Crowded Chess Board SAT Solver

Drake Moore, Justin Bian, Matthew Chan

General Problem:

Chess is a very popular game that has been around for hundreds of years and many puzzles have been derived from the game and its mechanics. One such problem that has ties to the ancient game is the crowded chessboard problem. In this problem, one is tasked with finding an optimal solution for placing a maximum number of pieces (Rooks, Bishops, Queens, Knights) onto a regular chess board in a way that no piece of some type can attack another piece of the same type. In the general definition of the problem, a regular chess board is used (8 x 8) and the maximum number of pieces that can be used is 51 with 8 Rooks, 14 Bishops, 8 Queens, and 21 Knights.

In this paper we consider the crowded chessboard problem and try to determine all the possible solutions to the problem. In order to develop a method to actually solve the problem, we define constraints to the problem in order to obtain solutions in a reasonable amount of time. The constraints were that there are N number of Rooks, N Queens, and 2N-2 Bishops that must be used to obtain a solution to the crowded chessboard problem. In order to achieve an optimal solution, the goal was to maximize the amount of Knights used on a board. Because of the constraints we defined we were also able to consider smaller chess board sizes for this paper in order to prove that our way of obtaining all the possible results was indeed correct.

Approach:

The crowded chess board problem is a relatively well known problem with a few documented example solutions. One solution we found was from the following research paper: https://www3.hhu.de/stups/downloads/pdf/crowdedchessboard.pdf. Here, the research paper introduces multiple approaches, at a high level, to generate solutions for the crowded chess board problem including a CLP solution, Scala solution, SMT encoding, and a SAT encoding. The following page: https://github.com/leuschel/crowded-chessboard/blob/master/sat/sat_generator.py is the repository for their SAT encoding solution implementation in Python, however, we were unable to get it to run using the limited resources and documentation provided. Some issues we noticed when looking at the source code for the Python crowded chess board SAT generator included that it did not seem to generate a solution from scratch or even use a SAT solver. For example, they hardcoded the total number of Knights based on the given board size instead of generating all of the possible number of Knight placements and maximizing the total number of valid Knight placements on a board. In contrast, our Python implementation actually uses a SAT solver, determines whether there is a valid crowded chessboard solution for any given board size, generates all valid board layouts from scratch and uses the generated boards to calculate the total number of each piece for any given board size as you will see later in our report.

Our implementation uses a SAT solver to generate all possible valid solutions for any given chess board size. We could not find any other solutions that could also calculate the maximum number valid Knights for any given board size in a way that was similar to how we did. Our program computes the best possible solution with the given constraints that we assigned for ourselves. Using those constraints we could determine how to maximize the Knights that could be placed on the chess board.

Our original proposal's metric in progress was quite binary in terms of measuring success: either we can generate a solution or we cannot. At that point, we were not quite sure whether we would be able to come up with a solution, so we thought that would be our first main goal. After completing research and working on devising a program to provide us with all the optimal solutions, we added an additional goal for success. Our second metric would be looking at the total runtime and efficiency of our SAT solving implementation in comparison to an additional program that we created (also in Python) that did not use SAT to generate valid solutions. This program instead used an algorithmic method to generate all the possible solutions for the specific sized chess board it was working on. If our SAT implementation was more efficient compared to the non-SAT implementation, then that would be considered a success. The higher the efficiency, the more successful our project would be.

Methodology:

To begin, we conducted research on the crowded chessboard problem itself. Determining if there were any proper solutions to the problem and considering how those solutions may have helped us to generate our programs. Additionally, we became more familiar with SAT solvers and their specific use case with Pysat. After familiarizing ourselves with the major concepts of what we were trying to solve we decided that we would want to create a program that could solve the crowded chessboard problem on its own without any reduction or conversion to a satisfiability problem. This program was used to compare efficiency of a traditional algorithm to a SAT solver solution to the problem. After completing our algorithm that would determine all the optimal solutions for a given board size for the crowded chessboard problem, we decided on working on a SAT solver way to find all optimal solutions.

Developing our traditional algorithm to find all optimal solutions proved to be fairly difficult as we did not have a concrete method for creating our algorithm. As we worked, we thought about how to implement an algorithm to find optimal solutions and reviewed the research that we had conducted earlier, we came upon the idea, through research and deconstruction of the problem, of splitting up the chess boards into three different boards, separating our computing into finding the optimal solution for Queens, for Rooks, and for Bishops. We realized that Knights would be the most difficult piece to find an optimal solution because they depended on how the boards were structured because we wanted to maximize the Knights on a given board. Thus, our solution was to construct the three boards for Queens, Rooks, and Bishops and then adding the Knights onto the final constructed board afterwards.

