Seat Arrangement Problem with SAT Solver

b12901193 Chin-Yen, Pan

November 25, 2025

Contents

1 Introduction

This project implements a seat arrangement problem solver using the MiniSat SAT solver. The problem involves assigning students to seats in a classroom while satisfying various constraints including preferred seating zones, desired neighbors, and prohibited neighbors. The solution converts the problem into a Conjunctive Normal Form (CNF) representation and uses MiniSat to find a valid assignment.

The complete source code and documentation for this project are available on GitHub: https://github.com/jasonpan0930/2025fall_EDA_project

2 Project Structure

The project directory contains the following files and their purposes:

2.1 Source Code Files

- `main.cpp` - Main program that orchestrates the entire workflow
- `config.h` - Configuration file defining file names and global variables
- `generate_random_seat.cpp` - Generates random seat configurations for testing
- `generate_student_choice.h` - Generates test student choice data
- `generate_cnf.h` - Generates CNF file from student preferences
- `decode_seat.h` - Decodes MiniSat output into seat assignment table
- `check.h` - Validates the solution against all constraints

2.2 Data Files

- `student_choice.txt` - Input file containing student preferences
- `cnf.cnf` - CNF file generated for MiniSat
- `result.out` - MiniSat solver output
- `seat_assignment.txt` - Final seat arrangement result
- `answer.txt` - Reference answer for testing

2.3 Documentation

- `README_choice_format.md` - Format specification for student choice input
- `seats arrangement problem.pdf` - slide for problem description and methodology

3 File Functionality

(To simplify the report, here I only mention important files with important parts)

3.1 main.cpp

The main program coordinates the entire solving process:

1. Optionally generates student choice file (if user chooses 'y')
2. Calls `generate_cnf()` to create the CNF file
3. Executes MiniSat solver on the CNF file
4. Decodes the solution using `decode_seat()`
5. Validates the solution using `validate_seating()`

3.2 generate_student_choice.h

This module generates test data for the seat arrangement problem. It is used to create realistic student preference files for testing and evaluation purposes.

Preference Generation Algorithm:

For each student i (from 1 to N), the function generates three types of preferences:

1. **Preferred Zone:**
   - Always includes the student's current seat position (r, c)
   - Randomly adds additional seat positions: generates $\frac{1}{3}$ to $\frac{2}{3}$ of the total grid size $(R \times C)$ additional single-seat rectangles
   - Each additional seat is specified as a rectangle with identical start and end coordinates (single cell)
   - Output format: `r1 c1 r2 c2 ; r3 c3 r4 c4 ; ...`
2. **Wants to Sit With:**
   - Identifies all neighbors of student i in the input arrangement (up to 4 adjacent seats: up, down, left, right)
   - For each neighbor, includes them in the "wants" list with probability $\frac{1}{4}$ (25% chance)
   - Output format: space-separated list of student IDs, or `-` if empty
3. **Doesn't Want to Sit With:**
   - Collects all students who are *not* neighbors of student i
   - Randomly shuffles this list
   - Selects 1 to $\lfloor \frac{\text{number of non-neighbors}}{10} \rfloor$ students randomly
   - Output format: space-separated list of student IDs, or `-` if empty

3.3 generate_cnf.h

This module implements the core constraint generation logic:

Variable Encoding:

$$\text{var} = (student\_id - 1) \times R \times C + (row - 1) \times C + col \tag{1}$$

Constraints Generated:

1. **Each seat has exactly one student:** For each seat (r, c) and each pair of students (i, j), adds clause $\neg x_{i,r,c} \vee \neg x_{j,r,c}$
2. **Each student has exactly one seat in preferred zone:**
   - At least one: $(x_{i,r_1,c_1} \vee x_{i,r_2,c_2} \vee \ldots)$ for all preferred seats
   - At most one: $\neg x_{i,r_1,c_1} \vee \neg x_{i,r_2,c_2}$ for each pair of preferred seats
3. **No student in unpreferred zone:** $\neg x_{i,r,c}$ for all unpreferred seats
4. **Wants to sit with:** If student i wants to sit with j at position (r, c), then:
$$\neg x_{i,r,c} \vee x_{j,r-1,c} \vee x_{j,r+1,c} \vee x_{j,r,c-1} \vee x_{j,r,c+1} \tag{2}$$
5. **Doesn't want to sit with:** If student i doesn't want to sit with j at position (r, c), then:
$$\neg x_{i,r,c} \vee \neg x_{j,r-1,c}, \quad \neg x_{i,r,c} \vee \neg x_{j,r+1,c}, \quad \ldots \tag{3}$$

4 Data Flow

The system processes student preferences through a pipeline that converts the problem into CNF format, solves it using MiniSat, and validates the result. Figure 1 illustrates the complete data flow of the system.

