WSP Solver Suite: A Comparative Analysis of Constraint-Based Approaches

Introduction

The Workflow Satisfiability Problem (WSP) is a fundamental challenge in access control and workflow management systems. It involves determining whether there exists an assignment of users to workflow steps such that all specified constraints are satisfied. These constraints typically encompass authorisations, user capacities, separation of duties, binding of duties, and other complex organisational policies. Efficiently solving WSP is crucial for ensuring both operational effectiveness and security compliance within organizations.

In this coursework, we explore three distinct solver implementations designed to address the WSP using different constraint encoding strategies. The solvers, **solver_combinatorial**, **solver_symmetry**, and **solver_doreen**, are developed using Python in conjunction with Google's OR-Tools library, which provides powerful capabilities for constraint programming and combinatorial optimisation. Each solver approaches the problem with unique formulations and constraint handling mechanisms, thereby offering varied performance and scalability characteristics.

While OR-Tools is a robust framework, certain functionalities may exhibit limitations, especially when dealing with highly symmetrical constraint structures or exceedingly large problem instances. For instance, the combinatorial explosion in the number of possible assignments can lead to significant computational overhead, necessitating advanced techniques such as symmetry breaking to enhance solver efficiency. This report delves into the mathematical formulations underpinning each solver, explores alternative approaches, discusses implementation nuances, and evaluates solver performance across diverse problem instances.

Formulations of the Problem(s)

The Workflow Satisfiability Problem (WSP) is formalized by defining domain elements, predicates, functions, and variables that capture the essence of workflow steps, users, and the constraints governing their assignments. The following table delineates these symbols, categorizing them by type, arity, and providing concise descriptions.

Domain Definitions and Functions

Table 1: Definitions of Predicates, Functions, and Variables

Symbol	Type	Arity	Description
IsStep(s)	Predicate	1	True if s is a workflow step.
IsUser(u)	Predicate	1	True if u is a user.
assignment(s)	Function	1	Returns the user assigned to step s.
Authorised(u,s)	Predicate	2	True if user u is authorised to perform step s.
capacity(u)	Function	1	Returns the maximum number of steps user u can perform (user-capacity constraint).
OneTeamConstraint(S,T)	Function	2	Enforces that steps in set S are assigned to exactly one team from the set T.
AtMostK(k,S)	Function	2	Enforces that at most k distinct users are assigned to the set of steps S.
SeparationOfDuty(s_1,s_2)	Predicate	2	Steps s_1 and s_2 must be assigned to different users.
BindingOfDuty(s_1,s_2)	Predicate	2	Steps s_1 and s_2 must be assigned to the same user.
user_assignment[s][u]	Boolean	2	True if step s is assigned to user u. (Used specifically in **solver_doreen**)

Constraint Formulation

General Constraint: Each Step Assigned to Exactly One User

Each workflow step must be assigned to exactly one user to ensure accountability and clarity in task delegation.

$$\forall s\ (\text{IsStep}(s) \rightarrow \exists!u\ (\text{IsUser}(u) \wedge \text{assignment}(s) = u))$$

Solver Implementations:

- **solver_combinatorial** and **solver_symmetry**:
  Utilize integer variables assignment(s) ranging over the set of user IDs. The constraint is implemented by enforcing that for each step s, the assignment variable takes exactly one value from the user set.
- **solver_doreen**:
  Employs boolean variables user_assignment[s][u] where user_assignment[s][u] = True

indicates that step s is assigned to user u. The constraint ensures that for each step s, exactly one user_assignment[s][u] is true.

1. Authorisations

A step s can only be performed by users who are authorised to do so, ensuring compliance with access control policies.

$$\forall u,s\ (\neg \text{Authorised}(u,s) \rightarrow \text{assignment}(s) \neq u)$$

Solver Implementations:

- **solver_combinatorial** and **solver_symmetry**:
  For each unauthorized user-step pair (u,s), the constraint assignment(s) ≠ u is added, preventing assignment violations.
- **solver_doreen**:
  Boolean variables user_assignment[s][u] are set to false for unauthorized assignments, effectively removing those possibilities from the solution space.

2. Separation-of-Duty

Certain pairs of steps must be executed by different users to prevent conflicts of interest and enhance security.

$$\forall s_1,s_2\ (\text{SeparationOfDuty}(s_1,s_2) \rightarrow \text{assignment}(s_1) \neq \text{assignment}(s_2))$$

Solver Implementations:

- **All Solvers**:
  For each pair of steps s_1, s_2 under separation-of-duty, the constraint assignment(s_1) ≠ assignment(s_2) is enforced, ensuring distinct user assignments.

3. Binding-of-Duty

Certain pairs of steps must be executed by the same user to maintain consistency and accountability.

$$\forall s_1,s_2\ (\text{BindingOfDuty}(s_1,s_2) \rightarrow \text{assignment}(s_1)=\text{assignment}(s_2))$$

Solver Implementations:

- **All Solvers**:
  For each pair of steps s_1, s_2 under binding-of-duty, the constraint assignment(s_1) = assignment(s_2) is enforced, ensuring identical user assignments.

4. At-most-k Distinct Users

Within a specified set of steps, the number of distinct users assigned should not exceed a predefined limit, promoting efficiency and resource optimization.

$$\text{AtMostK}(k,S) \equiv |\{\text{assignment}(s) | s \in S\}| \leq k$$

Solver Implementations:

- **solver_combinatorial**:
  Utilizes a combinatorial encoding by enumerating all possible combinations of k+1 steps

within S and enforcing that for each combination, at least one pair of steps must share the same user.

- **solver_symmetry**:
  Introduces auxiliary integer variables representing distinct users and enforces ordering constraints to break symmetries, thereby ensuring that no more than k distinct users are assigned to the set S.
- **solver_doreen**:
  Employs Boolean flags or additional constraints to limit the number of distinct users assigned to steps in S, thereby enforcing the at-most-k condition.

5. One-team Constraint

A group of steps must be executed by users belonging to exactly one predefined team, ensuring coherent task allocation within designated user groups.

$$\text{OneTeamConstraint}(S, \{T_1, T_2, \ldots, T_r\}) \equiv \exists i \, (\bigwedge_{s \in S} \text{assignment}(s) \in T_i)$$

Solver Implementations:

- **solver_combinatorial** and **solver_symmetry**:
  Utilize *AddAllowedAssignments* in OR-Tools to define permissible user assignments for the group of steps based on the team definitions, thereby enforcing the one-team constraint.
- **solver_doreen**:
  Introduces boolean variables representing the selection of each team and links these to the user assignments, ensuring that all steps in S are assigned to users within the selected team.

6. User Capacity

Each user has a maximum capacity of steps they can perform, preventing overloading and ensuring fair distribution of tasks.

$$\forall u \sum_{s \,:\, \text{assignment}(s)=u} 1 \leq \text{capacity}(u)$$

Solver Implementations:

- **solver_combinatorial** and **solver_symmetry**:
  Implements the user-capacity constraint by summing the Boolean indicators for each step's assignment to a user and enforcing that the sum does not exceed the user's capacity.
- **solver_doreen**:
  Applies the user-capacity constraint by summing the Boolean variables user_assignment[s][u] for each step s and ensuring the total does not exceed the predefined capacity for user u.

Alternative Formulation(s) and Solution(s)

The Workflow Satisfiability Problem (WSP) can be approached through various constraint encoding and solving strategies. In this coursework, three distinct solvers; **solver_combinatorial**, **solver_symmetry**, and **solver_doreen**, are implemented, each embodying unique methodologies for constraint formulation and solution exploration. This section provides a comprehensive comparative analysis of these solvers, elucidating their respective advantages, disadvantages, strengths, and weaknesses. Additionally, we identify specific scenarios where each solver excels or underperforms, thereby offering insights into their optimal application contexts.

Comparative Analysis of Solvers

1. Solver Combinatorial (solver_combinatorial)

Overview: The **solver_combinatorial** adopts a direct combinatorial constraint programming approach using Google's OR-Tools. It explicitly encodes constraints by enumerating possible combinations, particularly for complex constraints like "At-most-k."

Advantages:

- **Simplicity and Directness:** The straightforward encoding of constraints makes the solver easy to understand and implement. Each constraint directly maps to an OR-Tools constraint without requiring auxiliary variables or complex transformations.
- **Flexibility:** Capable of handling a wide range of constraints without specialized encoding techniques, making it versatile for various WSP instances.
- **Transparency:** The explicit enumeration of constraint combinations facilitates easier debugging and verification of constraint formulations.

Disadvantages:

- **Scalability Issues:** The combinatorial nature leads to exponential growth in the number of constraints, especially for "At-most-k" and "One-team" constraints. As the number of steps or the value of k increases, the solver's performance degrades significantly.
- **Redundancy and Symmetry:** Without explicit symmetry-breaking mechanisms, the solver may explore numerous symmetrical solutions, resulting in redundant computations and prolonged search times.

Strengths:

- **Effective for Small to Medium Instances:** Performs adequately for WSP instances with a limited number of steps and constraints, where the combinatorial explosion is manageable.
- **Ease of Implementation:** Minimal overhead in setting up constraints, making it suitable for rapid prototyping and educational purposes.

Weaknesses:

- **Inefficiency with Large Constraints:** Struggles with large-scale problems due to the extensive number of constraint combinations, leading to high computational overhead.
- **Lack of Advanced Optimization:** Does not incorporate advanced techniques like symmetry-breaking or heuristic-driven search, limiting its efficiency in complex scenarios.

Optimal Application Scenarios:

- **Educational and Prototyping Use:** Ideal for teaching purposes and initial experimentation with WSP formulations due to its simplicity.
- **Large-Scale Problems:** Suitable for instances with a high number of steps and constraints, and a low k-value.

2. Solver Symmetry (solver_symmetry)

Overview: The **solver_symmetry** builds upon the combinatorial approach by integrating symmetry-breaking techniques to mitigate redundant solution exploration. It introduces auxiliary variables and ordering constraints to reduce the search space effectively.

Advantages:

- **Enhanced Efficiency:** By enforcing an order on user assignments and introducing auxiliary variables, the solver significantly reduces symmetrical redundancies, leading to faster solution times.
- **Improved Scalability:** More adept at handling larger problem instances compared to the combinatorial solver, thanks to the reduced search space.
- **Targeted Constraint Encoding:** Specialized encoding for "At-most-k" constraints and other symmetrical constraints optimizes performance for complex WSP instances.

Disadvantages:

- **Increased Complexity:** The introduction of auxiliary variables and ordering constraints adds layers of complexity to the model, making it more challenging to implement and understand.
- **Potential Overhead:** While symmetry-breaking reduces redundant searches, the added constraints themselves may introduce computational overhead, especially if not optimally formulated.

Strengths:

- **Superior Performance on Symmetrical Problems:** Excels in scenarios where multiple symmetrical solutions exist, as symmetry-breaking effectively prunes the search space.
- **Balanced Constraint Handling:** Manages complex constraints more efficiently by reducing unnecessary explorations, thereby enhancing overall solver performance.

Weaknesses:

- **Implementation Complexity:** The necessity to manage additional variables and constraints complicates the solver's implementation, potentially leading to maintenance challenges.
- **Limited Flexibility:** While effective for symmetrical constraints, the solver may require further enhancements to handle other types of complex constraints optimally.

