Sorting as a Planning Problem

Master's Degree in Computer Science

Constraint Programming Course — 2025/2026

Crafa Raffaele and **Merenda Saverio Mattia**

January 25, 2026

Contents

1 Introduction

This project addresses **Project 5: Sorting as a Planning Problem**. Given a vector $\vec{v} = [v_1, \ldots, v_n]$ representing a permutation of integers $\{1, \ldots, n\}$, the objective is to find the **minimum number of pairwise swaps** required to sort it in ascending order.

A `swap(i, j)` operation exchanges the values at positions i and j: after the swap, $v[i]$ and $v[j]$ are interchanged. The problem asks for the **minimum length plan**—the shortest sequence of swaps—that transforms the initial permutation into the sorted sequence $[1, 2, \ldots, n]$.

1.1 Problem Formalization

The problem can be formally stated as follows:

- **Input:** A permutation $\vec{v} = [v_1, \ldots, v_n]$ of integers from 1 to n.
- **Output:** A sequence of swaps $\langle \text{swap}(i_1, j_1), \text{swap}(i_2, j_2), \ldots, \text{swap}(i_k, j_k) \rangle$ of minimum length k such that applying these swaps in order produces the sorted sequence $[1, 2, \ldots, n]$.

Example Consider the permutation $\vec{v} = [2, 3, 1, 5, 4]$ of size $n = 5$:

- **Initial state:** $[2, 3, 1, 5, 4]$
- **Step 1:** $\text{swap}(1, 2) \rightarrow [3, 2, 1, 5, 4]$ (swap positions 1 and 2)
- **Step 2:** $\text{swap}(1, 3) \rightarrow [1, 2, 3, 5, 4]$ (swap positions 1 and 3)
- **Step 3:** $\text{swap}(4, 5) \rightarrow [1, 2, 3, 4, 5]$ (swap positions 4 and 5)

The minimum plan length is $k = 3$. No sequence of fewer than 3 swaps can sort this permutation.

1.2 Sorting as a Planning Problem

This problem naturally maps to a **discrete planning problem**:

- **State:** The current configuration of the vector at each timestep t.
- **Action:** A swap operation $\text{swap}(i, j)$ that modifies the state.
- **Initial State:** The input permutation $\vec{v}$.
- **Goal State:** The sorted sequence $[1, 2, \ldots, n]$.
- **Horizon:** The number of timesteps k (plan length).

The problem is modeled as a **Constraint Satisfaction Problem (CSP)** where, for a fixed horizon k, the solver must determine whether a valid plan exists. The minimum k is found via **Iterative Deepening**: we start with $k = k_{\min}$ (a theoretical lower bound) and increment k until a satisfiable solution is found.

1.3 Theoretical Lower Bound: Cycle Decomposition

From **Group Theory**, every permutation can be decomposed into disjoint cycles. The minimum number of swaps required to sort a permutation is:

$$k_{\min} = n - c \tag{1}$$

where c is the number of disjoint cycles in the permutation.

Cycle Decomposition. Any permutation can be decomposed into disjoint **cycles**. A cycle describes a closed chain of positions where elements need to "rotate" to reach their correct places. To find cycles, we start at position i, follow the chain $i \to v[i] \to v[v[i]] \to \cdots$ until returning to i, then repeat for unvisited positions.

Example. For $\vec{v} = [2, 3, 1, 5, 4]$:

- **Cycle 1:** $1 \to 2 \to 3 \to 1$ (positions 1, 2, 3 form a cycle of length 3)
- **Cycle 2:** $4 \to 5 \to 4$ (positions 4, 5 form a cycle of length 2)

Total: $c = 2$ cycles.

Minimum Swaps. A cycle of length ℓ requires exactly $\ell - 1$ swaps to sort. Therefore:

$$k_{\min} = \sum_{i=1}^{c} (\ell_i - 1) = n - c$$

For our example: $(3 - 1) + (2 - 1) = 3$ swaps, matching $k_{\min} = 5 - 2 = 3$.

1.4 Project Structure

The project implementation is distributed across five files:

1. `sorting.mzn`: The core MiniZinc model defining variables, constraints, and the search strategy for a fixed horizon k.
2. `benchmark.py`: A Python meta-solver implementing Iterative Deepening with cycle-based lower bound computation.
3. `sorting_template.mzn`: An extended version of the base model with a placeholder for dynamic strategy injection.
4. `benchmark_strategies.py`: The experimental suite for comparing 12 different search heuristics.
5. `plot.py`: A visualization script for graphical analysis of benchmark results.

2 Problem Modeling: `sorting.mzn`

The core logical model is implemented in `sorting.mzn`. The problem is framed as a discrete-time state-transition system where a vector $\vec{v}$ of size n must be transformed into a sorted configuration within exactly k steps. To ensure the solver can handle high-dimensional instances ($N = 30$), the model integrates advanced propagation techniques and mathematical properties of permutations.

2.1 MiniZinc Overview

MiniZinc is a high-level constraint modeling language that separates the problem specification from the solver implementation. Key features used in this project include:

- **Global Constraints:** Predefined constraints like `alldifferent` and `inverse` that have specialized propagation algorithms.
- **Annotations:** Hints to the solver about propagation levels (e.g., `::  domain` for Generalized Arc Consistency) and search strategies.
- **Solver Independence:** The same model can be solved by different backends (Gecode, Chuffed, OR-Tools).

