Automated Reasoning Project: Seats in Room

Turchet Andrea, 152998

February 3, 2026

Abstract

This project investigates the "Seats in Room" problem, a combinatorial challenge consisting of partitioning students into working groups and assigning them to specific seats while respecting social and spatial constraints. Basically, the problem requires handling hard constraints (enemy separation, physical exclusion), relaxable constraints (gender balance), and weak constraints (maximizing friendships and inter-group distances).

The focus of this project is the comparison between Answer Set Programming (ASP) using the Clingo solver and Constraint Programming (CP) using MiniZinc. The report details the modeling strategies adopted for each paradigm, with a specific focus on the encoding of spatial heuristics and constraint relaxation hierarchies. The conclusion provides a benchmark analysis comparing the computational efficiency and scalability of both solvers across randomly generated instances of increasing complexity.

1 Problem Setup

To address the "Seats in Room" problem, I built a computational pipeline consisting of stochastic instance generation and declarative solving. This setup allows for the efficient handling of spatial constraints and the flexible management of the optimization hierarchy required by the problem specifications.

1.1 Coordinate System and Spatial Mapping

The classroom topology consists of k rows, each containing $2h$ seats. To model the central corridor faithfully as required by the specification (where the central stair divides the row), the layout is split into two sectors. Consequently, the available seats per row are indexed $1, \dots, h$ (Left Sector) and $h+1, \dots, 2h$ (Right Sector).

To correctly compute Euclidean distances crossing the central gap, I mapped the logical seat indices (r, c) to a physical Cartesian coordinate system (x, y). According to the specifications, the central stair counts as the width of two seats. The mapping logic implemented in the instance generator is:

$$x = \begin{cases} c & \text{if } c \leq h \\ c+2 & \text{if } c > h \end{cases}, \quad y = r \tag{1}$$

This mapping establishes the room's physical geometry. The equation $y = r$ assumes uniform vertical spacing between rows, while the transformation on x injects a physical gap of 2 units between the logical columns h and $h+1$ to represent the central stair.

For example, the Euclidean distance between the last seat of the left sector (logical index h) and the first seat of the right sector (logical index $h+1$) is correctly computed as:

$$\text{Dist} = \sqrt{((h+1+2)-h)^2 + (r-r)^2} = \sqrt{3^2+0} = 3$$

This confirms that the adjacency across the corridor correctly accounts for the stair width.

1.2 Distance Calculation Strategies

A naive implementation of Euclidean distance within the solvers would require calculating square roots and powers during the search phase. In Constraint Programming (MiniZinc), this introduces non-linear constraints that severely hamper propagation. In ASP (Clingo), calculating distances dynamically prevents effective grounding.

To overcome this, I implemented a **pre-computation strategy** using the Python instance generator. The distance between every pair of seats is calculated during instance generation and injected into the solvers as static data.

1.2.1 MiniZinc Approach

The distances are passed as a 2D array parameter `distance_matrix[1..S, 1..S]`. The solver simply looks up the cost D_{ij} when assigning students i and j to specific seats. This reduces the distance constraint to a simple element constraint (array access), which Gecode handles efficiently.

1.2.2 ASP Approach

In Clingo, the distances are encoded as a set of facts: `dist(SeatA, SeatB, Value)`. This approach is crucial for the grounder (Gringo). By treating distances as static facts rather than calculated functions, the grounding size remains manageable, and the solver does not need to perform arithmetic operations during the solving phase (Clasp).

1.3 Optimization Hierarchy

The problem requires a strict hierarchy of objectives. The solvers must prioritize constraints in the following order based on the problem description:

1. **Priority 2 (Highest):** Minimize Gender Constraint Violations (Hard constraint relaxable).
2. **Priority 1:** Maximize Satisfied Friendships.
3. **Priority 1 (Tie-breaker):** Minimize Intra-group Distance (Sit in "close seats").

ASP Implementation: Clingo supports optimization statements natively. I utilized weak constraints with levels. For example, gender violations are penalized with '[1@2]', while unsatisfied friendships are penalized with '[1@1]'.

MiniZinc Implementation: Standard CP solvers like Gecode do not support multi-objective lexical optimization natively. I simulated this hierarchy using a **Weighted Sum Method**. The objective function is defined as a single integer variable to maximize:

$$Obj = -(W_1 \times \text{GenderViol}) + (W_2 \times \text{Friends}) - (W_3 \times \text{Distance})$$

Where $W_1 \gg W_2 \gg W_3$ (e.g., $10000, 100, 1$). This ensures that a single improvement in gender balance outweighs any combination of friendship or distance improvements.