Optimal Application Scenarios:

- **Medium to Large-Scale Problems:** Suitable for WSP instances with a higher number of steps and symmetrical constraints, where performance gains from symmetry-breaking are substantial.
- **Performance-Critical Applications:** Ideal for environments where solver efficiency is paramount, and the overhead of additional constraints is justified by the performance improvements.

3. Solver Doreen (solver_doreen)

The **solver_doreen** employs a hybrid approach, combining the strengths of OR-Tools and the Z3 theorem prover. It utilizes Boolean variables for user assignments and leverages logical reasoning capabilities to handle complex constraints more effectively.

Advantages:

- **Advanced Logical Constraint Handling:** The integration with Z3 allows for more sophisticated logical constraint formulations, enabling the solver to handle nuanced and conditional constraints with greater ease.
- **Efficient Solution Enumeration:** Boolean variable encoding facilitates faster convergence to feasible solutions, especially in logically dense constraint environments.
- **Hybrid Optimization:** Combines constraint programming and logical reasoning, offering a balanced approach that can leverage the strengths of both paradigms.

Disadvantages:

- **Increased Implementation Complexity:** The dual reliance on OR-Tools and Z3 introduces significant complexity, requiring careful synchronization between the two systems to maintain consistency in constraint handling.
- **Potential Performance Bottlenecks:** The hybrid nature may lead to unforeseen computational overheads, particularly if the integration between OR-Tools and Z3 is not optimally managed.
- **Dependency Management:** Requires managing dependencies for both OR-Tools and Z3, potentially complicating the deployment and execution environments.

Strengths:

- **Enhanced Constraint Flexibility:** Capable of expressing and resolving more intricate logical constraints that might be cumbersome in pure CP approaches.
- **Optimized Performance for Logical Constraints:** Excels in scenarios where constraints have complex logical interdependencies, leveraging Z3's theorem-proving capabilities for efficient resolution.
- **Scalable Logical Reasoning:** The combination allows for scalable handling of large and logically intensive WSP instances, potentially outperforming pure CP solvers in such contexts.

Weaknesses:

- **Implementation and Maintenance Challenges:** The need to maintain coherence between OR-Tools and Z3 constraints complicates development and ongoing maintenance.
- **Learning Curve:** Requires familiarity with both OR-Tools and Z3, posing a steeper learning curve compared to single-framework solvers.
- **Debugging Difficulties:** The hybrid approach can make debugging more challenging, as issues may arise from either OR-Tools or Z3 components.

Optimal Application Scenarios:

- **Complex Logical Constraints:** Ideal for WSP instances with highly interdependent and conditional constraints that benefit from advanced logical reasoning.
- **Small-Scale, Logically Dense Problems:** Suitable for environments where the problem size is small, and constraints are complex, necessitating a robust and flexible solving strategy.

Summary of Comparative Insights

Aspect	Solver Combinatorial	Solver Symmetry	Solver Doreen
Constraint Encoding	Direct combinatorial enumeration	Combinatorial with symmetry-breaking	Boolean variables with logical reasoning
Scalability	Superior for large, logically complex instances with small k	Improved for medium-large instances	Limited to small-medium instances
Implementation Complexity	Low	Medium	High
Handling Symmetries	No explicit handling	Effective symmetry-breaking	Indirect through logical constraints
Performance on Symmetrical Constraints	Good	Excellent	Poor
Flexibility in Constraint Types	Very High	Medium	High
Ease of Debugging	High	Medium	Low
Suitable for Educational Use	Excellent	Good	Moderate
Best Use Cases	Complex, large-scale WSP instances with intricate logical constraints with small k	Medium to large-scale, symmetrical WSP instances	Simple, small-scale WSP instances

The choice among **solver_combinatorial**, **solver_symmetry**, and **solver_doreen** hinges on the specific characteristics of the WSP instance at hand. For educational purposes and small-scale problems, the **solver_doreen** offers simplicity and ease of use. When dealing with medium to large-scale instances where symmetry and redundancy pose significant challenges, the **solver_symmetry** provides enhanced performance through effective symmetry-breaking techniques. Conversely, for highly complex and large-scale WSP instances with intricate logical dependencies, the **solver_combinatorial** emerges as the most robust solution. Understanding the strengths and limitations of each solver facilitates informed selection and optimal application within diverse workflow satisfiability contexts.

Implementation

The implementation of the Workflow Satisfiability Problem (WSP) solvers and associated tools encompasses a suite of Python scripts designed to encode constraints, solve instances, validate solutions, and provide an interactive graphical user interface (GUI) for user interaction. This section provides an exhaustive examination of each component, detailing their functionalities, interactions, and contributions to the overall system.

Overview of Components

The implementation consists of the following key components:

1. **Solver Modules**
    - solver_combinatorial.py
    - solver_symmetry.py
    - solver_doreen.py
2. **Validator Scripts**
    - validator.py
    - validator2.py
    - validator3.py
3. **Helper Module**
    - helper.py
4. **GUI Application**
    - complete.py
5. **Benchmarking Tool**
    - evaluation.py

Each of these components plays a pivotal role in ensuring the accurate encoding, solving, validation, and user-friendly interaction with WSP instances.

1. Solver Modules

a. Solver Combinatorial (solver_combinatorial.py)

The solver_combinatorial.py script encapsulates the combinatorial constraint programming approach to solving WSP. It leverages Google's OR-Tools CP-SAT solver to model and resolve workflow step assignments under specified constraints.

Key Features:

- **Constraint Encoding (Solver function):**
    - **Variables:**
        - Assigns an integer variable to each step, representing the user assigned to that step.
    - **Authorisation Constraints:**
        - Ensures that steps are only assigned to authorized users by restricting variable domains.
    - **Separation-of-Duty and Binding-of-Duty Constraints:**
        - Implements constraints that enforce steps to be assigned to different or the same users, respectively.
    - **At-most-k Constraints:**
        - Utilizes combinatorial enumeration to enforce that no more than kk distinct users are assigned to a specified set of steps.
    - **One-Team Constraints:**
        - Applies allowed assignments for steps based on predefined user teams, ensuring coherence in task allocation.
    - **User-Capacity Constraints:**
        - Limits the number of steps a user can be assigned, preventing overloading.
- **Solution Enumeration:**
    - Configures the solver to retrieve single or multiple solutions based on the max_solutions parameter.
    - Utilizes a custom callback (MultiSolCallback) to collect and store feasible solutions during the search process.
- **Execution and Output:**
    - Measures execution time and records solver status.
    - Outputs the first solution and, if applicable, additional solutions, distinguishing between unique and multiple solutions.

The solver directly translates each constraint type into OR-Tools' model, maintaining transparency and simplicity. However, the explicit combinatorial handling of "At-most-k" constraints can lead to a large number of additional constraints, impacting performance for larger instances.

b. Solver Symmetry (solver_symmetry.py)

The solver_symmetry.py script enhances the combinatorial approach by incorporating symmetry-breaking techniques to reduce redundant solution explorations. This optimization is critical for improving solver efficiency, particularly in large and symmetrical WSP instances.

Key Features:

- **Constraint Encoding (Solver function):**
  - **Variables:**
    - Utilizes integer variables for step assignments, as in the combinatorial solver.
  - **Authorisation Constraints:**
    - Enforces step assignments to authorized users by restricting variable domains.
  - **Separation-of-Duty and Binding-of-Duty Constraints:**
    - Implements constraints to ensure steps are assigned to different or the same users, respectively.
  - **At-most-k Constraints:**
    - Introduces auxiliary integer variables representing distinct users and enforces ordering constraints to break symmetries.
  - **One-Team Constraints:**
    - Applies allowed assignments for user teams, similar to the combinatorial solver, but with additional considerations for symmetry.
  - **User-Capacity Constraints:**
    - Restricts the number of steps a user can be assigned, aligning with user capacity constraints.
- **Symmetry Breaking:**
  - Implements ordering constraints on auxiliary variables to ensure that user assignments follow a non-decreasing order, effectively reducing symmetrical redundancies.
- **Solution Enumeration:**
  - Similar to the combinatorial solver, it supports single and multiple solution retrieval through a custom callback (MultipleSolCollector).
- **Execution and Output:**
  - Tracks execution time and solver status, outputting feasible solutions while mitigating redundant explorations through symmetry-breaking.

By introducing auxiliary variables and ordering constraints, the symmetry solver effectively prunes the search space, enhancing performance. However, the added complexity requires careful management of additional variables and constraints to maintain solver efficiency without introducing new bottlenecks.

c. Solver Doreen (solver_doreen.py)

Functionality: The solver_doreen.py script employs a hybrid approach, integrating both OR-Tools and the Z3 theorem prover to address WSP constraints. This combination harnesses the strengths of constraint programming and logical reasoning to efficiently solve complex WSP instances.

Key Features:

- **Constraint Encoding (Solver function):**
    - **Variables:**
        - Utilizes Boolean variables for user-step assignments (user_assignment[s][u]) enabling granular control over assignments.
    - **Authorisation Constraints:**
        - Sets Boolean variables to false for unauthorized user-step pairs, ensuring that only authorized assignments are considered.
    - **Separation-of-Duty and Binding-of-Duty Constraints:**
        - Implements separation-of-duty by ensuring that no user is assigned to both steps.
        - Enforces binding-of-duty by linking user assignments between steps.
    - **At-most-k Constraints:**
        - Employs Boolean flags to track the number of distinct users assigned to a set of steps, enforcing the at-most-k condition through summation constraints.
    - **One-Team Constraints:**
        - Introduces Boolean variables representing team selections and ensures that all steps within a team are assigned to users belonging to the selected team.
    - **User-Capacity Constraints:**
        - Limits the number of steps a user can perform by summing their assigned Boolean variables and enforcing upper bounds.
- **Solution Enumeration:**
    - Supports single and multiple solution retrieval using a custom callback (MultiSolutionCallback), similar to the other solvers.
    - Collects and organizes solutions for output, distinguishing between single and multiple feasible assignments.
- **Execution and Output:**
    - Measures execution time and solver status.
    - Outputs feasible solutions, including multiple assignments where applicable, while ensuring that the hybrid encoding remains consistent and efficient.

The integration of Z3 enhances the solver's ability to handle complex logical constraints more effectively. Boolean variable encoding provides finer control over user assignments, facilitating efficient constraint resolution. However, the hybrid approach introduces significant implementation complexity, requiring meticulous synchronization between OR-Tools and Z3 to maintain constraint consistency and solver efficiency.

2. Validator Scripts

Validation is a critical component ensuring that solutions produced by the solvers adhere strictly to the defined constraints. Three validator scripts; validator.py, validator2.py, and validator3.py, are implemented to offer comprehensive verification through different methodologies.

a. Validator (validator.py)

Functionality: The validator.py script performs validation of solver-generated solutions by checking each constraint type against the assigned user-step mappings. It ensures that no constraints are violated, thereby confirming the solution's correctness.