2.2 Parameters and Decision Variables

A rigorous definition of the search space is provided through the following parameters and variables:

```
int: n;
array[1..n] of int: start_v;
int: k;
```

The parameters `n` and `start_v` define the instance size and the initial unsorted state. The parameter `k` represents the fixed planning horizon. In our architecture, `k` is controlled by the Python meta-solver to find the minimum plan length.

```
array[0..k, 1..n] of var 1..n: v;
array[0..k, 1..n] of var 1..n: pos;
```

We declare two 2D matrices to represent the state of the system at each time step $t \in [0, k]$.

- `v[t,p]`: The value contained in position p at time t.
- `pos[t,i]`: The position of value i at time t.

Motivation: This is a **Dual Modeling** approach (also known as **Channeling**). By representing the same state in two ways, the solver can propagate information bidirectionally. For example, assigning a value to a position in `v` immediately restricts the domain of the corresponding entry in `pos`. In this way we are combining two points of view.

```
array[0..k-1] of var 1..n: idx1;
array[0..k-1] of var 1..n: idx2;
```

These are the action variables. For each transition from t to t+1, the solver must decide the pair of indices ($idx1_t$,$idx2_t$) to be swapped.

2.3 Initial, Goal, and Parity Constraints

The boundary conditions and mathematical invariants of the problem are enforced here:

```
constraint forall(p in 1..n) (v[0, p] = start_v[p]);
constraint forall(p in 1..n) (v[k, p] = p);
```

The first constraint initializes the state at t=0 using the input data. The second constraint defines the **Goal State**: at time t=k, the vector must be the identity permutation $[1, 2, \ldots, n]$. This is stronger than simply requiring ascending order, as it provides better backward propagation—the solver knows exactly what value each position must contain at the final timestep.

```
int: initial_inv = sum(i, j in 1..n where i < j)(
    bool2int(start_v[i] > start_v[j])
);
constraint (k mod 2) == (initial_inv mod 2);
```

This constraint calculates the number of inversions in the initial permutation. That's because, in Group Theory, every swap (transposition) changes the parity of a permutation. Therefore, if the initial state requires an odd number of swaps to be reached, any even k is mathematically impossible (and viceversa, if we have an even number of swaps to do, any odd k is mathematically impossible). By enforcing this **Parity Invariant**, the solver can return `UNSAT` for half of the iterative deepening steps during the flattening phase, significantly reducing the total benchmarking time.

2.4 Channeling and Global Propagation

To maximize the inferential power of the solver, we link the dual representations:

```
constraint forall(t in 0..k) (
    inverse([v[t, p] | p in 1..n], [pos[t, i] | i in 1..n]) /\
    alldifferent([v[t, p] | p in 1..n]) :: domain
);
```

The `inverse` global constraint ensures that $v[t, p] = i \iff pos[t, i] = p$. We are enforcing **Generalized Arc Consistency (GAC)** via `:: domain` on the `alldifferent` constraint, that is much more powerful than simple binary decomposition. Combined with channeling, it allows Gecode to detect conflicts by looking at the "available slots" for values, pruning entire branches of the search tree before any value is assigned. To better understand GAC we can see the following example:

Let's consider a subset of the vector at time t with three variables v_1,v_2,v_3 and the following domains:

- $D(v_1) = \{1, 2\}$
- $D(v_2) = \{1, 2\}$
- $D(v_3) = \{1, 2, 3\}$

Applying the global constraint `alldifferent(`v_1`,`v_2`,`v_3`)` leads to different results depending on the consistency level:

1. **Binary Arc Consistency (AC):** If the solver decomposes the global constraint into binary inequalities ($v_1 = v_2$, $v_1 = v_3$, $v_2 = v_3$), it will fail to prune the domains.

For instance, for $v_3 = 1$, there exists a valid assignment for v_1 ($v_1 = 2$) and v_2 (though v_2 would be restricted, binary AC does not "see" the simultaneous pressure on values 1 and 2). Thus, no values are removed from D(v_3).

2. **Generalized Arc Consistency (GAC):** GAC analyzes all variables simultaneously. It identifies that the set $\{v_1, v_2\}$ constitutes a "Hall set" of size 2, requiring two values $\{1, 2\}$. Consequently, these values are mathematically unavailable for any other variable in the constraint. GAC instantly prunes 1 and 2 from $D(v_3)$, resulting in $D(v_3) = \{3\}$.

In our sorting model, where N can be as large as 30, the ability to prune values before starting the search tree (branching) is vital. By using `alldifferent :: domain`, we enforce the solver to use Régin's algorithm, which identifies these bottlenecks immediately. Combined with **Channeling**, this ensures that the solver never explores permutations that are logically impossible, transforming an exponential search into a highly efficient deductive process.

2.5 Transition Logic and Optimal Search Pruning

This section defines the physics of the environment and applies aggressive pruning based on the properties of minimal plans.

```
1  constraint forall(t in 0..k-1) (
2      idx1[t] < idx2[t] /\
3
4      (v[t, idx2[t]] = idx1[t] \/ v[t, idx1[t]] = idx2[t]) /\
5
6      v[t+1, idx1[t]] = v[t, idx2[t]] /\
7      v[t+1, idx2[t]] = v[t, idx1[t]] /\
8
9      forall(p in 1..n where p != idx1[t] /\ p != idx2[t]) (
10         v[t+1, p] = v[t, p]
11     )
12 );
```

What are we doing with the code section above:

1. **Canonical Representation:** `idx1[t] < idx2[t]` prevents the solver from considering swap(2,1) as different from swap(1,2), halving the branching factor.