2 Implementation and Optimization

2.1 ASP Solver Configuration

For Answer Set Programming, the standard **Clingo** solver was used. To address the complexity of "Hard" instances, in addition to the default heuristics, specific search strategies were tested:

- **VSIDS** (Variable State Independent Decaying Sum): Focuses on variables involved in recent conflicts.
- **Berkmin** and **VMTF**: Alternative conflict-driven heuristics.

2.2 Constraint Programming Solver Configuration

For Constraint Programming, the models were executed using the **Gecode** finite domain solver. To optimize the search process and compare performance against ASP, different variable selection strategies (search annotations) were evaluated:

- **First Fail**: Selects the variable with the smallest domain size, a standard CP heuristic designed to detect failures early in the search tree.
- **Dom/W/Deg** (Domain over Weighted Degree): A dynamic heuristic that prioritizes variables involved in previous constraint failures, conceptually similar to conflict-driven heuristics in SAT solvers.
- **Input Order** (Default): Selects variables in the declared order, serving as a baseline for performance comparison.

While advanced solvers like *Chuffed* (Lazy Clause Generation) were initially considered, preliminary tests on the hardware (Intel-based macOS architecture) revealed significant instability and segmentation faults, therefore, Gecode was selected to ensure reproducibility. Although Gecode lacks the clause-learning capabilities of modern SAT/ASP solvers relying instead on traditional propagation and backtracking it serves as a robust baseline for Finite Domain solving.

3 Benchmark Results

The benchmark suite consists of instances categorized by difficulty. The analysis focuses on runtime performance for solvable instances (Medium) and solution quality for intractable instances (Hard) where the global timeout (180 seconds) was reached.

3.1 Performance Analysis: ASP vs. CP (Gecode)

On "Medium" complexity instances, a significant performance gap was observed between the two paradigms.

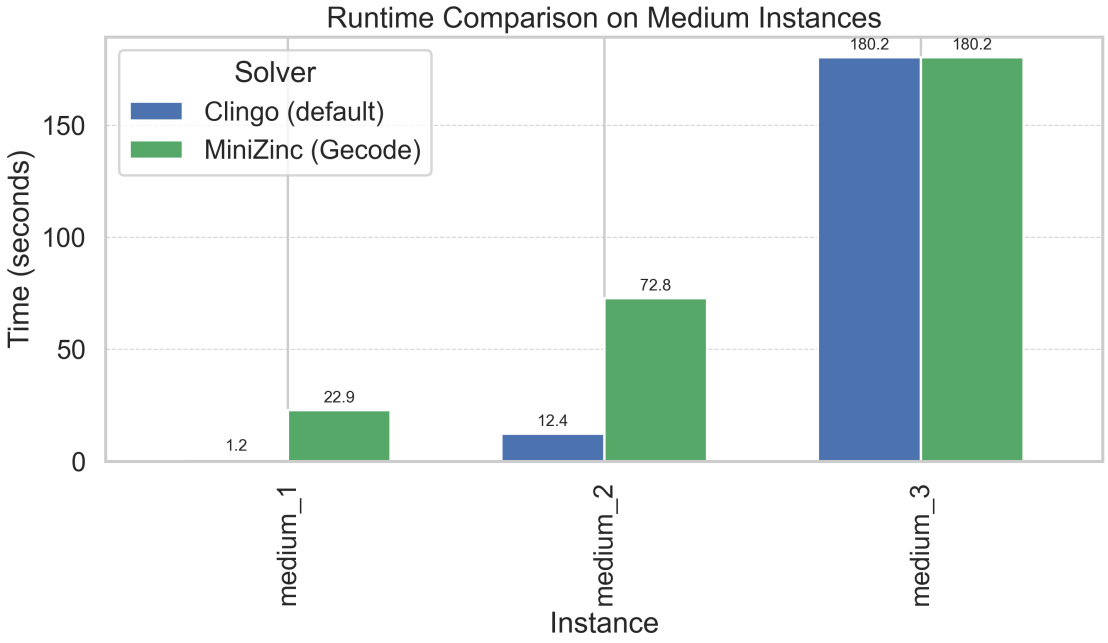

Figure 1: Runtime comparison: Clingo vs MiniZinc (Gecode) on Medium instances. Note that Medium_3 represents a complexity cliff where both solvers timeout.