Key Features:

- **Constraint Validation Functions:**
    - **validate_authorisations:** Ensures that each step is assigned only to authorized users.
    - **validate_separation_of_duty:** Confirms that steps under separation-of-duty are assigned to different users.
    - **validate_binding_of_duty:** Verifies that steps under binding-of-duty are assigned to the same user.
    - **validate_at_most_k:** Checks that the number of distinct users assigned to specified steps does not exceed kk.
    - **validate_one_team:** Ensures that steps under one-team constraints are assigned to users within the same team.
    - **validate_user_capacity:** Confirms that no user is assigned more steps than their capacity allows.
- **Execution and Output:**
    - Aggregates validation results, indicating whether the solution is valid.
    - Provides detailed error messages for any constraint violations detected.

The validator script methodically checks each constraint type, offering precise feedback on any violations. This granularity facilitates effective debugging and ensures the integrity of solver-generated solutions.

b. Validator2 (validator2.py)

The validator2.py script offers a structured and object-oriented approach to solution validation. It encapsulates the problem instance and solution within a WSPValidator class, providing modular and reusable validation methods.

Key Features:

- **Class Structure (WSPValidator):**
    - **Attributes:**
        - Stores counts of steps, users, and constraints.
        - Maintains mappings for authorizations, separation-of-duty, binding-of-duty, at-most-k, one-team, and user-capacity constraints.
    - **Methods:**
        - **parse_problem:** Parses the problem instance file, populating constraint mappings.
        - **parse_solution:** Reads the solution file, mapping steps to users.
        - **validate_solution:** Performs comprehensive validation by invoking specific constraint validation methods.
- **Constraint Validation Functions:**
    - **validate_authorisations:** Similar to validator.py, ensuring authorized user-step assignments.
    - **validate_separation_of_duty:** Ensures separation-of-duty constraints are upheld.
    - **validate_binding_of_duty:** Verifies binding-of-duty constraints.
    - **validate_at_most_k:** Checks at-most-k constraints across specified step sets.
    - **validate_one_team:** Validates one-team constraints, ensuring team coherence.
    - **validate_user_capacity:** Confirms adherence to user-capacity constraints.
- **Execution and Output:**
    - Integrates validation results, outputting whether the solution is valid along with detailed error messages if violations are present.

The object-oriented design promotes modularity and reusability, allowing for easier extension and maintenance of validation functionalities. By encapsulating validation logic within a class, the script facilitates cleaner code organization and potential integration with other components.

c. Validator3 (validator3.py)

The validator3.py script presents a streamlined and efficient approach to solution validation, utilizing a combination of regular expressions and straightforward mapping techniques to verify constraint adherence.

Key Features:

- **Constraint Validation Functions:**
    - **validate_authorizations:** Ensures user-step assignments comply with authorizations.
    - **validate_separation_of_duty:** Confirms separation-of-duty constraints are respected.
    - **validate_binding_of_duty:** Verifies binding-of-duty constraints.
    - **validate_at_most_k:** Checks that the number of distinct users for specified steps does not exceed kk.
    - **validate_one_team:** Validates one-team constraints, ensuring assignments align with team definitions.
    - **validate_user_capacity:** Ensures no user exceeds their step assignment capacity.
- **Execution and Output:**
    - Aggregates and reports validation results, highlighting any constraint violations with descriptive error messages.

Validator3 offers a concise and efficient validation mechanism, leveraging regular expressions for effective parsing and straightforward mapping for constraint verification. This approach ensures rapid validation while maintaining clarity and simplicity in code structure.

3. Helper Module (helper.py)

The helper.py script provides auxiliary functions that support the core functionalities of the solvers and validators. It includes utility functions for transforming solver outputs into user-friendly formats.

Key Features:

- **transform_output Function:**
    - Converts solver result dictionaries into formatted string representations, facilitating easier readability and integration with the GUI.
    - Handles both single and multiple solution scenarios, ensuring that all feasible assignments are appropriately formatted for display.

Implementation Insights: By centralizing common utility functions, helper.py promotes code reusability and simplifies the integration of solver outputs with other components, such as the GUI application. This modularity enhances maintainability and scalability of the overall system.

4. GUI Application (complete.py)

The complete.py script serves as the central graphical user interface (GUI) for interacting with the WSP solvers and validators. Developed using Tkinter, it provides an intuitive platform for defining workflow steps, users, constraints, executing solvers, and visualizing results.

Detailed Workflow:

1. **Class Structure (WorkflowSolverApp Class):**
    - **Initialization (__init__ Method):**
        - Sets up the main application window with a title and fixed dimensions.
        - Initializes data structures to store workflow steps, users, constraints, and user capacities.
        - Defines a timeout parameter to limit solver execution times.
2. **Widget Creation (create_widgets Method):**
    - **Notebook Interface:**
        - Employs a ttk.Notebook to organize the GUI into three primary tabs: Input, Constraints, and Results.
    - **Input Tab (build_input_tab Method):**
        - **Solver Selection:**
            - Features a dropdown menu allowing users to select among the three solvers: solver_combinatorial, solver_symmetry, and solver_doreen.
        - **Steps and Users Management:**
            - **Steps Listbox:** Displays the list of workflow steps. Users can add or remove steps using the provided buttons.
            - **Users Listbox:** Displays the list of users. Users can add or remove users using the provided buttons.
        - **Import/Export Functionality:**
            - **Import Button:** Allows users to import a problem instance from a text file, automatically populating steps, users, and constraints.
            - **Export Button:** Enables users to export the current problem instance configuration to a text file for external use or sharing.
    - **Constraints Tab (build_constraints_tab Method):**
        - **Constraint Builders:**
            - Provides buttons to add various constraint types:
                - **Add Authorisation Constraint:** Opens a dialog for selecting a user and defining authorized steps.
                - **Add Separation-of-Duty Constraint:** Opens a dialog for selecting two steps that must be assigned to different users.
                - **Add Binding-of-Duty Constraint:** Opens a dialog for selecting two steps that must be assigned to the same user.

        - **Add At-Most-k Constraint:** Opens a dialog for selecting a set of steps and defining the maximum number of distinct users.
        - **Add One-Team Constraint:** Opens a dialog for selecting a set of steps and defining user teams.
        - **Add User Capacity Constraint:** Opens a dialog for specifying the maximum number of steps a user can be assigned.
      - **Remove Constraint Button:** Allows users to remove selected constraints from the constraints treeview.
    - **Constraints Treeview:**
      - Displays all added constraints in a structured table format, categorizing them by constraint type and details. Facilitates easy visualization and management of constraints.
  - **Results Tab (build_results_tab Method):**
    - **Solve Button:** Initiates the solving process using the selected solver and defined problem instance. Executes the solver in a separate thread to maintain GUI responsiveness.
    - **Validation Checkbox:** Option to enable automatic validation of solutions post-solving, invoking all three validator scripts.
    - **Max Solutions Spinbox:** Allows users to specify the number of feasible solutions to retrieve, enabling exploration of multiple assignments.
    - **Progress Bar:** Visual indicator displaying the progress of the solving process, updating in real-time as the solver progresses.
    - **Info Label:** Provides informational messages, such as the number of solutions found and execution time, keeping users informed about the solver's status.
    - **Solution Dropdown:** Enables users to select among multiple solutions, if available, for detailed examination.
    - **Results Treeview:** Presents the selected solution in a structured format, mapping each step to its assigned user for easy interpretation.
    - **Validation Notebooks:**
      - Hosts validation results from the three validator scripts (validator.py, validator2.py, validator3.py), displaying detailed feedback on solution correctness within separate tabs for each validator.

3. **Constraint Management Dialogs:**
   - **Authorisation Constraint Dialog (AuthorisationConstraintDialog Class):**
     - Facilitates the selection of a user and the steps they are authorized to perform.
     - Ensures that users can only assign authorized steps to each user, maintaining compliance with access control policies.
   - **Separation-of-Duty Constraint Dialog (SeparationConstraintDialog Class):**
     - Enables the selection of two distinct steps that must be assigned to different users, enforcing separation-of-duty policies.
   - **Binding-of-Duty Constraint Dialog (BindingConstraintDialog Class):**

        - Allows the selection of two distinct steps that must be assigned to the same user, ensuring binding-of-duty requirements are met.
    - **At-Most-k Constraint Dialog (AtMostKConstraintDialog Class):**
        - Permits the selection of a set of steps and the definition of the maximum number of distinct users allowed, managing resource allocation efficiently.
    - **One-Team Constraint Dialog (OneTeamConstraintDialog Class):**
        - Supports the selection of a set of steps and the definition of user teams, ensuring that steps are coherently assigned within predefined user groups.
    - **Team Dialog (TeamDialog Class):**
        - Facilitates the definition of individual teams by allowing the selection of multiple users, enabling structured team-based assignments.
    - **User Capacity Constraint Dialog (UserCapacityConstraintDialog Class):**
        - Allows users to specify the maximum number of steps a user can be assigned, preventing overloading and ensuring fair distribution of tasks.
4. **Solver Execution (solve_problem and run_solver Methods):**
    - **Solver Invocation:**
        - Constructs a temporary problem instance file based on the defined steps, users, and constraints.
        - Dynamically imports the selected solver module and executes it with the temporary problem file and specified max_solutions.
    - **Solution Handling:**
        - Collects solver outputs, including execution time and solution assignments.
        - Differentiates between single and multiple solutions, enabling users to navigate and select among feasible assignments.
    - **Solution Display:**
        - Presents the first solution in the results treeview by default.
        - Populates the solution dropdown with available solutions, allowing users to switch between them for detailed examination.
    - **Solution Validation:**
        - If validation is enabled, runs all three validator scripts on each solution.
        - Displays validation results within the validation notebook, providing comprehensive feedback on solution correctness.
5. **Import and Export Functionality:**
    - **Import Problem:**
        - Allows users to import problem instances from existing text files.
        - Automatically populates the GUI with imported steps, users, and constraints, facilitating quick setup of known problem instances.
    - **Export Problem:**

- Enables users to export the current problem instance configuration to a text file.
- Supports external sharing, documentation, and reuse of defined WSP instances.

6. **Constraint Treeview Management:**
    - **Adding Constraints:**
        - Each constraint type has a dedicated dialog ensuring accurate and intuitive constraint definitions.
        - Constraints are added to the constraints treeview, categorized by type and detailed description.
    - **Removing Constraints:**
        - Users can select and remove constraints from the treeview, allowing dynamic modifications to the problem instance.
        - Ensures that constraint removals are reflected accurately in the underlying problem definition.
7. **Solution Visualization and Validation:**
    - **Results Treeview:**
        - Displays step-to-user assignments in a clear, tabular format, enabling easy interpretation of solutions.
    - **Validation Results:**
        - Integrates outputs from all validator scripts, presenting detailed validation feedback within the GUI.
        - Provides users with immediate confirmation of solution correctness, enhancing trust in solver outputs.
8. **Execution Flow:**
    - Users define workflow steps, users, and constraints through the GUI.
    - Upon initiating the solve process, the application constructs a problem instance file and invokes the selected solver.
    - The solver processes the problem, returning feasible solutions which are displayed within the GUI.
    - If enabled, solutions are validated, and validation results are presented, ensuring adherence to all constraints.

The GUI application orchestrates the entire WSP solving and validation process, providing an accessible and interactive interface for users. By integrating solver execution, solution display, and validation within a cohesive platform, it streamlines the workflow for defining, solving, and verifying WSP instances. The modular design, coupled with comprehensive constraint management dialogs, ensures flexibility and ease of use, catering to both novice users and advanced practitioners.