2. **Optimal Swap Property:** The constraint at line 4 is the most significant optimization. In a minimum-length plan for the sorting problem, it is never optimal to make a swap that doesn't put at least one element into its correct final position. This **Local Optimality** constraint reduces the possible choices at each step from $O(N^2)$ to approximately $O(N)$, allowing the model to scale to N=30.

3. **State Transition:** The third and fourth constraints (lines 6 and 7) define the exchange of values between t and $t + 1$.

4. **Frame Axiom:** The nested `forall` ensures that all values not involved in the swap remain static. Without this, the solver could "teleport" values between steps.

2.6 Symmetry Breaking for Independent Moves

```
constraint forall(t in 0..k-2) (
    (idx1[t] != idx1[t+1] \/ idx2[t] != idx2[t+1]) /\
    let {
        var bool: disjoint = (idx1[t] != idx1[t+1] /\
                              idx1[t] != idx2[t+1] /\
                              idx2[t] != idx1[t+1] /\
                              idx2[t] != idx2[t+1])
    } in (
        disjoint -> (idx1[t] < idx1[t+1])
    )
);
```

This constraint enforces two symmetry-breaking rules:

1. Preventing the solver from performing a swap and then immediately reversing it (e.g., swap(1,2)→swap(1,2)).

2. If two consecutive swaps are disjoint (acting on four different indices), the order in which they are performed is irrelevant, therefore we force a lexicographical order on `idx1` for disjoint moves to avoid exploring identical plans. Consider, for exmple, the vector $V_t = [2, 1, 4, 3]$. To sort it, two disjoint swaps are required: S_A at indices $(1, 2)$ and S_B at indices $(3, 4)$. The solver could explore two equivalent sequences:

   (a) $[2, 1, 4, 3] \xrightarrow{S_A} [1, 2, 4, 3] \xrightarrow{S_B} [1, 2, 3, 4]$ (Sequence: $idx1[0] = 1, idx1[1] = 3$)

   (b) $[2, 1, 4, 3] \xrightarrow{S_B} [2, 1, 3, 4] \xrightarrow{S_A} [1, 2, 3, 4]$ (Sequence: $idx1[0] = 3, idx1[1] = 1$)

   The constraint `disjoint -> (idx1[t] < idx1[t+1])` ensures that only the first sequence is explored, as $1 < 3$ is true while $3 < 1$ is false. This effectively removes redundant search branches that lead to the same state.

2.7 Fixed Point Propagation

This constraint prevents the solver from involving already-fixed elements in future swaps:

```
constraint forall(t in 0..k-1, p in 1..n) (
    v[t, p] = p -> (idx1[t] != p /\ idx2[t] != p)
);
```

If an element is already in its correct position ($v[t, p] = p$), involving it in any swap would displace it unnecessarily. In an optimal plan, no move should "undo" progress already made.

Example Consider the state $[1, 3, 2, 4, 5]$ at time t. Elements 1, 4, and 5 are already fixed. The constraint ensures that positions 1, 4, and 5 cannot appear in $idx1[t]$ or $idx2[t]$, reducing the branching factor from $\binom{5}{2} = 10$ possible swaps to $\binom{2}{2} = 1$ (only swap(2,3) is valid).

2.8 Global Uniqueness of Swaps

This constraint ensures that each swap pair (i, j) appears at most once in the entire plan:

```
constraint forall(t1 in 0..k-2, t2 in t1+1..k-1) (
    idx1[t1] != idx1[t2] \/ idx2[t1] != idx2[t2]
);
```

In an optimal plan, repeating the same swap is never useful:

- **Consecutive repetition:** Two identical swaps cancel each other (already prevented by the first rule of the Symmetry Breaking constraint in Section 2.6).
- **Non-consecutive repetition:** If swap(i, j) appears at times t_1 and t_2 $(t_1 < t_2)$, the elements at positions i and j return to their original values after t_2. This implies the intermediate swaps could be reorganized into a shorter plan.

This constraint has $O(k^2)$ comparisons, but since $k \leq n - 1$ for optimal plans, the overhead is negligible.

2.9 Summary of Constraints

The complete model includes 7 main constraints:

#	**Constraint**	**Effect**
1	Boundary Conditions	Fixes initial and goal states
2	Parity Invariant	Prunes 50% of k values
3	Channeling + GAC	Maximum propagation via dual model
4	Optimal Swap Property	Reduces branching from $O(N^2)$ to $O(N)$
5	Symmetry Breaking	Eliminates equivalent plans
6	Fixed Point Propagation	Avoids moves on already-sorted elements
7	Global Swap Uniqueness	Prevents redundant swap repetitions

Table 1: Summary of constraints in `sorting.mzn`

3 The Benchmark Engine: `benchmark.py`

MiniZinc solves problems for a **fixed** horizon k. To find the *minimum* k, we need an external controller. The script `benchmark.py` acts as a **meta-solver** implementing Iterative Deepening with intelligent lower bound computation.