Our algorithm involved building the optimal amount of pieces for each of three board types individually. These boards were built on the premise that two pieces cannot be placed in a setup in which they are either on the same square nor can they attack each other. Then, after

computing those best solutions, combining them such that the optimal solution is an optimal solution for Queens, Rooks, and Bishops combined. After completing this step we focused on finding the optimal solution for Knights on the chess board. Because it was impossible to know what the max number of Knights would be before computing optimal solutions (without previous knowledge) we figured that coming up with all possible solutions for the Knights on the previous boards would be the best way to find what the optimal solutions were. Utilizing the max Queen, Bishop, and Rook boards we were able to find all optimal solutions with any amount of Knights but we had to restrict our algorithm into only providing us the solutions with the maximum number of Knights placed. Thus using this information we removed all the less than optimal Knight solutions from our final list of optimal solutions. We decided to construct this algorithm mainly because we knew that it was possible to form a satisfiability problem out of a crowded chess board problem and we wanted to determine how much more efficient a SAT solver could be in computing correct answers to such a problem over a more traditional algorithm. This algorithm that computed the optimal solutions and the time it took to produce those results became the benchmark that we would compare our SAT solving solution to.

After we had constructed our traditional algorithm to determine the optimal solutions for the crowded chess board problem, we wanted to construct a SAT solving solution that would reduce the problem into a set of boolean expressions in which we could determine satisfiability to provide optimal solutions. Our implementation of the SAT solving method of determining optimal solutions began similarly to our traditional algorithm in which we wanted to construct four different boards for Rooks, Bishops, Queens, and Knights, however in order to actually produce our new method of obtaining the optimal solutions, we realized that each board's square had to be boolean variable in order to properly reduce the crowded chess board problem into a satisfiability problem. With the help of Professor Hemann, we realized that in order to get optimal solutions we would have to apply constraints to these boolean variables. We would have to add clauses to our already constructed boards in order to make it solvable by a SAT solver. Originally we wanted to obtain a solvable satisfiability problem by constructing a large amount of clauses that would limit the amount of pieces and where the pieces could go on each board. At this point we were stuck on how we might add those clauses as expressions to the SAT solver and were a little confused on how those clauses would interact with each other. After contemplating this idea and studying the Pysat documentation more in-depth we discovered a function that would allow us to limit the amount of boolean variables that could be true in a given clause. This was the function that allowed us to ultimately get over the hump that we were stuck at and allow us to complete our SAT solving implementation of providing optimal solutions of the crowded chessboard problem. So utilizing this newly found function we were able to come up with clever solutions to limiting the movement of different pieces. Rather than considering how a piece may attack and considering how to devise a clause that prohibits pieces from being placed at a location (true) and still attacking each other (some more complex clause), we could just limit the amount of pieces in an attack range. In order to implement this solution to our clause problem, we first constructed the Rook board in which we had boolean variables x_1 through $x_{n \times n}$ in which n was the size of the board. In a regular chess board n would be 8. Each of

the n x n variables represented a square on the board, allowing us to construct an entire board of n x n boolean variables.

Determining how to place Rooks was as simple as figuring out if a combination of all variables representing the squares were satisfiable, but limiting the amount of pieces in each row and in each column. The max number of Rooks that could be placed in each row and in each column could be limited to just one. We could use the function that limits the number of true values (add_atmost) to constrain the initial n x n boolean variable combination such that only one variable can be true (there is a piece there) in each row and column. This allowed us to easily determine Rook board by just checking satisfiability of the variables and using the solution provided to us, if a true value is present for boolean variable, there would be a Rook placed on that square in the solution, and a false value would mean that there would not be Rook place on that square. In order to make this encoding of our problem work with our SAT solver we added a clause containing all the possible squares and their boolean variables as our one clause for the SAT solver but each of the row and column conditions were added on as constraints through the add_atmost function in order to constrain that one clause. Additionally, for the Rook board we needed to make sure that in order for a board to be constructed as an optimal solution, there would have to be no Rooks. Because of this constraint, we added another add_atmost condition in which the max amount of false boolean variables was capped at n^2 - n which forces the minimum amount of true values for those boolean variables to be n. This same logic was applied to Bishops and Queens. However because we wanted to construct a board for each type of piece we add an extra n x n boolean variables that must be considered and construct our constraints on these variables rather than the ones created previously for squares on a board for Rooks. Because of this set up for our board representation, we could essentially stack the boards on top of each other allowing us to construct optimal Bishop boards that only work on optimal Rooks boards, and optimal Queen boards that only work on those Bishop boards.

