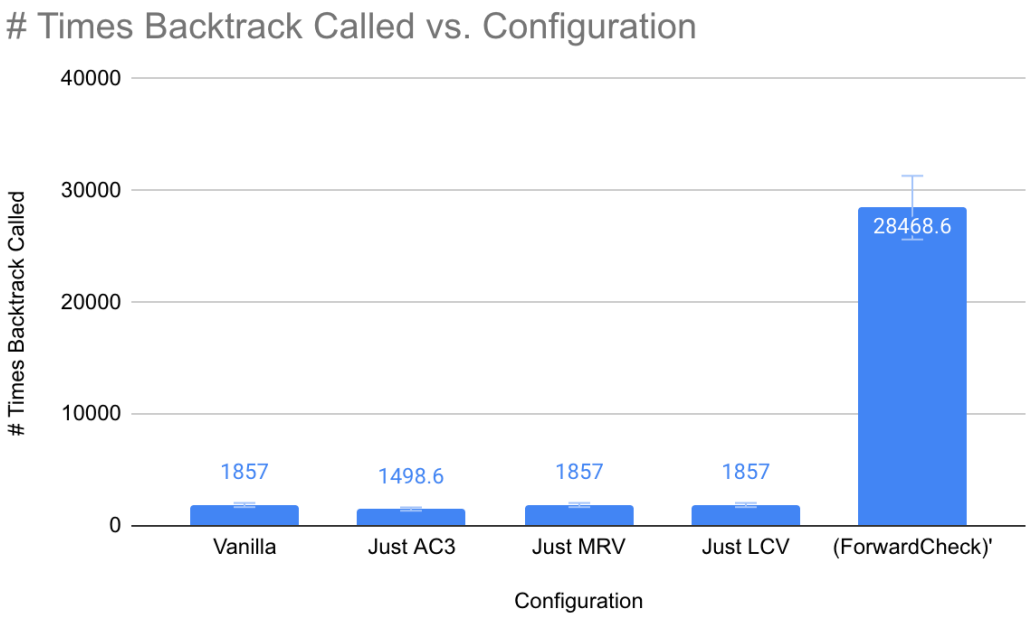
The Killer boards tended to be pretty sparse to begin with, so it makes sense that AC3 was not able to make significant improvements to the board. There was an average reduction of 8% to the number of tiles. Domain sizes were slightly improved by the ac-3 search.



We can see that the vanilla case was actually pretty good, compared to when we tried to use some of the improvements. AC-3 led to some improvements. MRV and LCV had slightly higher run times, possibly because of the overhead computation costs of checking the sum constraints, which are more involved than the binary arcs.

Similar to the standard case, we saw that with all the improvements except for the forward check, the runtime became quite high. We can presume that if forward check was correctly implemented, then it could be used to guide the agent to better answers in an All case.





From the above graphs, we can see that the Just AC3 case ran the best. MRV and LCV did not seem to reduce the search space. This makes sense since they were always operating on the assumption that a domain was {1,..9} and the domains were never whittled down by the forward check or the AC3. For everything but forward check case (ForwardCheck)' we can see that without Forward checking, the heuristics tended to misguide the agents. The heuristics for the sum constraints could have improved implementation; usually the heuristic was only used if the number of unassigned tiles within a constraint was 0, 1, or 2. Higher ones were not considered. This could have been fixed by implementing the heuristics more precisely or reducing sum constraints to binary constraints upfront which would have led to higher precision and faster runtime.

🍡 Summary of Results

Overall, we saw how the different improvements interacted with one another. In the case of Standard, Overlapping, and in an ideal case for Killer, the most efficient searches would happen when all the improvements were used together.

AC-3 was a huge improvement regardless of initial tile configuration. It reduced domains across the board and was able to assign at least a portion of the unassigned tiles on the board. This reduced the search space and at times was able to completely solve the puzzle.

MRV was generally effective in pruning the search tree, reducing the number of calls to backtrack and hence the depth of the search tree. It was relatively more effective in the standard case where there were fewer tiles assigned once backtracking was called. There was some overhead in using MRV, since it called sort() but this was balanced out if the number of tiles that were unassigned was high.

LCV was generally effective in reducing the branching factor of the puzzle. The gains were relatively small in the standard puzzles, likely because the gains from the other improvements overshadowed it. However in the overlapping case, we saw that when there were relatively few tiles to assign (so MRV and forward checking didn't have much to do), then the LCV had a chance to shine through and lead to significant gains. If the domains were not whittled down upfront by ac-3 or during the search by forward checking, LCV tended to have

very little to work with and make decisions off of, leading to an overhead computation cost with few performance gains.

Forward Checking was really important in directing the search since it reduced the domains during the search. When the other heuristics were used and forward checking was not used, then the agent was often misguided, leading to poorer performance. Therefore, if heuristics are used, then inference plays a key role in keeping the search on track so that the heuristics have better data to make choices off of.

In conclusion, sudoku puzzles of different variants can be interpreted as a CSP and solved quickly by CSP agents. Performance can be used with heuristics and inference.

💡 References

Russell and Norvig. *Artificial Intelligence: A Modern Approach, 4th Edition*. Hoboken, NJ. Pearson Education Inc, 2021.