3.1 Benchmark Suite Generation

Following the project specification, we generate 60 benchmark instances:

```
def generate_benchmarks():
    benchmarks = []
    sizes = [5, 10, 15, 20, 25, 30]  # As per specification
    for n in sizes:
        for _ in range(10):  # 10 permutations per size
            vec = list(range(1, n + 1))
            random.shuffle(vec)
            benchmarks.append((n, vec))
    return benchmarks
```

This produces:

- 10 instances for $N = 5$ (instances 1–10)
- 10 instances for $N = 10$ (instances 11–20)
- 10 instances for $N = 15$ (instances 21–30)
- 10 instances for $N = 20$ (instances 31–40)
- 10 instances for $N = 25$ (instances 41–50)
- 10 instances for $N = 30$ (instances 51–60)

3.2 Cycle Decomposition for Lower Bound

Instead of starting with $k = 1$ and incrementing blindly, we compute the theoretical minimum using cycle decomposition:

```
def count_cycles(perm):
    n = len(perm)
    visited = [False] * n
    num_cycles = 0
    for i in range(n):
        if not visited[i]:
            num_cycles += 1
            j = i
            while not visited[j]:
                visited[j] = True
                j = perm[j] - 1  # Follow the cycle
    return num_cycles

def compute_starting_k(perm):
    n = len(perm)
    num_cycles = count_cycles(perm)
    k_min = n - num_cycles  # Theoretical lower bound

```

```
    # Parity correction
    initial_inv = count_inversions(perm)
    if (k_min % 2) != (initial_inv % 2):
        k_min += 1
    return k_min
```

The cycle decomposition gives us the exact minimum for most permutations. The parity check ensures we don't start with an impossible k value (which would waste one iteration).

3.3 Iterative Deepening with Intelligent Start

```
def solve_sorting_instance(model, solver, n, start_v):
    # Start from theoretical lower bound, not k=1
    k = compute_starting_k(start_v)

    while not found:
        instance = minizinc.Instance(solver, model)
        instance["n"] = n
        instance["start_v"] = start_v
        instance["k"] = k

        result = instance.solve(timeout=timedelta(seconds=300))

        if result.status == minizinc.Status.SATISFIED:
            return k, result  # Found optimal!
        elif result.status == minizinc.Status.UNSATISFIABLE:
            k += 1  # Try longer plan
```

Optimization impact: For $N = 30$, a random permutation typically has $k_{\min} \approx 25$. Without cycle decomposition, the solver would waste time on 24 UNSAT iterations. With it, we often find the solution on the first attempt.

3.4 Output Format

Results are saved to individual text files in `result_benchmark/`:

```
def save_result_to_file(index, n, vec, k, time_taken, result):
    filename = f"result_{index:02d}_N{n}.txt"
    # Writes: dimension, input vector, K found, time, plan steps
```

Each file contains the complete solution trace, including every swap performed and the resulting intermediate states.

4 Search Strategies and Template Model

To systematically compare different search strategies, we created `sorting_template.mzn`, which is **identical** to `sorting.mzn` with two differences: (1) the `solve` statement is replaced by a placeholder {{`SOLVE_STRATEGY`}} for dynamic strategy injection, and (2) an auxiliary array `all_moves` is defined to aggregate the decision variables.

The array `all_moves` interleaves `idx1` and `idx2` as follows:

```
array[1..2*k] of var 1..n: all_moves = [
    if i mod 2 == 1
    then idx1[i div 2]
    else idx2[(i-1) div 2]
    endif
    | i in 1..2*k
];
```

This produces the sequence $[idx1_0, idx2_0, idx1_1, idx2_1, \ldots, idx1_{k-1}, idx2_{k-1}]$. By alternating the swap indices, the solver decides a complete move $(idx1_t, idx2_t)$ before advancing to the next timestep, which is more natural for this problem than the simple concatenation `idx1 ++ idx2`.

The Python script `benchmark_strategies.py` reads the template, replaces the placeholder with the desired search strategy, and writes a temporary model file for execution. This meta-programming approach allows testing multiple strategies without modifying the core model.

Each strategy is a combination of three components: (1) a **variable selection heuristic** that determines which variable to branch on next, (2) a **value selection heuristic** that determines which value to try first, and (3) a **restart policy** that determines when to abandon the current search path.

4.1 Variable Selection Heuristics

4.1.1 Default (Gecode's Internal Heuristic)

When no explicit heuristic is specified, Gecode uses its internal variable ordering, typically equivalent to `input_order` with `indomain_min`. This serves as our baseline for comparison.

```
solve :: restart_luby(250) satisfy;
```

4.1.2 First-Fail

The `first_fail` heuristic selects the variable with the smallest domain. The intuition is that variables with small domains are more constrained and likely to cause failures—by branching on them first, the solver detects inconsistencies early and avoids exploring large barren subtrees.

```
int_search(all_moves, first_fail, indomain_random, complete)
```

Example. Consider two move variables at time $t = 0$ and $t = 1$. After constraint propagation, suppose $D(idx1_0) = \{1, 2, 3, 4, 5\}$ and $D(idx1_1) = \{2, 3\}$. The heuristic selects $idx1_1$ first because its domain has only 2 values. If this choice leads to a conflict, the solver discovers it immediately rather than after exploring all 5 possibilities for $idx1_0$.