For the Bishops, instead of limiting the amount of Bishops per row, and therefore the amount of true values, we limited the amount of pieces on diagonals. Something that we had to consider for Bishops was which diagonals to consider because we needed to make sure that our constraints represented chess piece moves directly so we had to consider diagonals in both directions. When implementing the Bishops we had to modify the min amount of true values needed in the boolean variables from n^2 - n to n^2 - 2n - 2 The implementation for the Queens was straightforward after we had done the implementation for the Bishops because we had already implemented limiting the amount of pieces to one on all diagonals (Bishops) and limiting the amount of pieces to one on all columns and rows (Rooks). The satisfiability problem for the Queens was easily added by utilizing the functionalities that we previously constructed in the Bishops and Rooks, still limiting the amount of Queens to n^2 - n. When we first implemented the Knights board we believed it would be the same exact method as the other three boards however when we designed our first iteration, we would get boards that would not be at the max capacity. The main reason for this bug was that when we added the constraints to the Knights through the add_atmost function we made it so that only one Knight can exist in every square that can be attacked by the current square, and itself, we are considering. This implementation made it possible for our program to produce less than optimal results because the limits were incorrect.

There could have been situations where two Knights are not attacking each other but the board is only allowed to have one Knight on either of the squares but not both. This was a hard step to overcome as we were unsure of where to go from this implementation of the Knight board. Eventually we figured out that we had to introduce our constraints of having only one square fill on pieces that are mutually attacking. So for each square that the square we are considering can attack, we limit the amount of true values between the attacked square and the current square to be one, essentially isolating the limitation of placements to one move with each constraint. Adding this functionality solved our issue with not having max boards when constructing Knights on top of the additional boards.

After constructing the possible Knights we only had two things left to consider. The first thing we had to consider was adding constraints such that a piece of one type cannot be placed at the same position as a piece of another type. This was easily accomplished by adding a simple constraint limiting each corresponding square in the different boards to just have one true value. The second thing we had to consider was how to determine the max number of Knights to be placed on the board. The design choice that we made in order to implement this utilized satisfiability to our benefit. We just tested all values starting from 0 Knights until we reached an amount of Knights that would no longer make the crowded chess board boolean variables satisfiable. Thus, the number of Knights one less than that would be the max Knights needed to construct the board with the given size. This step was pretty difficult to actually accomplish because it was tough to figure out the approach to finding the max number of Knights without being extremely inefficient. When we reached our solution, we felt that the way we found the max Knights was quite elegant and an interesting application of satisfiability to produce a solution to a problem that arose in the middle of our program creation.

We chose not to implement symmetry removal into our two solutions because we were unable to create a way of removing symmetric solutions in a manner that increased the overall efficiency of our algorithms. This is due to the fact that all symmetric solutions for each type of piece can not immediately be discarded, as they may be a part of a potential solution when merged with the boards of the other three pieces. The only viable solution for removing symmetries that we could implement would be removing symmetric solutions after all solutions have been generated, which would ultimately defeat the purpose of implementing it as a way to increase efficiency. As a result, the number of solutions our algorithm found were all divisible by 8, which indicates there are 8 symmetries for each board (4 rotational and 1 mirrored for each rotation).

Results:

The following is an example output of our algorithm with an input of board size 8.

Figure 1 – One of the 27,856 chess board layouts outputted by our program.

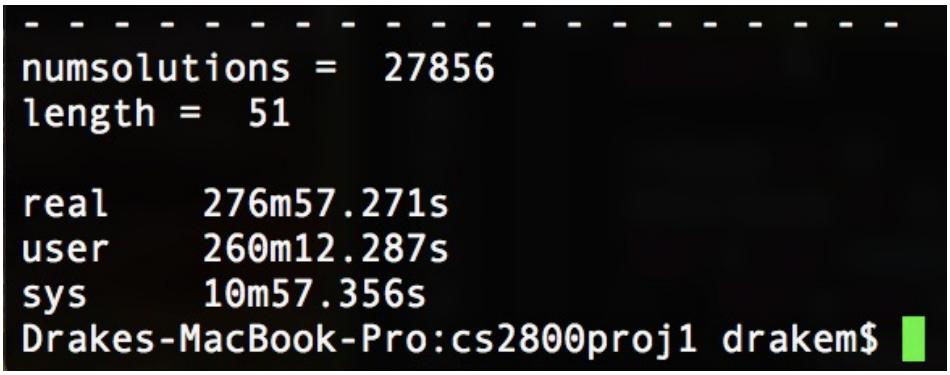

Figure 2 – Output includes the total number of valid solutions and total max number of pieces.