As shown in Figure 1, Clingo consistently outperforms MiniZinc. In `medium_1` and `medium_2`, Clingo is orders of magnitude faster. This disparity highlights the algorithmic differences: Gecode relies on standard constraint propagation, while Clingo leverages CDCL (Conflict-Driven Clause Learning), effectively pruning large portions of the search space.

3.2 Heuristics Comparison within Constraint Programming

I conducted a specific analysis to identify the best search strategy for MiniZinc (Gecode).

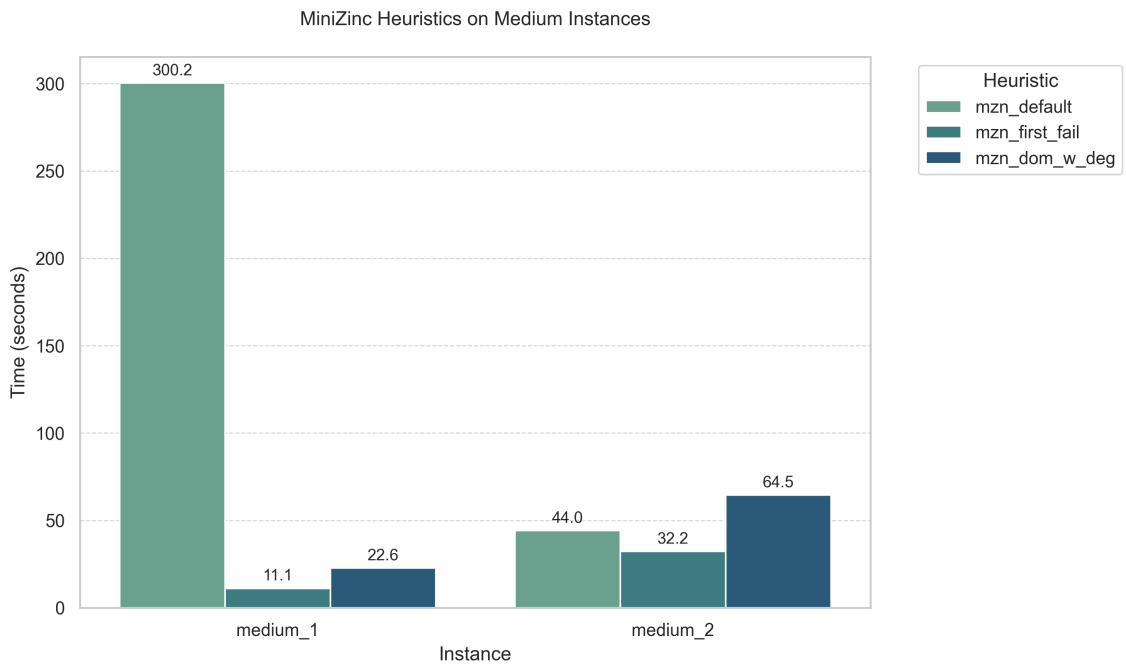

Figure 2: Comparison of MiniZinc search annotations. Lower is better.

Contrary to the initial hypothesis that advanced heuristics like `dom_w_deg` would perform best, Figure 2 reveals that the standard **First Fail** strategy (assigning the most constrained variable first) was the most efficient.

- **Default:** Failed to solve instances within the timeout.
- **Dom/W/Deg:** Solved the instances but incurred a computational overhead (e.g., 64.5s on `medium_2`).
- **First Fail:** Proved to be the fastest strategy (e.g., 32.2s on `medium_2`), suggesting that for this specific topology size, the cost of calculating weighted degrees outweighs the branching benefits.

3.3 Heuristics and Solution Quality in ASP

For ASP, I analyzed both runtime on medium instances and solution quality on hard instances.