4.1.3 Domain/Weighted Degree

The `dom_w_deg` heuristic selects the variable that minimizes $\frac{|D(x)|}{w(x)}$, where $w(x)$ is the *weighted degree*—a counter that tracks how many times constraints involving x have caused failures during search. Unlike static heuristics, `dom_w_deg` **learns from failures**, adapting its variable ordering to the specific instance.

```
int_search(all_moves, dom_w_deg, indomain_random, complete)
```

Example. Initially, all weights are equal and the heuristic behaves like `first_fail`. Suppose the constraint `v[t, idx2[t]] = idx1[t]` frequently causes failures when $idx2_0 = 3$. The weight of $idx2_0$ increases, making it more likely to be selected early. Over time, the solver "learns" that certain variables are problematic and prioritizes them.

4.1.4 Smallest

The `smallest` heuristic selects the variable whose minimum domain value is smallest. Combined with `indomain_min`, this creates a deterministic, reproducible search that prefers low indices.

```
int_search(all_moves, smallest, indomain_min, complete)
```

Example. If $D(idx1_0) = \{3, 4, 5\}$ and $D(idx1_1) = \{1, 2\}$, the heuristic selects $idx1_1$ because its minimum value (1) is smaller than the minimum of $idx1_0$ (3).

4.1.5 Most Constrained

The `most_constrained` heuristic works like `first_fail` but breaks ties by selecting the variable involved in the most constraints. This prioritizes variables that are both tightly bounded and highly connected in the constraint graph.

```
int_search(all_moves, most_constrained, indomain_random, complete)
```

Example. Suppose $idx1_0$ and $idx2_0$ both have domain size 2. However, $idx1_0$ appears in 5 constraints while $idx2_0$ appears in 3. The heuristic selects $idx1_0$ because it is more "central" to the constraint network.

4.1.6 Max Regret

The `max_regret` heuristic selects the variable where the difference between the best and second-best value is largest. The intuition is that if one value is clearly superior, the decision is "critical"—making the wrong choice is costly, so the solver should resolve it early.

```
int_search(all_moves, max_regret, indomain_random, complete)
```

Example. Consider a scheduling-like scenario where choosing $idx1_0 = 2$ would satisfy 4 constraints immediately, while $idx1_0 = 3$ would only satisfy 1. The "regret" of not choosing 2 is high (3 constraints lost), so the heuristic prioritizes this variable.

4.1.7 Anti First-Fail

The `anti_first_fail` heuristic is the opposite of `first_fail`: it selects the variable with the largest domain. This is useful when early commitment to constrained variables leads to thrashing, as it allows more propagation to occur before making critical decisions.

```
int_search(all_moves, anti_first_fail, indomain_random, complete)
```

Example. If $D(idx1_0) = \{1, 2, 3, 4, 5\}$ and $D(idx1_1) = \{2, 3\}$, the heuristic selects $idx1_0$ first. By assigning the less constrained variable, constraint propagation may further reduce $D(idx1_1)$ before the solver commits to a value.

4.2 Value Selection Heuristics

4.2.1 Random Selection

The `indomain_random` heuristic selects a random value from the domain. Combined with restarts, this provides diversification—different restart iterations explore different parts of the search space, reducing the probability of getting stuck.

4.2.2 Minimum Value

The `indomain_min` heuristic always selects the smallest value in the domain. This is deterministic and reproducible, but may get stuck without restarts if the optimal solution requires larger values.

4.2.3 Domain Splitting

The `indomain_split` heuristic does not enumerate individual values. Instead, it bisects the domain: first try $x \leq \text{mid}$, then $x > \text{mid}$. This achieves logarithmic depth instead of linear.

```
int_search(all_moves, dom_w_deg, indomain_split, complete)
```

Example. For a variable with domain $\{1, \ldots, 30\}$, traditional enumeration might try up to 30 values. With binary splitting, the solver reaches any value in at most $\lceil \log_2 30 \rceil = 5$ decisions.

4.3 Restart Policies

4.3.1 Luby Restart

The Luby sequence is $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$ With `restart_luby(250)`, the solver restarts after $250 \times L$ failures, where L is the current Luby number. This sequence is proven to be **universally optimal** for randomized algorithms—no other restart sequence is asymptotically better.

4.3.2 Geometric Restart

With `restart_geometric(1.5, 100)`, the restart sequence is $100, 150, 225, 337, \ldots$ (multiply by 1.5 each iteration). This starts aggressively with short restarts and becomes more patient over time.

4.3.3 Linear Restart

With `restart_linear(250)`, the restart sequence is $250, 500, 750, 1000, \ldots$ (add 250 each iteration). This provides predictable, steady growth but is less adaptive than Luby.

4.3.4 No Restart

Without any restart annotation, the solver performs complete enumeration. This can be effective for small instances but risks getting stuck in barren subtrees for larger problems.

4.4 Complete Strategy List

#	Name	Variable	Value	Restart
1	default	(Gecode)	(Gecode)	Luby(250)
2	firstfail	first_fail	random	Luby(250)
3	domwdeg	dom_w_deg	random	Luby(250)
4	smallest	smallest	min	Luby(250)
5	mostconstrained	most_constrained	random	Luby(250)
6	maxregret	max_regret	random	Luby(250)
7	antifirstfail	anti_first_fail	random	Luby(250)
8	domwdeg_split	dom_w_deg	split	Luby(250)
9	firstfail_split	first_fail	split	Luby(250)
10	geometric	dom_w_deg	random	Geometric(1.5, 100)
11	linear	dom_w_deg	random	Linear(250)
12	norestart	dom_w_deg	random	None

Table 2: All 12 search strategies tested.

