Richard Ying A10954259

CSE150 PA3 Report

Design Description

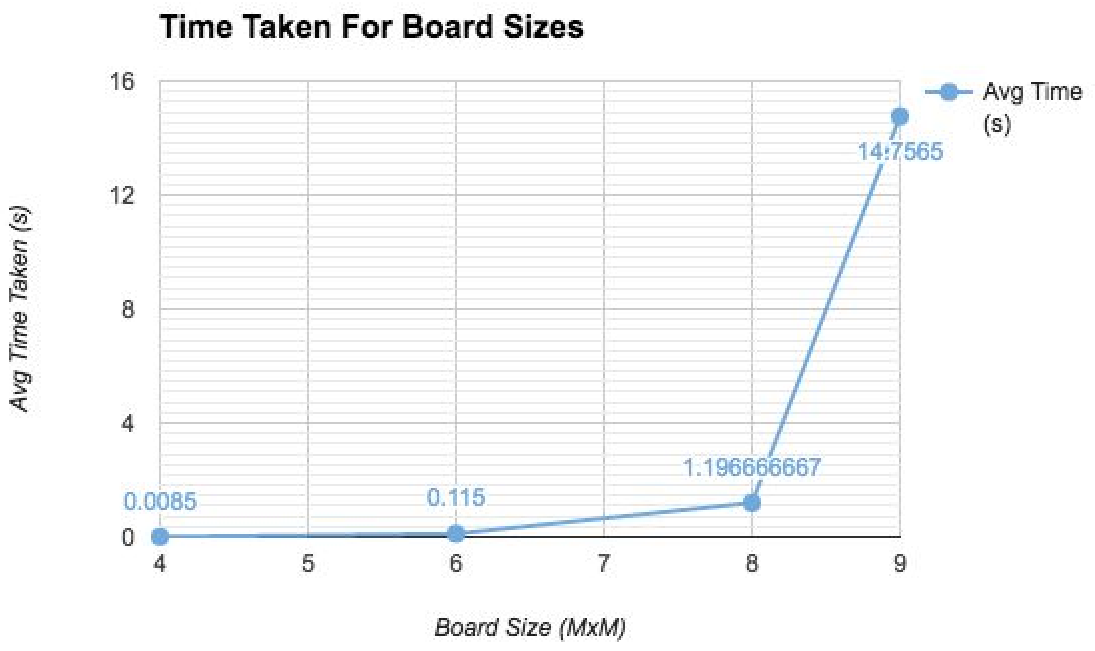
1. Problem one was a simple implementation which returns true when all variables in the CSP have been assigned. This was done by accessing each variable in the csp by a for loop iterating through v in csp.variables. If it encounters a variable that is not assigned, it returns false. Otherwise all the variables are assigned and it returns true.
2. The consistency check was a little more complex, but still fairly straightforward. It first loops through all the constraints for a particular variable input. It then checks whether the two variables in each binary constraint is assigned, as specified in the project description. If both are assigned, then it proceeds to check whether the constraint is satisfied for both values. If it encounters a variable that is not satisfied, it returns false. Otherwise all the constraints are satisfied and it returns true.
3. This was a simple implementation of the basic backtracking method for a given csp. From the imported solutions of problems 1 and 2, I was able to simply follow the pseudocode given in the professor's lecture slides in implementing the backtracking algorithm. It first returns the base case True for a complete csp. If incomplete, it selects the next unassigned variable and for each value in that domain, it checks if each value is consistent with the assignment. If so, it "begins a transaction" as specified by the assignment hint, which saves the current state before modifying changes by setting the value as the assignment and checking the inference, then calling backtrack recursively, which again checks if it's complete and consistent. If not, then the assignments are removed since the state of begin_transaction and the next values are attempted. Although it utilizes a rudimentary select_unassigned_variable method and does not use an inference method, I implemented them anyway, following the pseudocode. This runs in about 4 seconds for the provided tests.
4. This was an implementation of the AC-3 method for a given csp, and optionally an arcs parameter. I was also able to follow the pseudocode given in the professor's lecture slides in implementing the AC-3 algorithm, but I had to write a helper function described in the pseudocode as "if no value in the domain allows x,y to satisfy the constraint xi and xj". Given the queue of arcs, I iterated through the queue to get variables xi and xj, calling the revise method, which attempts to satisfy the constraint between xi and xj by revising a value y, which calls no_satisfy to check for each constraint, in xi, whether a variable y satisfies the constraint between xi and xj. If so, then AC-3 proceeds to check the length of the domain and then iterates through xi's neighbors, adding them to the queue, and finally returning True if the check succeeds.