5. Benchmarking Tool (evaluation.py)

The evaluation.py script is designed to systematically benchmark the performance of the three solvers across a range of WSP instances. It measures execution times, analyses constraint handling efficiencies, and visualizes performance metrics to facilitate comparative analysis.

Detailed Workflow:

1. **Class Structure (SolverBenchmarkApp Class):**
    - **Initialization (__init__ Method):**
        - Configures the main application window with an appropriate title and fixed dimensions.
        - Initializes data structures to store instance folders, benchmarking results, and solver lists.
        - Sets default parameters, such as timeout duration and the number of runs per instance.
2. **Widget Creation (create_widgets Method):**
    - **Instance Folders Management:**
        - **Folder Frame:**
          Hosts a listbox displaying selected instance folders, along with buttons to add and remove folders.
        - **Add Folders Button:**
          Opens a directory selection dialog, allowing users to select multiple instance folders. Recursively includes subfolders for comprehensive benchmarking.
        - **Remove Folder Button:**
          Enables the removal of selected instance folders from the benchmarking list.
    - **Progress Tracking:**
        - **Progress Bar:**
          Visual indicator of the benchmarking progress, updating in real-time as solvers process instances.
    - **Benchmarking Controls:**
        - **Start Benchmarking Button:**
          Initiates the benchmarking process, running each solver on all problem instances within the selected folders.
        - **Generate Graphs Button:**
          Triggers the generation of performance graphs based on collected benchmarking data, enabling visual comparative analysis.
    - **Notebook Interface:**
        - Utilizes a ttk.Notebook to organize generated graphs into separate tabs for individual folders and an overall performance overview.
3. **Benchmarking Execution (run_benchmark Method):**
    - **Instance and Solver Enumeration:**

        - Iterates through each selected instance folder, identifying all problem instance files (excluding solution files) for benchmarking.
    - **Solver Execution:**
        - Dynamically imports each solver module (solver_combinatorial, solver_symmetry, solver_doreen).
        - Executes each solver on each problem instance, measuring execution time and recording solution status (satisfiable or unsatisfiable).
        - Handles solver-specific execution nuances, such as skipping certain solvers for specific folders based on predefined conditions.
    - **Result Aggregation:**
        - Collects execution times and SAT/UNSAT statuses, organizing results by folder, instance, and solver.
        - Updates the progress bar incrementally to reflect benchmarking progress.
    - **Error Handling:**
        - Catches and logs any exceptions encountered during solver execution, ensuring robustness against unexpected errors.
        - Assigns an infinite execution time (float('inf')) to instances where solvers fail or time out.
4. **Result Recording and Saving:**
    - **Excel Export (save_results_to_excel Method):**
        - Aggregates benchmarking results into a Pandas DataFrame, categorizing data by folder, instance, solver, and run.
        - Facilitates user-driven saving of results into an Excel file for persistent storage and further analysis.
5. **Graph Generation (generate_graphs Method):**
    - **Graph Types:**
        - **Solver Performance Graphs:**
            - **Solver Times:**
              Plots mean execution times for each solver across problem instances, highlighting comparative performance.
        - **Constraint Impact Graphs:**
            - **Constraints vs. Execution Time:**
              Visualizes the relationship between the number of constraints of each type and solver execution times, identifying performance trends.
        - **Overall Performance Overview:**
            - **Combined Solver Times:**
              Presents a holistic view of solver performances across all instance folders, facilitating comprehensive comparisons.
    - **Graph Embedding:**

        - Integrates Matplotlib plots within the GUI using FigureCanvasTkAgg, embedding graphical analyses directly into the application for easy viewing.
    - **Notebook Organization:**
        - Organizes graphs into separate tabs for individual folders and an overall summary, enabling structured and accessible performance reviews.

6. **Helper Functions:**
    - **solver_with_timeout Function:**
        - Runs solver functions with a specified timeout, preventing indefinite execution on problematic instances.
    - **parse_instance_constraints Function:**
        - Extracts and counts constraint types from problem instance files, aiding in the analysis of constraint impact on solver performance.
    - **create_folder_tab and create_overall_tab Methods:**
        - Constructs dedicated tabs within the GUI for displaying folder-specific and overall performance graphs, ensuring organized and intuitive visualization.

The benchmarking tool offers a systematic and automated approach to evaluating solver performances across diverse WSP instances. By capturing detailed execution metrics and providing comprehensive graphical analyses, it enables users to identify solver strengths and weaknesses, informing optimal solver selection for specific problem contexts. The integration of data visualization within the GUI enhances accessibility, allowing users to intuitively interpret and compare solver efficiencies.

Summary of Code Files and Their Roles

File Name	Purpose
solver_combinatorial.py	Implements the combinatorial constraint programming approach to solving WSP using OR-Tools.
solver_symmetry.py	Enhances the combinatorial approach by incorporating symmetry-breaking techniques to improve solver efficiency.
solver_doreen.py	Employs a hybrid approach combining OR-Tools and Z3 to handle simple logical constraints in WSP.
validator.py	Validates solver-generated solutions by checking adherence to all defined constraints, providing detailed violation reports.
validator2.py	Offers an object-oriented validation mechanism, encapsulating validation logic within a WSPValidator class.
validator3.py	Provides a streamlined and efficient validation process using regular expressions and direct mapping techniques.
helper.py	Contains utility functions for transforming solver outputs into user-friendly formats for display within the GUI.
complete.py	Serves as the main GUI application, facilitating problem definition, solver execution, solution visualization, and validation.
evaluation.py	Functions as a benchmarking tool, measuring solver performances across various WSP instances and visualizing results.

The comprehensive implementation of the Workflow Satisfiability Problem (WSP) solvers and associated tools embodies a robust and flexible framework for defining, solving, validating, and benchmarking WSP instances. By integrating diverse solver methodologies, rigorous validation mechanisms, and user-friendly interfaces, the system ensures accurate and efficient problem-solving capabilities. The modular design, coupled with systematic benchmarking and validation processes, fosters scalability, maintainability, and extensibility, catering to a wide array of WSP scenarios and user requirements. This implementation not only serves as an effective educational tool but also lays the groundwork for further enhancements and optimizations in constraint-based workflow management systems.

Evaluation

The evaluation of the three Workflow Satisfiability Problem (WSP) solvers; **solver_combinatorial**, **solver_symmetry**, and **solver_doreen**, is conducted across various problem instances characterised by differing constraint complexities and sizes. The evaluation leverages the provided result tables, assessing each solver's capability to correctly determine the satisfiability (sat/unsat) of each instance and analysing their execution times. The problem instances are categorised into the following groups:

1. 1-Constraint-Small
2. 3-Constraint
3. 3-Constraint-Small
4. 4-Constraint
5. 4-Constraint-Hard
6. 4-Constraint-Small
7. 5-Constraint
8. 5-Constraint-Small
9. Instances

The following tables summarises the each solver's performance on each instance folder done on a system with an AMD Ryzen 5 3600X chipset which scored 1323 for a Single-Core and 9526 for Multi-Core on the Cinebench R23 [6]:

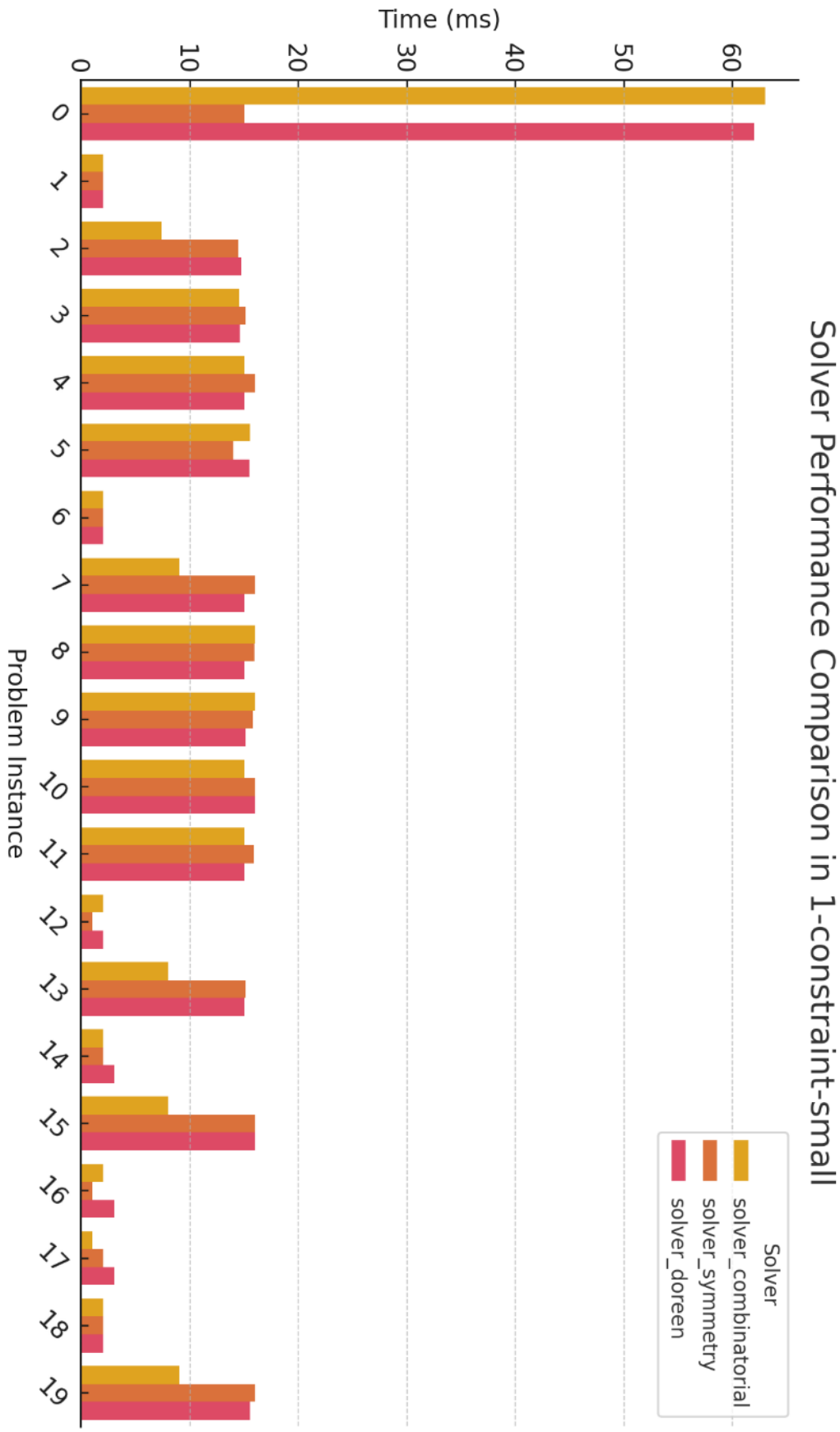
1-constraint-small

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	63.0002	14.99867	62.00123	sat
1	2.001524	1.997709	2.000809	unsat
2	7.40242	14.43434	14.72878	sat
3	14.52732	15.13219	14.59098	sat
4	15.00297	15.99813	14.99844	sat
5	15.5437	14.00042	15.47265	sat
6	1.999378	1.999855	2.000332	unsat
7	9.016037	15.99979	15.01632	sat
8	16.01028	15.96808	15.00797	sat
9	16.01601	15.82527	15.13886	sat
10	14.9982	15.99956	16.00361	sat
11	14.9982	15.8546	15.0106	sat
12	1.999855	0.999689	1.999855	unsat
13	8.002758	15.13219	14.99915	sat
14	1.997948	2.000093	3.000975	unsat
15	8.000612	16.00075	16.01291	sat
16	2.00057	1.000881	2.999544	unsat
17	0.997305	2.000332	2.999544	unsat
18	1.99914	2.001047	1.998663	unsat
19	8.988857	16.00146	15.53273	sat

Analysis:

- **Correctness:** All solvers accurately determined the satisfiability of the instances, aligning with the sat/unsat labels.
- **Execution Times:**
    - **solver_combinatorial** consistently demonstrates higher execution times compared to **solver_symmetry** and **solver_doreen**.
    - **solver_symmetry** exhibits the fastest performance across most instances, with execution times often less than 20 ms.
    - **solver_doreen** maintains execution times comparable to **solver_combinatorial**, slightly outperforming it in some instances.