4.5 Parallel Execution

To reduce benchmark time, strategies are executed in parallel by leveraging multi-threading: each strategy is run in a separate thread, allowing simultaneous comparison on the same instance. Results are collected in a shared CSV file with appropriate locking to ensure data consistency.

5 Experimental Results

This section presents the experimental evaluation of the 12 search strategies described in Section 4, comparing their performance across instances of varying complexity.

5.1 Experimental Setup

The benchmarks were executed under the following conditions:

- **Solver:** Gecode via MiniZinc 2.x
- **Timeout:** 300 seconds per instance (as specified by the project requirements)
- **Instances:** 60 random permutations organized as follows:
  - Vector sizes: $N \in \{5, 10, 15, 20, 25, 30\}$
  - 10 different permutations for each size
- **Strategies tested:** 12 (see Table 2)
- **Total executions:** $60 \times 12 = 720$

5.2 Reliability Analysis

Figure 1 shows the number of instances solved (out of 10) for each strategy at different vector sizes.

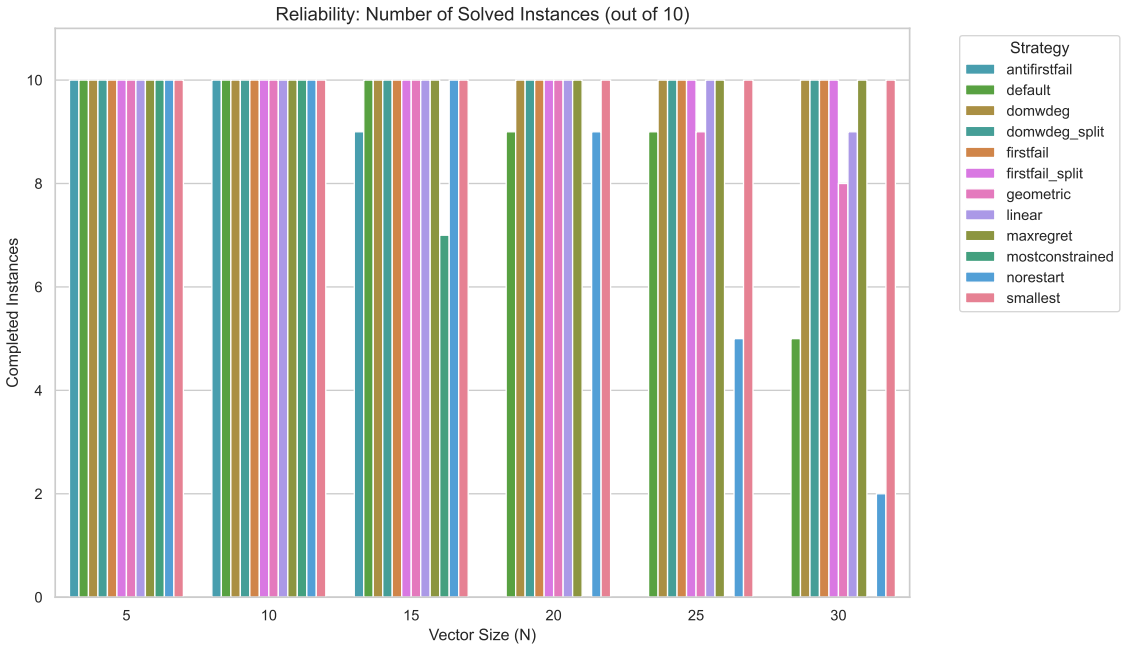

Figure 1: Number of solved instances per strategy and vector size. All strategies achieve 100% success for small instances ($N \leq 10$), but significant differences emerge for larger problems.

The success rate heatmap (Figure 2) provides a clearer view of strategy reliability.