5. This problem was the most difficult for me, having to implement the heuristics for MRV, degree, and LCV. In select_unassigned_variables, I started with the MRV, which has a very large initial smallest value. I iterate through the csp variables checking only the unassigned ones, saving the new smallest value of the variable with the fewest legal values, or the variable with the smallest domain. It then sets that as the current variable, returning it in the end. If there is a tie such that the length of the checked variable's domain is the same as the current smallest, I implemented the degree heuristic. This compares the constraints of both variables, incrementing them for each matching constraint. If the checked variable has more constraints than the current variable, then the new variable becomes the current variable as defined by the degree heuristic, and returns that.
6. This problem entailed combining all previous parts in creating a faster solver by augmenting the basic backtracking algorithm. Since most of the framework was already provided, I only had to insert my solutions to part 3, 4, and 5. Now the rudimentary select_unassigned_variables and inference methods aforementioned in part 3 was replaced by the MRV and degree heuristic I wrote for part 5, and the useless inference check was replaced by the ac3 check. The augmented algorithm halved the time it took in part 3, requiring only about 2 seconds to compute the provided tests.

Solver Results

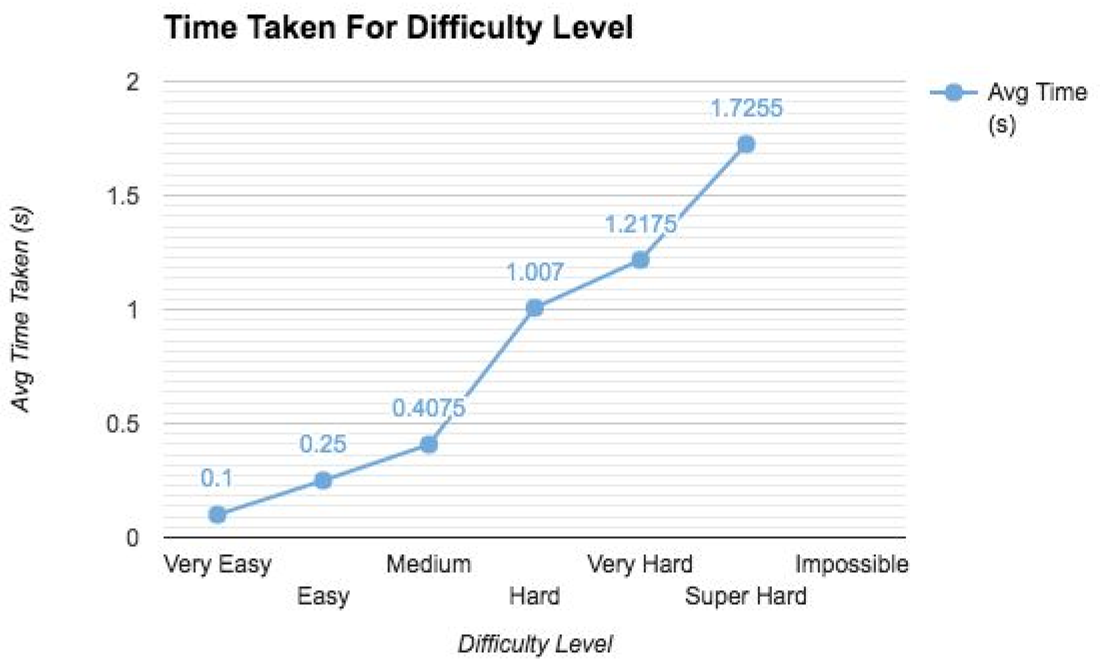
Board Size	Total Time (s)	Num of Tests	Avg Time (s)
4	0.017	2	0.0085
6	0.69	6	0.115
8	7.18	6	1.196666667
9	59.026	4	14.7565

For the tests, I used various boards from the provided menneske.no links, varying from very easy to super hard. Categorizing my tests by board size, it seems like the time taken to solve the puzzles of varying difficult goes up exponentially as the board size increases linearly. This is consistent with the AC-3 time complexity of O(n^2*d^3)where n is the number of nodes and d is the max number of domain values, since increasing the board size is directly correlated with an increase of nodes.



Difficulty Level	Total Time (s)	Num of Tests	Avg Time (s)
Very Easy	0.2	2	0.1
Easy	0.5	2	0.25
Medium	0.815	2	0.4075
Hard	2.014	2	1.007
Very Hard	2.435	2	1.2175
Super Hard	3.451	2	1.7255
Impossible	x	1	x

I again used various tests from the provided menneske.no links, timing each difficulty with a 6x6 board and 8x8 board (except the impossible difficulty). Categorizing the tests by difficulty level, the relationship between difficulty level and time taken to solve the puzzle is approximately linear. Because “difficulty level” is a very subjective and qualitative categorization, I presume that the boards provided on menneske are categorized based on the time taken to solve the games using a similar algorithm. This corresponds with the traditional linear definition of difficulty preferred by players in games.