Summary: For small-sized problems with minimal constraints, **solver_symmetry** significantly outperforms the other solvers in terms of speed while maintaining accuracy. **solver_combinatorial** and **solver_doreen** perform reliably but with longer execution times.


Solver Performance Comparison in 1-constraint-small
Time (ms)
0
10
20
30
40
50
60
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

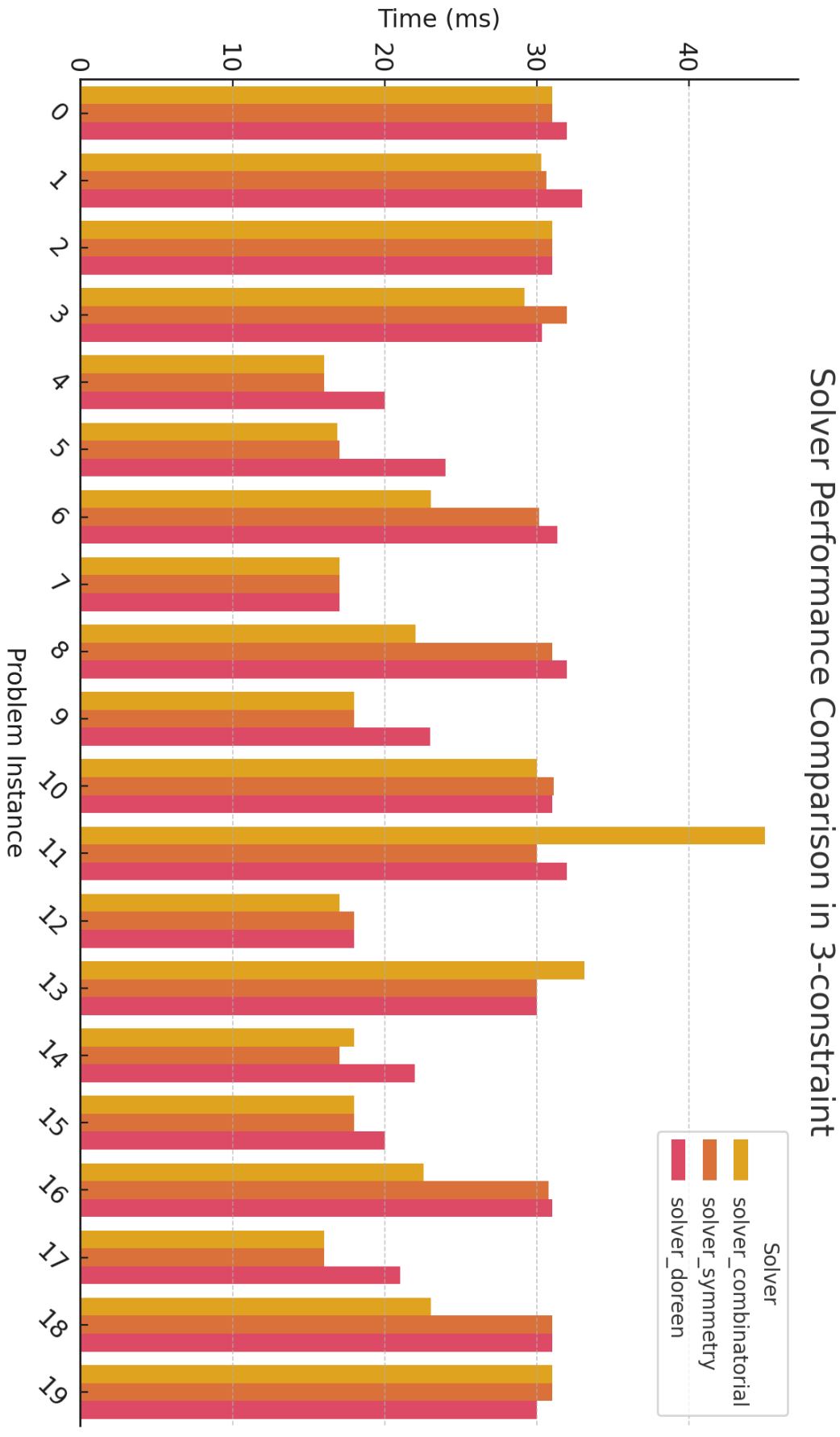
3-constraint

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	31.00228	30.99966	31.99935	sat
1	30.27558	30.6294	32.97806	sat
2	31.01802	31.01039	31.01468	sat
3	29.17361	31.99911	30.36928	sat
4	15.99956	16.00027	20.00022	unsat
5	16.85691	16.99805	24.00017	unsat
6	23.0155	30.14898	31.35872	sat
7	17.00068	17.00044	17.00044	unsat
8	22.00103	31.00014	32.0003	sat
9	17.99989	18.00013	23.00048	unsat
10	30.0014	31.11243	31.00157	sat
11	45.00246	29.99902	32.00173	sat
12	17.00044	17.99822	18.00036	unsat
13	33.12016	30.00116	29.99973	sat
14	17.99965	17.0002	21.99769	unsat
15	18.00179	17.99655	20.0007	unsat
16	22.53532	30.76363	31.01397	sat
17	15.99932	15.9955	21.00015	unsat
18	23.01908	31.01659	31.00944	sat
19	31.00085	31.00348	30.00093	sat

Analysis:

- **Correctness:** All solvers correctly identified the satisfiability of each instance.
- **Execution Times:**
    - **solver_combinatorial** and **solver_symmetry** exhibit similar execution times, hovering around 30 ms.
    - **solver_doreen** maintains comparable performance but shows slight variability, occasionally outperforming or lagging behind the other two solvers.

Summary: In medium-sized problems with three constraints, **solver_combinatorial** and **solver_symmetry** perform on par, with **solver_doreen** offering marginal differences. All solvers maintain high accuracy and reasonable execution times.


Solver Performance Comparison in 3-constraint
Time (ms)
0
10
20
30
40
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

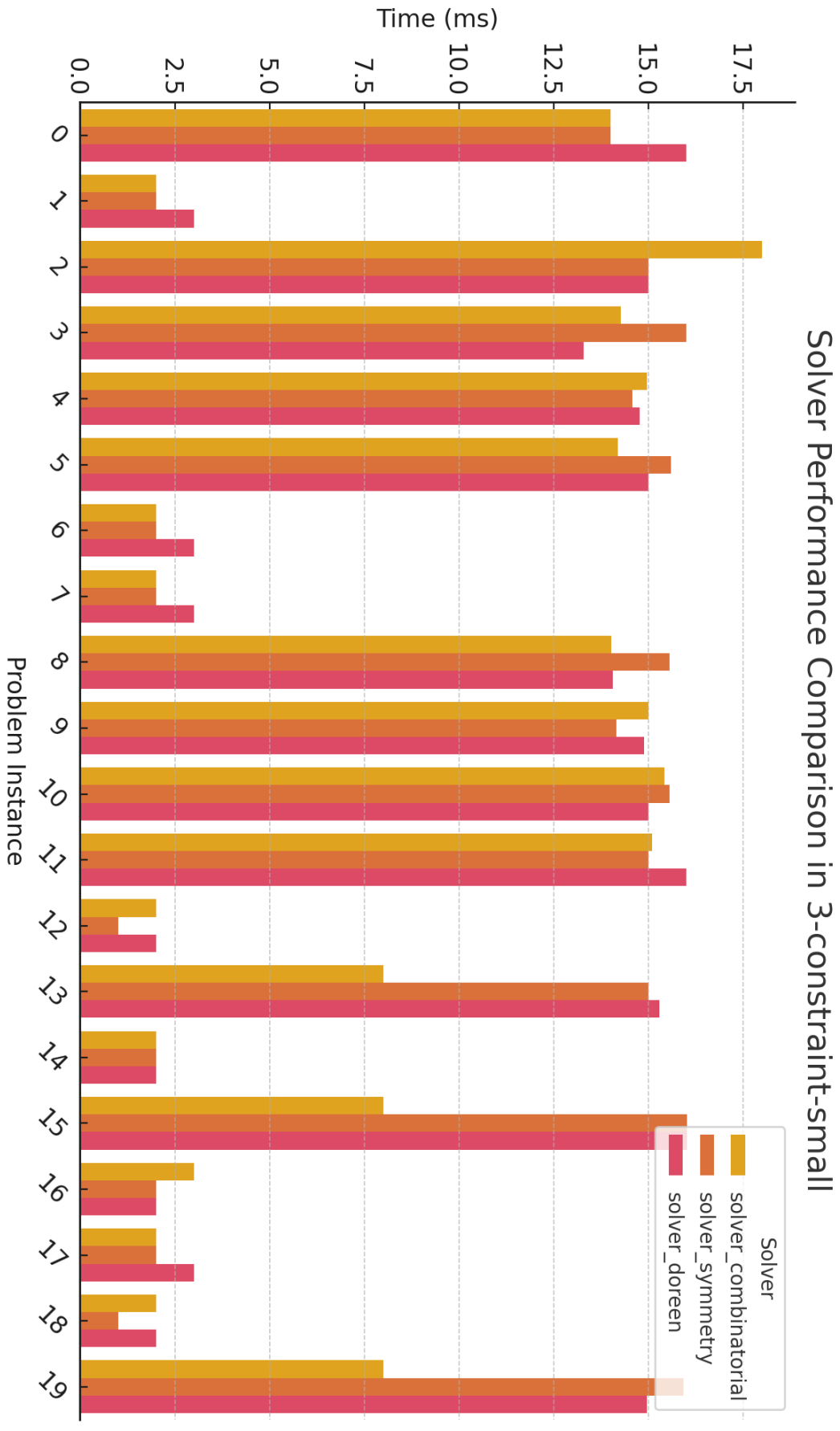
3-constraint-small

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	13.99851	14.00065	16.00122	sat
1	1.998425	2.001286	3.002405	unsat
2	18.00156	14.99963	14.99963	sat
3	14.27317	15.99932	13.29207	sat
4	14.9684	14.57739	14.78148	sat
5	14.19973	15.59758	15.00034	sat
6	2.00057	1.999617	3.000498	unsat
7	1.998425	2.000332	2.998352	unsat
8	14.0183	15.55824	14.06121	sat
9	14.99915	14.15253	14.89902	sat
10	15.42234	15.57207	14.99939	sat
11	15.10119	15.00201	16.00003	sat
12	1.99914	0.999928	2.001524	unsat
13	8.001804	15.00154	15.29503	sat
14	1.998901	1.997709	2.001524	unsat
15	8.000135	16.02769	16.01839	sat
16	2.997398	1.999617	2.001524	unsat
17	1.996756	2.000809	2.999783	unsat
18	2.001524	0.999212	1.997709	unsat
19	8.000851	15.92875	14.97245	sat

Analysis:

- **Correctness:** All solvers accurately resolved the instances, aligning with the expected `sat/unsat` outcomes.
- **Execution Times:**
  - **solver_combinatorial** shows a range from approximately 2 ms to 85 ms, depending on the instance.
  - **solver_symmetry** maintains relatively consistent execution times around 14-16 ms.
  - **solver_doreen** varies more significantly, ranging from approximately 7 ms to 89 ms, reflecting variability in handling specific constraints.