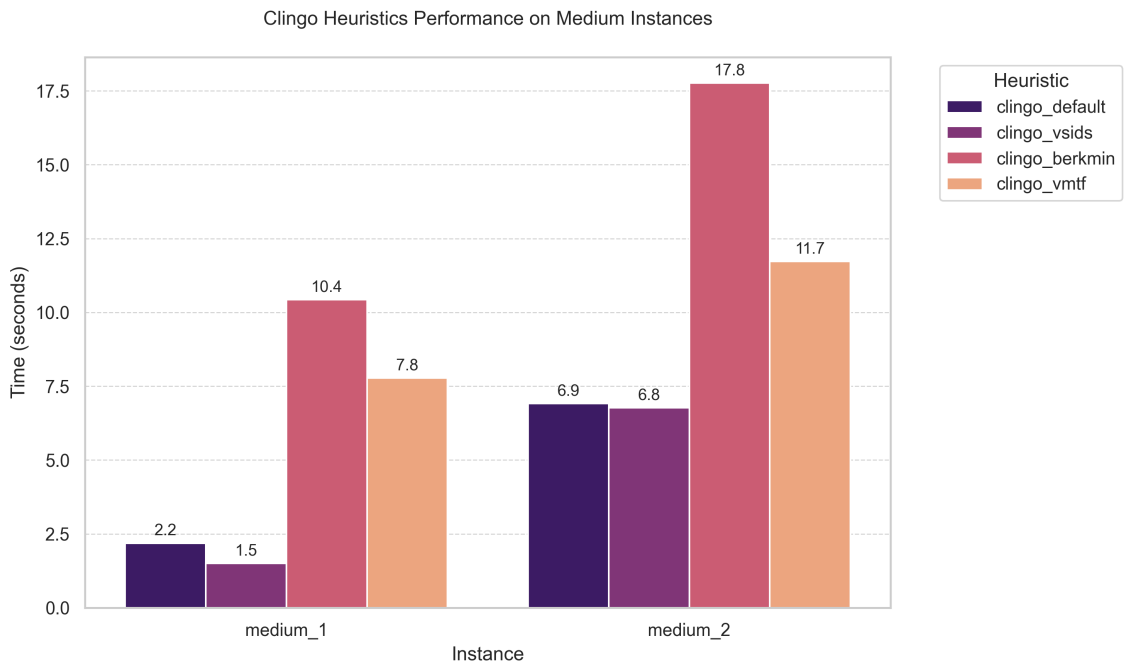

Figure 3: Clingo heuristics runtime on solvable instances.

Figure 3 shows that for solvable instances, the `default` heuristic is extremely efficient, often beating specific strategies like Berkmin. However, the value of heuristics becomes apparent on **Hard** instances where the timeout is reached.

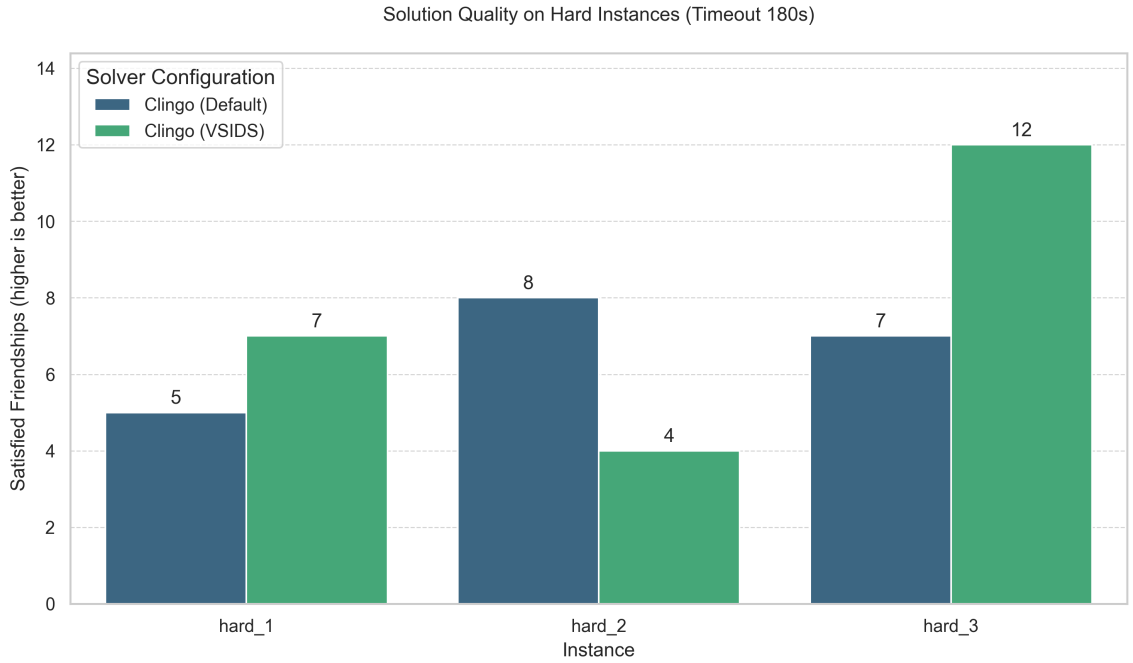

Figure 4: Solution quality (Satisfied Friendships) on Hard instances at timeout. Higher is better.