If the program's input is a board size less than 5, there are no valid solutions with the constraints defined in our project, and our programs output a False statement with 0 solutions.

Our original metric for success was just to create a method of finding all possible solutions for the crowded chessboard problem utilizing satisfiability. We were successful in creating a program that uses a SAT solver to generate all valid crowded chess board solutions for any given board size. We also created a similar program that used similar logic to generate all valid solutions but without using SAT solver which gives us an opportunity to compare and have a baseline of the performance and efficiency of the SAT implementation. Our non-SAT implementation was made to be as efficient as we could without the use of a SAT solver, and was not purposely made inefficient for the sake of comparison. The following is a table with the results which we gathered by logging the time it took to run each Python program.

Board Size	Non-SAT Puzzle Solver	SAT Puzzle Solver	Efficiency Improvement	Number of Valid Solutions	Number of unique solutions
5	0.599s	0.234s	60.9%	8	1
6	2m 13s	1.3s	99.0%	200	25
7	> 24 hours	3m 4s	N/A	1952	244
8	> 24 hours	4h 36m 57s	N/A	27856	3482

Figure 3 – Table showing the data and results collected for board sizes 5 – 8.

Both implementations produced the same number of solutions for each corresponding board size. Because of the sheer number of solutions generated from our solvers, we were unable to verify every single produced solution for larger board sizes. However, we did check each produced board for the board size of 5 to verify each was unique and valid, as well as spot-checking random solutions from the larger board sizes. We did this by having our solver print out the board layouts in the terminal prompt. Board sizes of 4 and less produced the expected false statement because there is no possible valid crowded chessboard solution for a board size of less than 5. Additionally to ensure the validity of our solvers' outputs, we tested each helper function designated for generating the boards for each piece and the helper functions that combined the boards. By verifying our solver works for a smaller board size, the validity of our solution should also extend for any larger board sizes.

Comparing the times of the non-SAT crowded chessboard solver with the SAT crowded chessboard solver for all board sizes that we tested, it is quite clear that our SAT implementation is far more efficient than the non-SAT. As demonstrated by the table above, the greater the board size, the larger the gap in efficiency the SAT puzzle solver displays. We were also unable to verify the computing time of our non-SAT puzzle solver for board sizes of 7 and 8. We decided to stop the program after 24 hours of running, as it was affecting the usability of our personal laptop.

Our original measure of success was whether we would be able to create a SAT solver implementation for the crowded chessboard problem. In regards to this, we definitely succeeded. We were able to generate all the valid solutions for a given board size and determine whether there is a valid chessboard layout for any given board size. Our other measure of success was seeing the efficiency comparison between a SAT implementation of the solver with a non-SAT implementation. The difference in efficiency between the different approaches our two algorithms took can be attributed to the progression in efficiency that SAT solvers have seen in the last few years.

Summary:

Our results show the huge advantage of utilizing a SAT solver and using boolean logic when solving problems that scale up. When approaching a problem with a defined set of rules and constraints, while a traditional iterative approach is a valid solution, implementing a boolean SAT solver has proven to be much more efficient and scalable. This is especially true for the problem that we chose for our project, which has a very constrained ruleset that allowed the boolean SAT solver to be implemented very efficiently into.

With the massive leaps in efficiency of SAT solvers from recent years, our project is a perfect example of the power and applicability of SAT solvers in large and complicated problems. Converting a real world problem into a boolean logic problem is an approach that individuals that care about the efficiency of their solution should consider. There are still an infinite number of problems that have not been solved using boolean logic and a SAT implementation, however our project is an example that it may be easier and more effective to implement than one may think. Being able to apply SAT solving to problems is a skill individuals that have limited computing power to solve problems should learn. With our non-SAT implementation, we were not realistically able to come up with the solution for board sizes of 7 and 8, while our SAT implementation was able to run and finish in less than 5 hours on the same computer. Even though we were able to come up with a valid solution without applying boolean logic, without utilizing the SAT solver, we would not be able to generate the valid crowded chessboard solutions with the computing power we currently have at our disposal.

In conclusion, we showed that it is possible and worthwhile to approach developing a solution to a problem using boolean logic. We were not only able to determine whether there is a valid crowded chessboard solution for any given board size and generate all possible solutions, we were also able to show with a direct comparison with a non-SAT solver that a SAT puzzle solver is more efficient.