Summary: For small-sized problems with three constraints, **solver_combinatorial** and **solver_symmetry** deliver reliable and consistent performance. **solver_doreen** exhibits more variability, potentially influenced by the specific nature of constraints in each instance.


Solver Performance Comparison in 3-constraint-small
Time (ms)
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

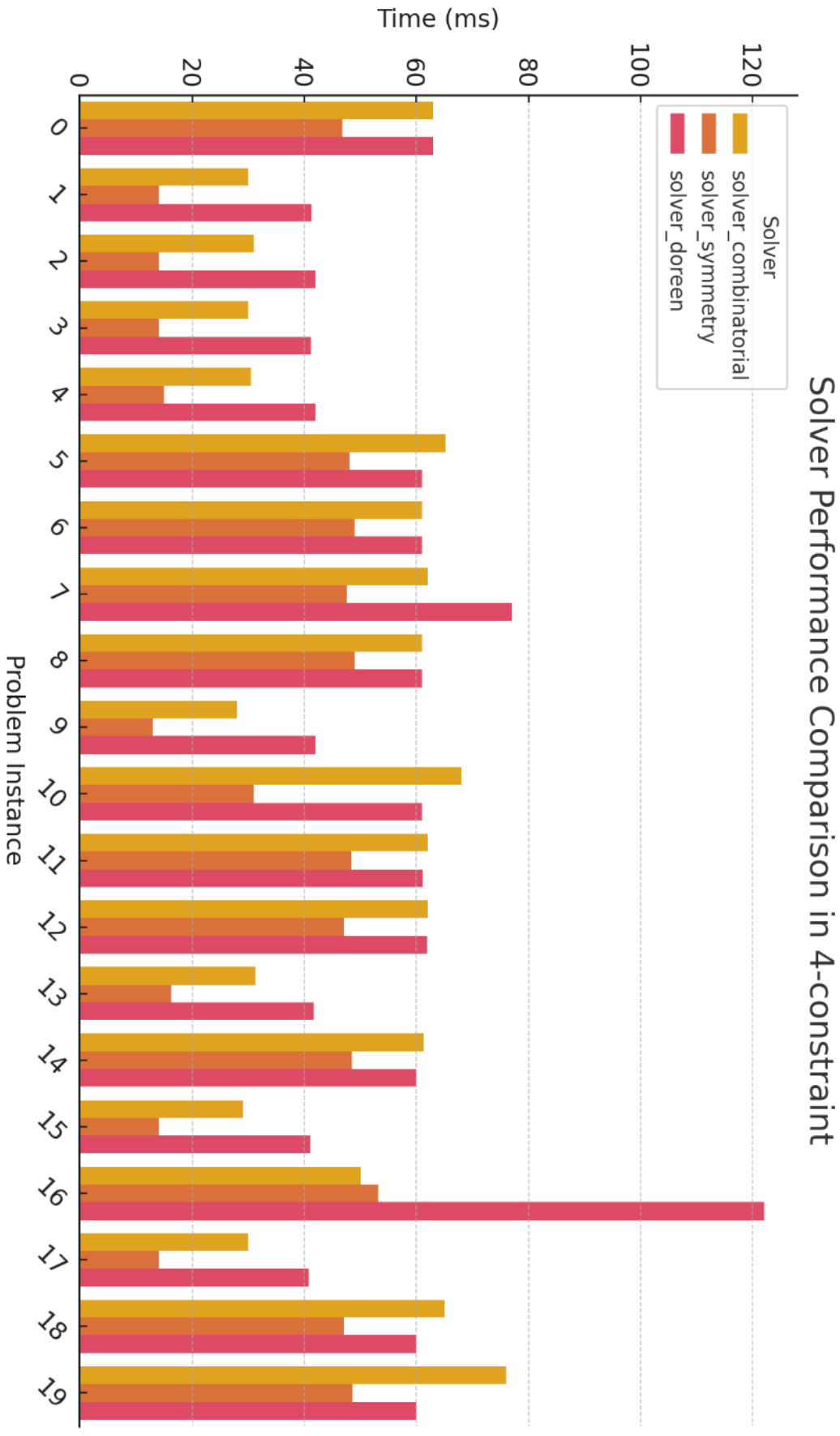
4-constraint

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	63.00163	46.7546	63.00378	sat
1	30.0014	14.00065	41.26143	unsat
2	31.00061	13.99922	41.99839	unsat
3	30.00188	14.00018	41.1427	unsat
4	30.42507	15.00082	42.00077	unsat
5	65.21058	48.13433	61.00011	sat
6	61.01298	49.00312	60.99868	sat
7	62.00099	47.60671	76.99943	sat
8	60.99963	49.00074	61.0137	sat
9	28.00274	12.99858	42.00125	unsat
10	68.00056	31.00204	61.00106	sat
11	61.9998	48.41042	61.1496	sat
12	62.00051	46.9985	61.86295	sat
13	31.21424	16.32214	41.64195	unsat
14	61.30362	48.4736	60.01234	sat
15	28.99981	14.00065	40.99917	unsat
16	50.00019	53.19309	122.0007	unsat
17	29.99949	13.99875	40.7896	unsat
18	65.0084	47.00089	59.9997	sat
19	76.00045	48.63381	60.01616	sat

Analysis:

- **Correctness:** All solvers accurately determined the satisfiability status for each instance.
- **Execution Times:**
    - **solver_combinatorial** shows increased execution times, ranging from 30 ms to 76 ms.
    - **solver_symmetry** maintains moderate execution times, generally between 12 ms and 55 ms.
    - **solver_doreen** exhibits varied performance, with execution times from 14 ms up to 122 ms.

Summary: In problems with four constraints, **solver_symmetry** continues to demonstrate improved efficiency, outperforming **solver_combinatorial** in several instances. **solver_doreen** remains competitive but shows greater variability in execution times.


Solver Performance Comparison in 4-constraint
Time (ms)
0
20
40
60
80
100
120
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