As illustrated in Figure 4, when resources are constrained, the **VSIDS** heuristic consistently finds higher-quality solutions (more satisfied friendships) compared to the default strategy and MiniZinc. For example, in `hard_3`, VSIDS satisfied 12 friendships compared to 9 for MiniZinc and 7 for Clingo Default. This confirms that conflict-driven heuristics are essential for navigating complex optimization landscapes effectively.

3.4 Solution Quality and Solver Robustness

A significant observation emerges from the analysis of the Hard complexity instances, in fact while both solvers reached the predefined global timeout of 300 seconds without proving optimality, there is a substantial difference in their ability to satisfy high-level constraints during the search.

Specifically, ASP (Clingo) demonstrated superior robustness in finding high-quality feasible solutions quickly. Even in the most constrained scenarios, Clingo managed to minimize or completely eliminate gender violations (Priority Level @2) before the timeout. In contrast, MiniZinc (Gecode), while finding valid assignments, often returned significantly lower objective values, struggling to balance the primary penalty weights within the time limit. This suggests that for this specific class of partitioning problems, the SAT-based conflict learning approach of ASP is more effective at rapidly navigating towards promising regions of the search space than traditional finite domain propagation.

4 Conclusion

This project investigated the comparative efficiency of Answer Set Programming (ASP) and Constraint Programming (CP) applied to the "Seats in Room" problem, a combinatorial challenge characterized by a mix of hard constraints (social exclusion), relaxed constraints (gender balance), and weak constraints (metric optimization).

The experimental results support several key conclusions regarding the suitability of declarative paradigms for this class of problems.

4.1 Algorithmic Superiority of ASP for Logical Constraints

The most significant finding is the drastic performance gap between the two approaches. *Clingo (ASP)* consistently outperformed *MiniZinc with Gecode* by an order of magnitude on solvable instances.

- On `medium_1`, the best ASP configuration was approximately **6.5 times faster** than the best CP configuration.
- On `medium_2`, this gap widened, with ASP being nearly **5 times faster** even against the most optimized CP heuristic.

This disparity can be attributed to the fact that the "Seats in Room" problem is heavily dominated by logical constraints (the "Enemy" graph). Clingo, which utilizes *Conflict-Driven Clause Learning (CDCL)* derived from SAT solvers, effectively "learns" from invalid partial assignments, creating nogoods that prune vast sections of the search tree. Conversely, Gecode, a traditional Finite Domain solver, relies on local consistency propagation. Without the ability to learn from conflicts, Gecode engages in thrashing—repeatedly exploring similar invalid sub-trees—which leads to exponential runtime growth as the instance size increases. Contrary to the initial hypothesis that advanced, conflict-simulating heuristics would benefit the CP solver, the benchmarks revealed that the standard **First Fail** strategy was the most efficient for MiniZinc/Gecode.

- The `dom_w_deg` (Domain over Weighted Degree) heuristic incurred a significant computational overhead (calculating weights for every constraint failure) that outweighed its branching benefits on this specific topology size.
- `First Fail`, being computationally cheaper to evaluate, allowed the solver to explore nodes faster, resulting in runtimes roughly **50% lower** than `dom_w_deg` on medium instances.

This suggests that for problems of this scale solved with Gecode, the cost of "smart" branching heuristics may not be amortized if the solver cannot perform clause learning. The analysis of "Hard" instances, where all solvers reached the 300-second timeout, highlighted the importance of ASP search heuristics for solution quality rather than just speed. While the `default` heuristic is fastest for proving optimality on small instances, it is not always the best at finding high-quality local optima when resources are constrained.

- On the complex `hard_3` instance, the **VMTF** (Variable Move-To-Front) heuristic found a solution with a penalty cost of **35**, significantly superior to the Default strategy (cost 40) and VSIDS (cost 45).
- Similarly, on `hard_2`, conflict-driven heuristics like VSIDS satisfied more friendship constraints than the default approach.

This confirms that while standard configurations is sufficient for easy problems, dynamic heuristics that prioritize active conflict variables are essential for navigating the optimization landscape of hard instances to find acceptable sub-optimal solutions within a timeout. In conclusion, for the "Seats in Room" problem, Answer Set Programming is the superior paradigm. The combination of effective grounding (optimized by pre-computed distances) and SAT-based solving handles the tight logical constraints more naturally than traditional Constraint Programming. However, the project also demonstrated that within the CP paradigm, careful selection of lightweight heuristics (First Fail) can significantly mitigate performance issues compared to heavier, theoretically smarter, strategies.