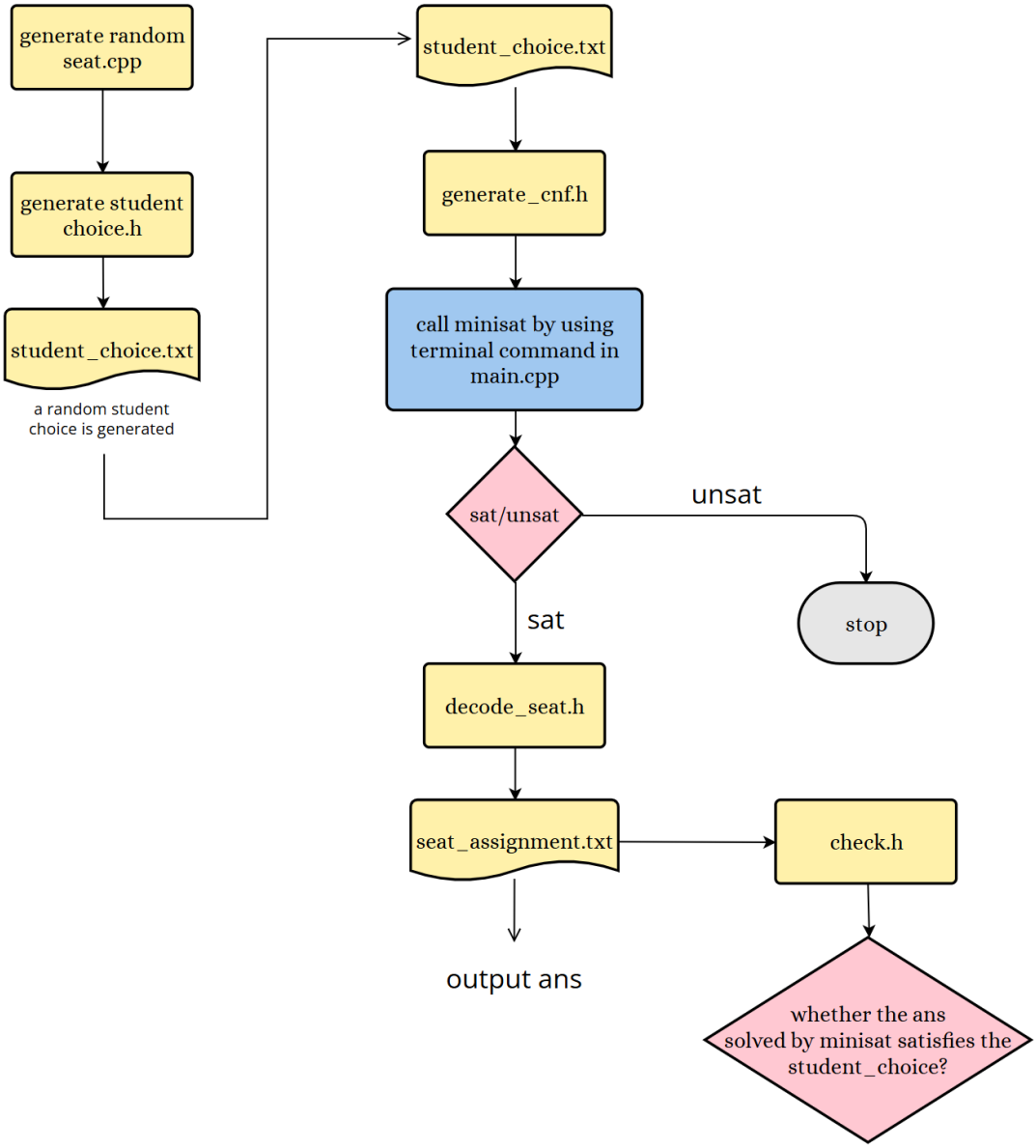

Figure 1: Data flow diagram of the seat arrangement system. The diagram illustrates the complete workflow from student choice generation through CNF conversion, SAT solving, decoding, and validation.