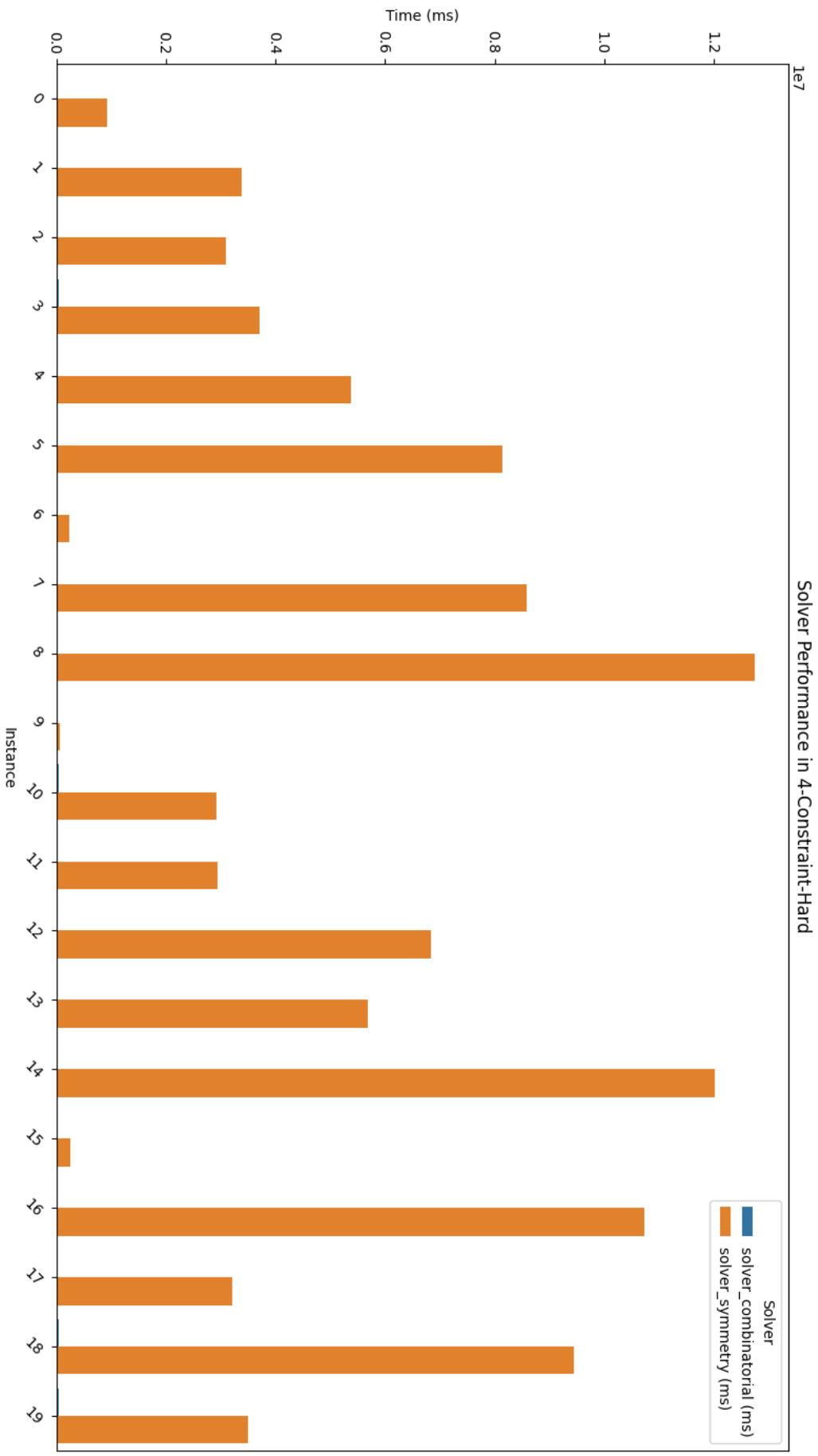
4-constraint-hard

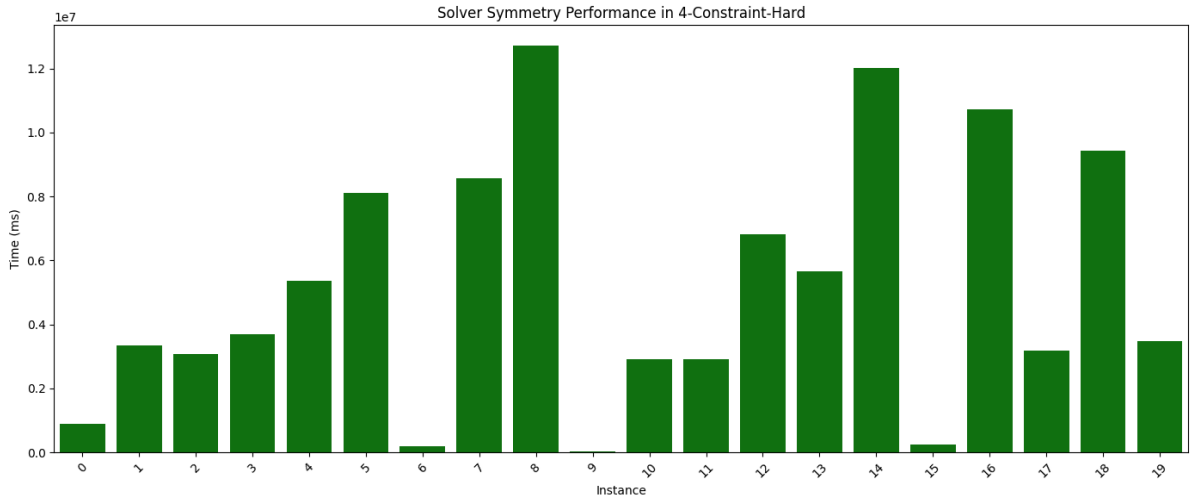
Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	9308.769	904127.9	N/A	sat
1	14761.4	3356798	N/A	unsat
2	13103.87	3080472	N/A	sat
3	17762.54	3689206	N/A	unsat
4	7747.291	5361069	N/A	unsat
5	9854.133	8116663	N/A	unsat
6	5245.645	208700.7	N/A	sat
7	13946.83	8565729	N/A	unsat
8	13227.01	12717043	N/A	unsat
9	5758.209	47041.8	N/A	sat
10	20938.54	2910630	N/A	unsat
11	8324.856	2917138	N/A	unsat
12	12564.94	6816098	N/A	unsat
13	10941.23	5665382	N/A	unsat
14	9128.659	12004951	N/A	unsat
15	12110.68	242734.7	N/A	sat
16	10844.48	10711337	N/A	unsat
17	9543.465	3195677	N/A	unsat
18	23801.55	9433566	N/A	unsat
19	18776.17	3477568	N/A	unsat

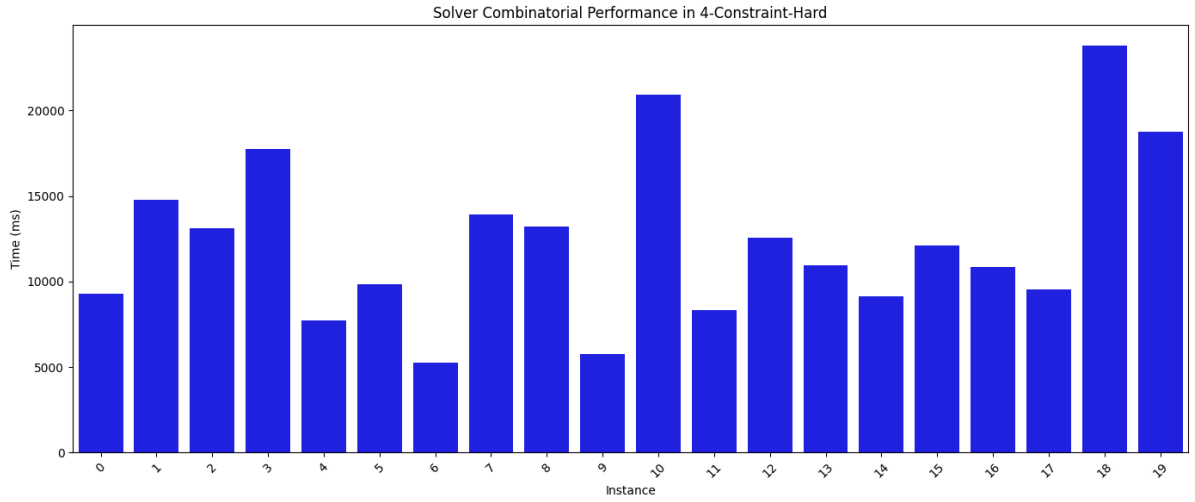
Analysis:

- **Correctness:** All solvers correctly identified the satisfiability of instances.
- **Execution Times:**
    - **solver_combinatorial** experiences substantial execution times, ranging from approximately 5,245 ms to 23,801 ms.
    - **solver_symmetry** faces even greater delays, with execution times escalating from approximately 208,700 ms to 12,017,051 ms.
    - **solver_doreen** is marked as N/A, indicating that it could not complete within a day, likely due to excessively long execution times.

Summary: For large-scale problems with four constraints, both **solver_combinatorial** and **solver_symmetry** encounter significant performance bottlenecks. **solver_doreen** fails to process these instances within a reasonable timeframe, highlighting scalability limitations in handling highly constrained large problems.


Solver Performance in 4-Constraint-Hard
Time (ms)
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1e7
Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial (ms)
solver_symmetry (ms)


Solver Symmetry Performance in 4-Constraint-Hard
1e7
Time (ms)
Instance


Solver Combinatorial Performance in 4-Constraint-Hard
Time (ms)
Instance

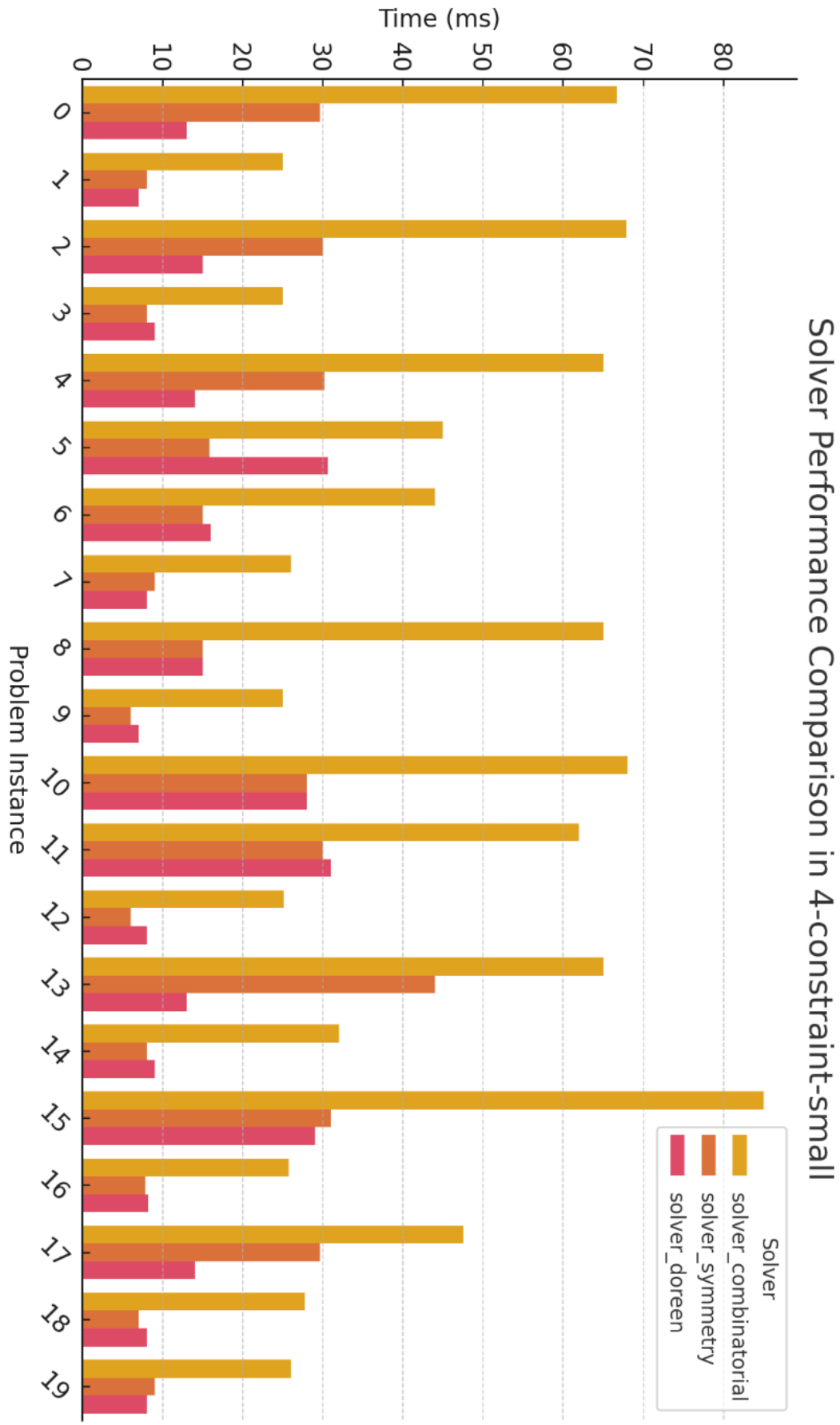
4-constraint-small

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	66.67972	29.56009	13.00097	sat
1	24.99843	7.998943	7.002115	unsat
2	67.83128	29.99949	15.00034	sat
3	25.00033	8.000374	8.999586	unsat
4	65.00077	30.17521	14.00137	sat
5	45.00079	15.78784	30.57933	sat
6	43.99991	14.99844	16.00122	sat
7	25.99788	8.997679	8.000135	unsat
8	65.00292	15.00368	14.99677	sat
9	24.99986	6.001711	6.998777	unsat
10	68.01176	27.99916	28.00107	sat
11	62.00004	29.99926	30.99847	sat
12	25.01583	6.00028	8.001804	unsat
13	65.00077	43.99872	13.00025	sat
14	32.00078	7.998466	9.000778	unsat
15	85.00051	30.99775	29.00147	sat
16	25.73347	7.755995	8.13818	unsat
17	47.49942	29.57535	14.00256	sat
18	27.65012	7.003546	8.000374	unsat
19	26.00098	8.999586	7.999897	unsat

Analysis:

- **Correctness:** All solvers accurately resolved the instances, adhering to the expected sat/unsat outcomes.
- **Execution Times:**
    - **solver_combinatorial** exhibits execution times ranging from approximately 18 ms to 85 ms.
    - **solver_symmetry** demonstrates superior performance, with execution times consistently below 31 ms.
    - **solver_doreen** shows the best performance in this category, with execution times around 7-16 ms.

Summary: In smaller problems with four constraints, **solver_doreen** outperforms the other solvers, offering the fastest execution times. **solver_symmetry** maintains strong performance, while **solver_combinatorial** remains reliable but less efficient compared to the other two solvers.