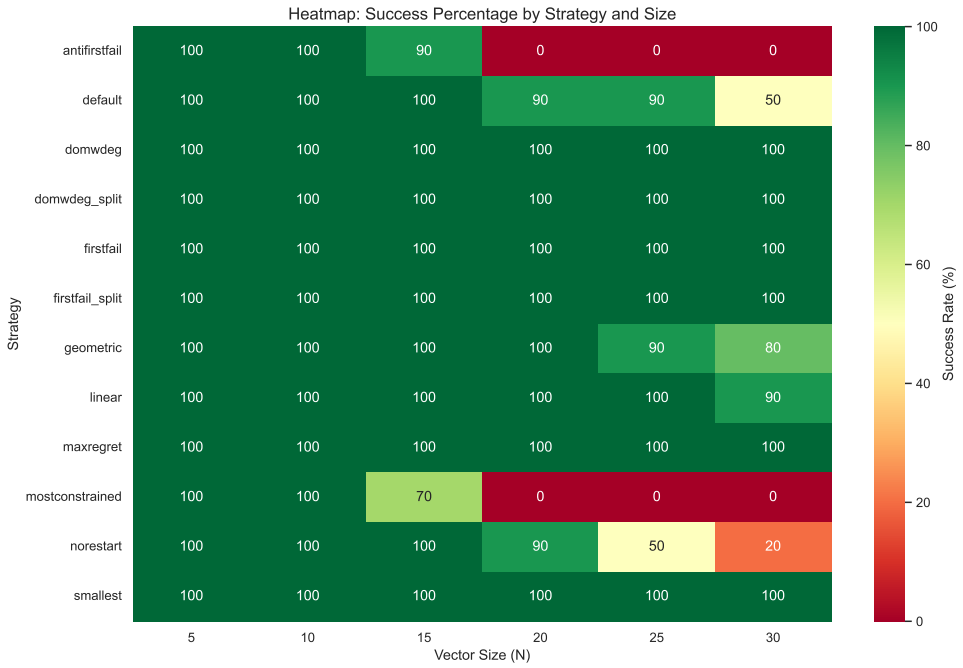

Figure 2: Success rate (%) by strategy and vector size. Green indicates 100% success, red indicates complete failure (0%).

Six strategies maintain perfect reliability (100% success rate) across all instance sizes: `smallest`, `firstfail`, `firstfail_split`, `domwdeg`, `domwdeg_split`, and `maxregret`. These strategies prove robust even for the most challenging instances with $N = 30$.

On the opposite end, the `mostconstrained` and `antifirstfail` strategies fail completely for $N \geq 20$, reaching timeout on all instances. This suggests that these variable selection heuristics are fundamentally unsuitable for our problem structure.

The `norestart` strategy shows progressive degradation, dropping from 100% success ($N \leq 15$) to just 20% at $N = 30$. This confirms the critical importance of restart policies for avoiding search stagnation in larger instances.

Finally, the default Gecode configuration maintains reasonable performance for medium-sized instances but drops to 50% success rate at $N = 30$, indicating that customized search strategies can significantly outperform the solver's default behavior.

5.3 Performance Analysis

Figure 3 presents the average solving time for each strategy-size combination.

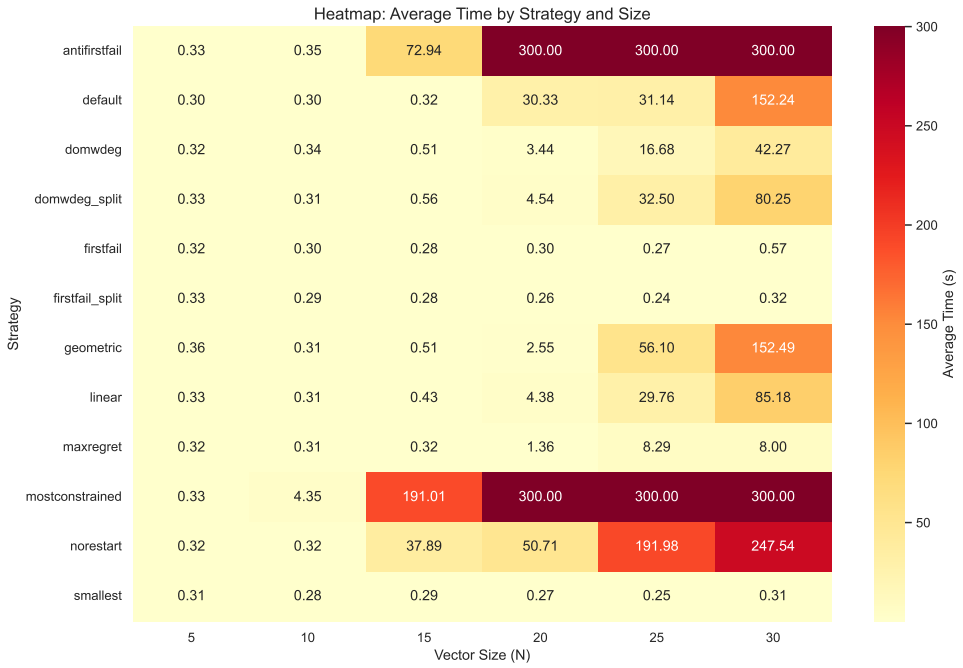

Figure 3: Average solving time (seconds) by strategy and vector size. Darker colors indicate longer times; values of 300s represent timeout.

The `smallest`, `firstfail`, and `firstfail_split` strategies consistently achieve sub-second solving times across all instance sizes, with times ranging from 0.31s to 0.57s even for $N = 30$. These strategies demonstrate excellent scalability.

In contrast, strategies like `geometric`, `linear`, and `domwdeg_split` show significant time increases for larger instances, reaching 80–150 seconds at $N = 30$. This suggests that certain restart policies and value selection heuristics introduce overhead that becomes problematic as problem size grows.

The `mostconstrained` strategy already struggles at $N = 15$ with an average time of 191 seconds, and times out completely for $N \geq 20$. This is a particularly important finding, as the "most constrained" heuristic is often considered a reasonable default choice.

Comparing `norestart` with other strategies clearly demonstrates the benefit of restart mechanisms: without restarts, solving time explodes from 0.32s at $N = 5$ to 247.54s at $N = 30$. This three-orders-of-magnitude increase underscores how easily the solver can get trapped in unproductive search regions.

5.4 Strategy Ranking

Figure 4 presents the overall ranking of strategies based on average solving time (computed only on successfully solved instances).