5 Experimental Results

Experiments were conducted with varying numbers of students and classroom dimensions. The following tables summarize the results:

5.1 Performance Metrics

Number of students(N)	25	120	150	200	200	250	300	350	400	500
#Row(R)	5	15	15	20	15	25	20	25	20	25
#Column(C)	5	10	10	10	15	10	15	14	20	20
#Variable	625	18000	22500	40000	45000	62500	90000	122500	160000	250000
#clause	10878	1608631	2402055	5587683	6504631	10850762	18876845	30187591	44643499	86578692
generate_cnf_time(s)	0.1	1.69	2.53	5.662	7.16	10.95	16.1	28.2198	45.663	83.881
cpu time(s)	0.01	1.76	3.3	10.1147	11.9	25.7	55.0189	104.898	202.396	474.892
simplification time(s)	0.01	1.46	3	9.3	10	23.9	51.82	99.66	194.27	457.4
search time(s)	0	0.3	0.3	0.8147	1.9	1.8	3.1989	5.238	8.126	17.492

Figure 2: Performance metrics for various problem sizes. The table shows the number of variables, clauses, and execution times for different configurations of students, rows, and columns.

5.2 Observations

1. **Problem Size Growth:** The number of variables grows as $O(N \times R \times C)$, and clauses grow approximately as $O(R^3 \times C^3)$ for the constraint types implemented. As illustrated in Figure 3, the number of clauses exhibits cubic growth with respect to the number of students, following the polynomial trendline $y = 0.6924x^3 - 5.2803x^2 + 2832.5x - 26524$. This confirms the theoretical complexity analysis showing that clause generation scales as $O(R^3C^3)$.

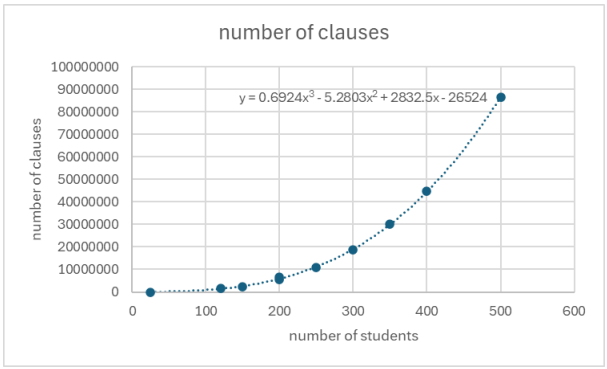

Figure 3: Growth of the number of clauses as a function of the number of students.

2. **CNF Generation Time:** The time to generate CNF files scales roughly linearly with the number of clauses, ranging from 0.1 seconds for small problems (25 students) to 83.88 seconds for large problems (500 students). Figure 4 shows the relationship between CNF generation time and the number of clauses, demonstrating a clear positive correlation. The generation time increases proportionally with clause count, indicating that the CNF generation process is primarily I/O bound rather than computationally complex.

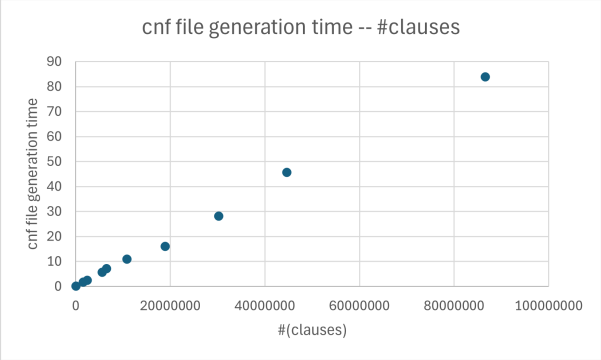

Figure 4: CNF file generation time as a function of the number of clauses.

3. **Solver Performance:** MiniSat's total CPU time is dominated by simplification time, which accounts for 90-96% of total time. The actual search time is relatively small (0-17.5 seconds), indicating that most constraints are resolved during preprocessing. Figure 5 illustrates the CPU time scaling with the number of students, showing a rapid acceleration beyond 250 students. The exponential-like growth pattern reflects the increasing complexity of constraint satisfaction as problem size increases.