Heuristics Analysis

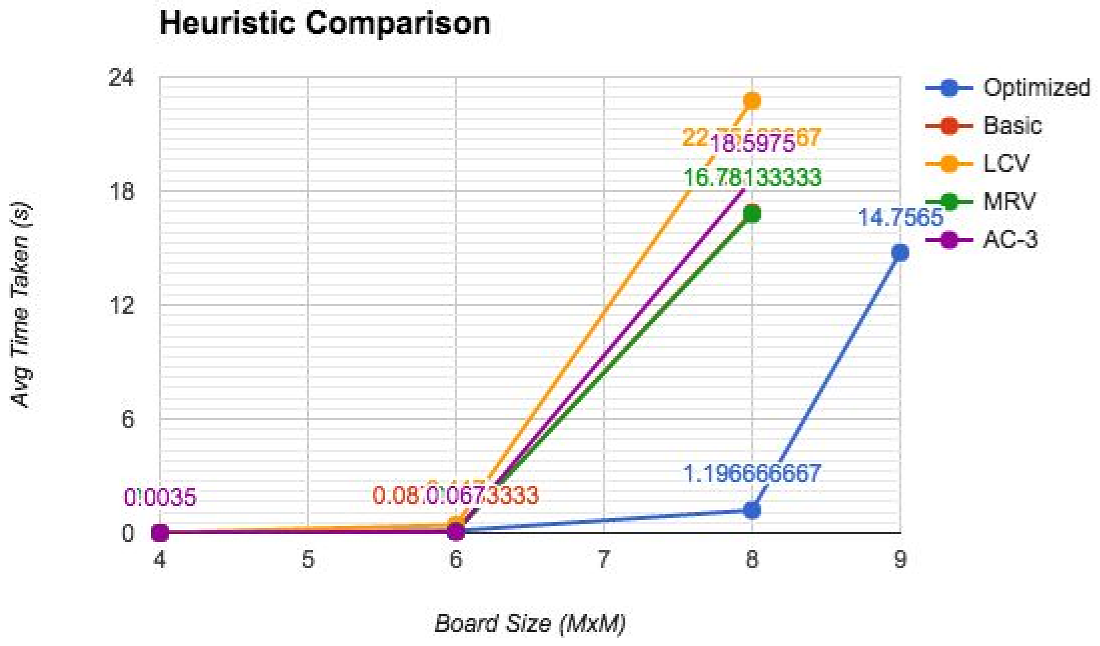
Basic Backtracking Solver

Board Size	Total Time (s)	Num of Tests	Avg Time (s)
4	0.011	2	0.0055
6	0.524	6	0.08733333333
8	101.152	6	16.85866667
9	x	4	x

Compared to the optimized solver, the basic solver also takes exponentially increasing time for increasing board sizes, but much longer for the same board size when it's greater than 6. Surprisingly, it takes a shorter average time for board sizes 6 and smaller, presumably because the extra checks of heuristics in such small CSPs outweigh the benefits of optimal selection they provide.

Table of all average times taken for each implemented heuristic backtracking algorithm

Board Size	Optimized	Basic	LCV	MRV	AC-3
4	0.0085	0.0055	0.0145	0.004	0.0035
6	0.115	0.08733333333	0.417	0.07	0.067
8	1.196666667	16.85866667	22.75166667	16.78133333	18.5975
9	14.7565	x	x	x	x
Notes	fast	fast for small	slow	similar to basic	similar to basic



X marks the algorithms that take too long (over 300 seconds). As noted and shown, each individual heuristic is not much faster than the basic algorithm. The inference heuristic AC-3 probably does not work well when the given ordered domain values aren't heuristically selected, since the inference has to look through poor choices. Similarly, the ordered domain values for a particular variable may not be a good choice since it lacks a heuristic for selecting the next unassigned variable. Likewise, the variable selection heuristic may choose a good next variable, but its ordered domain may be poor, and the ac-3 inference of the subsequent values in the domain may also be poor. Finally, with the value ordering in the domain heuristic, the algorithm again may choose a poor unassigned variable to order the domain, and lacks an inference heuristic. Thus the data makes sense, since all three individually implemented heuristic algorithms perform similarly to the basic backtracking algorithm. Only when the three heuristics are combined do they vastly outperform the basic algorithm.

Member Contribution
Since I worked alone, I learned everything aforementioned in the report as well as the specific algorithms for solving games with CSPs, including the basic backtracking algorithm, the AC-3 algorithm, the MRV and Degree heuristics, and domain value ordering.