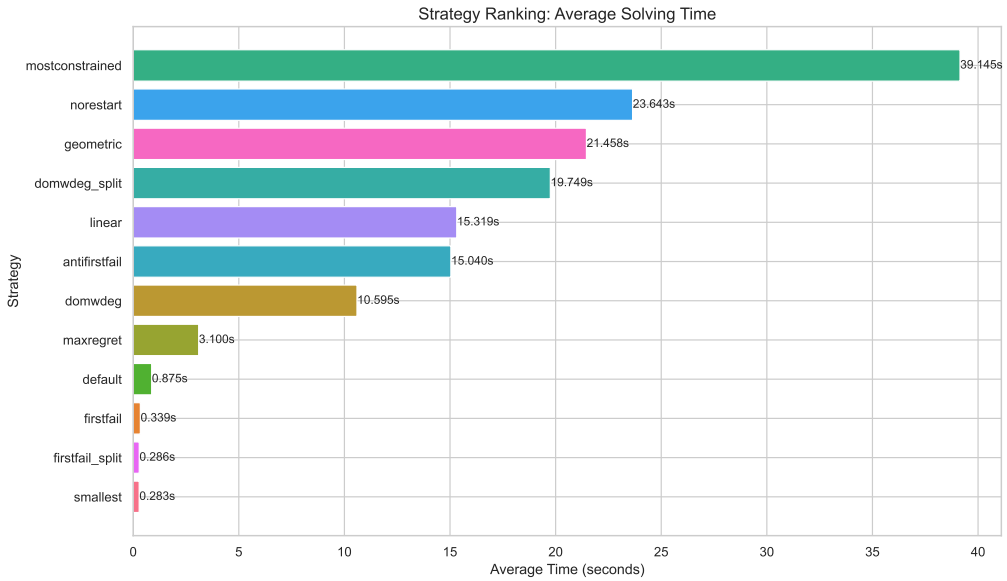

Figure 4: Strategy ranking by average solving time. Lower times indicate better performance.

The ranking reveals three distinct performance tiers. The **top tier** (under 1 second average) includes `smallest` (0.283s), `firstfail_split` (0.286s), `firstfail` (0.339s), and `default` (0.875s). These strategies are suitable for real-time applications where response time is critical.

The **middle tier** (1-15 seconds average) comprises `maxregret` (3.1s), `domwdeg` (10.6s), `antifirstfail` (15.0s), and `linear` (15.3s). While slower, these strategies still complete within reasonable time bounds for most applications.

The **bottom tier** (over 15 seconds average) includes `domwdeg_split` (19.7s), `geometric` (21.5s), `norestart` (23.6s), and `mostconstrained` (39.1s). These strategies are not recommended for production use, as their performance degrades significantly with problem size.

5.5 Discussion

The experimental results lead to several important conclusions regarding search strategy design for the sorting problem.

Variable selection is crucial. The `smallest` heuristic, which selects the move variable with the smallest current domain, proves to be the most effective choice. This is intuitive for our problem: moves with smaller domains are closer to being fixed and represent more constrained decisions. By committing to these variables first, the solver quickly narrows the search space.

The first-fail principle works. Both `firstfail` and `firstfail_split` confirm the effectiveness of the first-fail principle for this problem. Prioritizing variables with smaller domains leads to earlier detection of dead ends, allowing the solver to backtrack sooner and avoid exploring fruitless subtrees.

Restarts are essential. The dramatic performance difference between strategies with and without restarts demonstrates that restart mechanisms are critical for avoiding search stagnation. The `norestart` strategy's 80% failure rate at $N = 30$ versus 100% success for strategies with Luby restarts provides compelling evidence for this conclusion.

Domain-weighted degree has mixed results. While `domwdeg` maintains a 100% success rate, its average time is significantly higher than simpler heuristics like `firstfail`. The learning overhead of tracking constraint weights may not pay off for instances of this size, where simpler static heuristics suffice.

Anti-first-fail fails. The `antifirstfail` strategy, which prioritizes variables with larger domains, performs poorly as expected. This confirms that our problem benefits from early commitment to constrained choices rather than deferring difficult decisions.

Most-constrained is misleading. Despite its name, the `mostconstrained` heuristic (selecting the variable involved in the most constraints) fails completely for larger instances. This suggests that constraint count is not a good proxy for variable importance in our model, where all move variables participate in similar constraint structures.

5.6 Future Work

A natural direction for future work is to compare the constraint programming approach not only in terms of execution time, but also in terms of memory accesses and the number of swaps performed, with respect to classical sorting algorithms such as quicksort, mergesort, or bubble sort. This would provide a more meaningful comparison of the computational effort required by different paradigms, especially since classical algorithms are highly optimized for time but may differ in the number of elementary operations.

Another interesting extension would be to experiment with new constraints or to adapt the model to handle vectors with arbitrary values, not just permutations of 1 to N. This would allow the approach to be applied to a wider range of sorting problems, including those with repeated or missing values, and could require the development of new constraint formulations or search strategies.

6 Conclusions

This work shows that constraint programming, when paired with well-designed constraints and effective search strategies, can efficiently solve the sorting-as-planning problem even for challenging instances. The best results are achieved by combining strong propagation, mathematical insights (such as cycle decomposition), and restart policies. While no single strategy is always optimal, the "smallest" heuristic with Luby restarts proved both fast and reliable. The full code is available at: `https://github.com/raffaelecrafa/sorting-as-a-planning-problem`