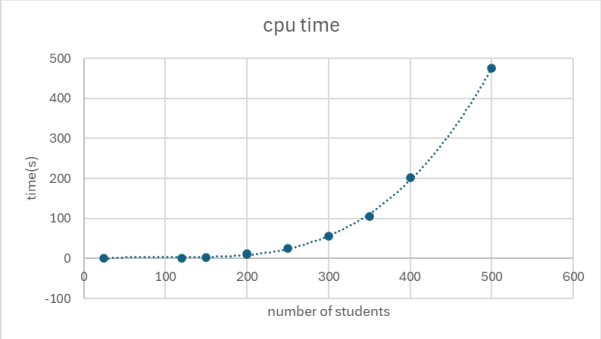

Figure 5: CPU time scaling with the number of students.

4. **Scalability:** The approach successfully handles problems up to 500 students in a 25×20 classroom (250,000 variables, 86.6 million clauses) within reasonable time (under 8 minutes total).

5.3 Example Test Case

A test case with 80 students in a 10×10 classroom was successfully solved:

- The solver found a valid seat arrangement
- All constraints were verified to be satisfied
- Although the result is different from the source of generate_student_choice.h (random_seat), it still satisfies the student_choice.txt

21	-1	44	54	22	10	17	74	38	24
6	59	32	58	69	15	-1	63	64	8
35	-1	53	-1	-1	76	3	47	67	79
-1	56	68	9	-1	42	-1	80	1	14
49	78	72	43	57	29	75	-1	4	50
61	66	-1	62	30	36	52	20	-1	-1
37	27	40	60	71	65	-1	-1	48	73
55	-1	39	-1	-1	33	11	18	28	46
-1	45	51	70	25	16	41	-1	-1	19
13	77	31	23	5	12	7	34	26	2

Figure 6: Random seat arrangement used as input for generating student preferences

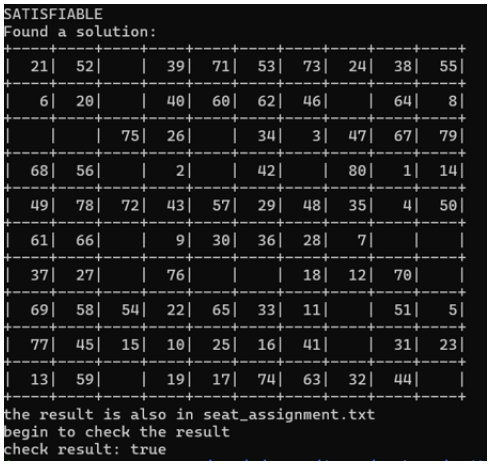

Figure 7: Solution found by MiniSat solver

6 Complexity Analysis

6.1 Variable Count

The number of Boolean variables is:

$$\#\text{Variables} = N \times R \times C = O(NRC) = O(R^2C^2) \text{ (when } N \approx RC) \tag{4}$$

6.2 Clause Count

The clause count is dominated by:

1. **Each seat has one student:** $O(N^2RC) = O(R^3C^3)$
2. **Each student in preferred zone:** $O(N(RC)^2) = O(R^3C^3)$
3. **Neighbor constraints:** $O(N^2RC) = O(R^3C^3)$

Total clause count: $O(R^3C^3)$

7 Conclusion

7.1 Conlusion

This project successfully demonstrates the application of SAT solving to the seat arrangement problem. Key achievements include:

1. **Successful Implementation:** The system correctly models all constraints (preferred zones, desired neighbors, prohibited neighbors) as CNF clauses.
2. **Effective Solving:** MiniSat efficiently solves problems with up to 500 students, finding valid solutions when they exist.
3. **Verification:** The validation module ensures solution correctness by checking all constraints.

4. **Scalability:** The approach handles realistic problem sizes (hundreds of students) within reasonable time constraints.

5. **Modularity:** The codebase is well-structured with clear separation of concerns (CNF generation, solving, decoding, validation).

7.2 Limitations and Future Work

- The current implementation may struggle with very large instances (1000+ students) due to $O(N^3)$ clause growth
- The constraint generation could be optimized to reduce redundant clauses
- Additional constraints (e.g., gender balance, academic performance distribution) could be incorporated
- Alternative SAT solvers or SMT solvers could be evaluated for comparison

The project provides a solid foundation for solving seat arrangement problems using constraint satisfaction techniques, demonstrating the power of SAT solvers for combinatorial optimization problems.