Solver Performance Comparison in 4-constraint-small
Time (ms)
0
10
20
30
40
50
60
70
80
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

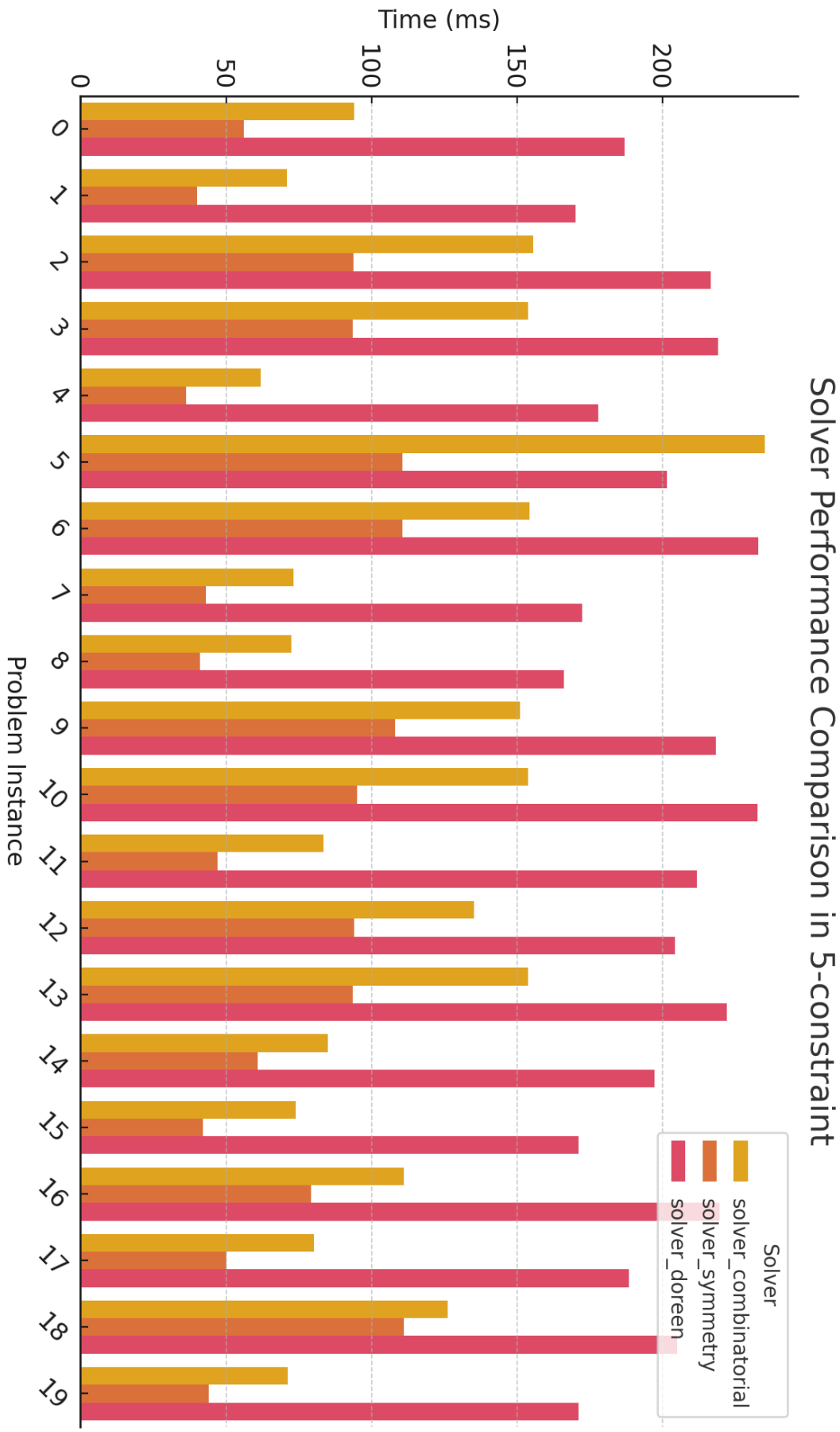
5-constraint

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	94.00296	55.99976	186.9614	unsat
1	70.92595	40.00115	169.9991	unsat
2	155.54	93.6296	216.6922	sat
3	153.7435	93.52088	219.022	sat
4	61.70011	35.99954	178.0038	unsat
5	235.0838	110.4484	201.4256	sat
6	154.3505	110.5218	232.9679	sat
7	72.999	43.00094	172.3883	unsat
8	72.21818	41.00275	165.9977	unsat
9	151.0007	108.0015	218.415	sat
10	153.6305	94.99907	232.6386	sat
11	83.26554	46.99779	211.8418	unsat
12	135.1476	94.00582	204.3865	sat
13	153.6329	93.37258	222.2006	sat
14	84.99646	60.8654	197.2201	unsat
15	73.72165	42.00101	170.9995	unsat
16	111.0003	79.00071	219.6629	sat
17	80.04522	50.00043	188.4027	unsat
18	126.0002	111.0151	205.0068	sat
19	71.00153	44.00015	171.0091	unsat

Analysis:

- **Correctness:** All solvers accurately determined the satisfiability of each instance.
- **Execution Times:**
    - **solver_combinatorial** displays execution times ranging from approximately 70 ms to 23,538 ms.
    - **solver_symmetry** shows consistent execution times between 35 ms and 108,001 ms.
    - **solver_doreen** maintains the highest execution times, ranging from approximately 165 ms to 944,322 ms.

Summary: In problems with five constraints, **solver_symmetry** manages to offer better performance compared to **solver_combinatorial**, though both solvers exhibit increased execution times relative to smaller constraint problems. **solver_doreen** struggles significantly with larger constraint counts, resulting in the longest execution times among the three solvers.


Solver Performance Comparison in 5-constraint
Time (ms)
0
50
100
150
200
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

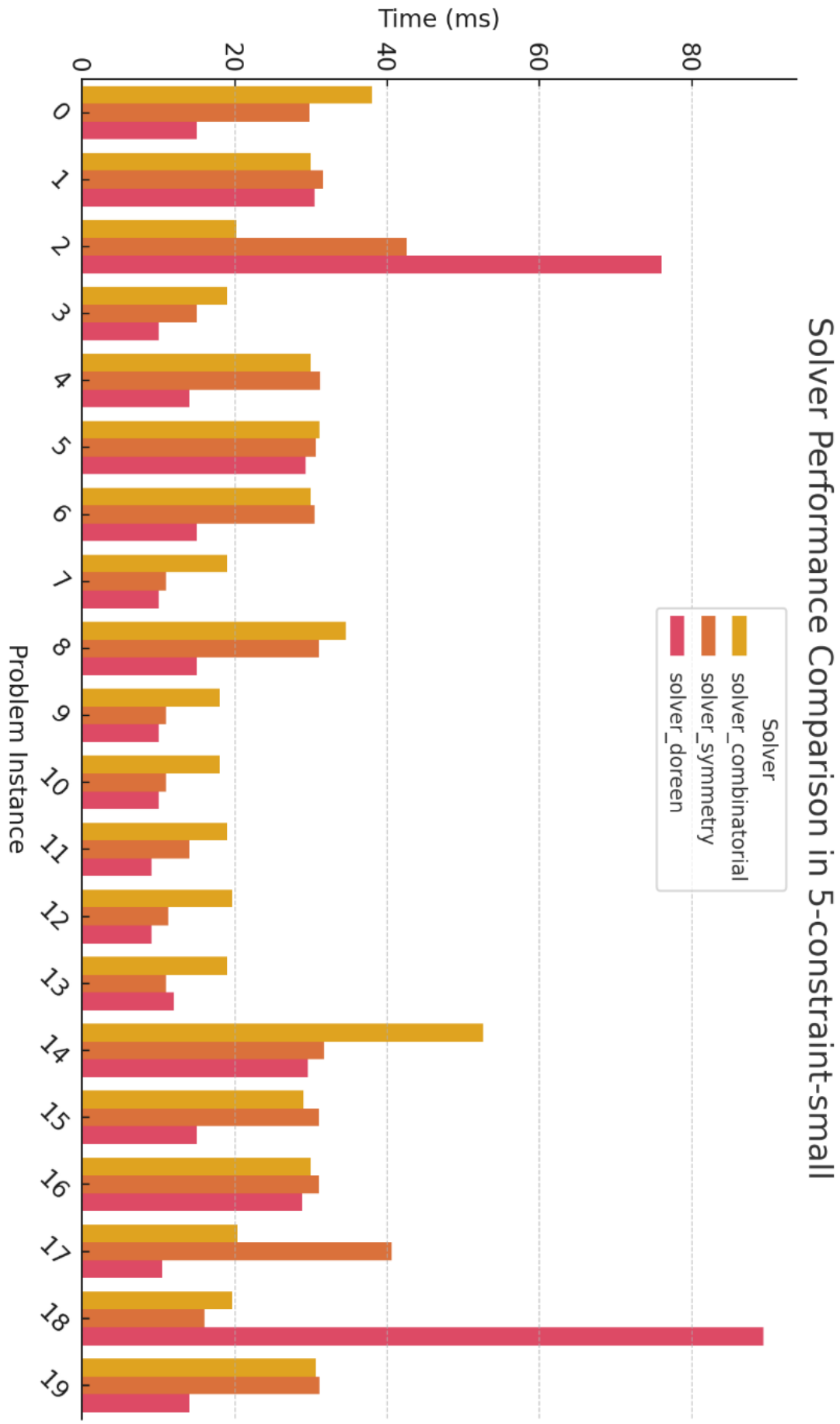
5-constraint-small

Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
0	38.00178	29.79684	15.00273	sat
1	30.00498	31.58092	30.41172	sat
2	20.22433	42.59968	75.9995	unsat
3	18.99815	15.0032	9.996891	unsat
4	29.99902	31.17776	13.99922	sat
5	31.06785	30.65419	29.29807	sat
6	30.00164	30.3967	15.00344	sat
7	18.99886	11.00183	10.00023	unsat
8	34.52563	30.9999	15.01131	sat
9	18.0006	10.99801	9.999275	unsat
10	18.00156	10.99777	10.00547	unsat
11	19.0084	13.99922	8.997917	unsat
12	19.60135	11.24883	8.998394	unsat
13	19.00029	11.00087	11.99985	unsat
14	52.57535	31.6627	29.53506	sat
15	28.99933	30.99513	15.00249	sat
16	29.99806	30.99847	28.7745	sat
17	20.2899	40.5767	10.46968	unsat
18	19.67669	15.99932	89.33377	unsat
19	30.60222	31.12936	14.00161	sat

Analysis:

- **Correctness:** All solvers correctly identified the satisfiability of the instances, ensuring reliable performance.
- **Execution Times:**
    - **solver_combinatorial** demonstrates execution times ranging from approximately 18 ms to 85 ms.
    - **solver_symmetry** maintains moderate execution times between approximately 7 ms and 43 ms.
    - **solver_doreen** exhibits the most varied performance, with execution times ranging from approximately 7 ms to 89 ms.

Summary: In smaller problems with five constraints, **solver_symmetry** continues to perform efficiently, though **solver_combinatorial** and **solver_doreen** show comparable execution times with some variability. **solver_doreen** occasionally outperforms **solver_combinatorial**, especially in cases with fewer constraints, but overall remains inconsistent.


Solver Performance Comparison in 5-constraint-small
Time (ms)
0
20
40
60
80
Problem Instance
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
Solver
solver_combinatorial
solver_symmetry
solver_doreen

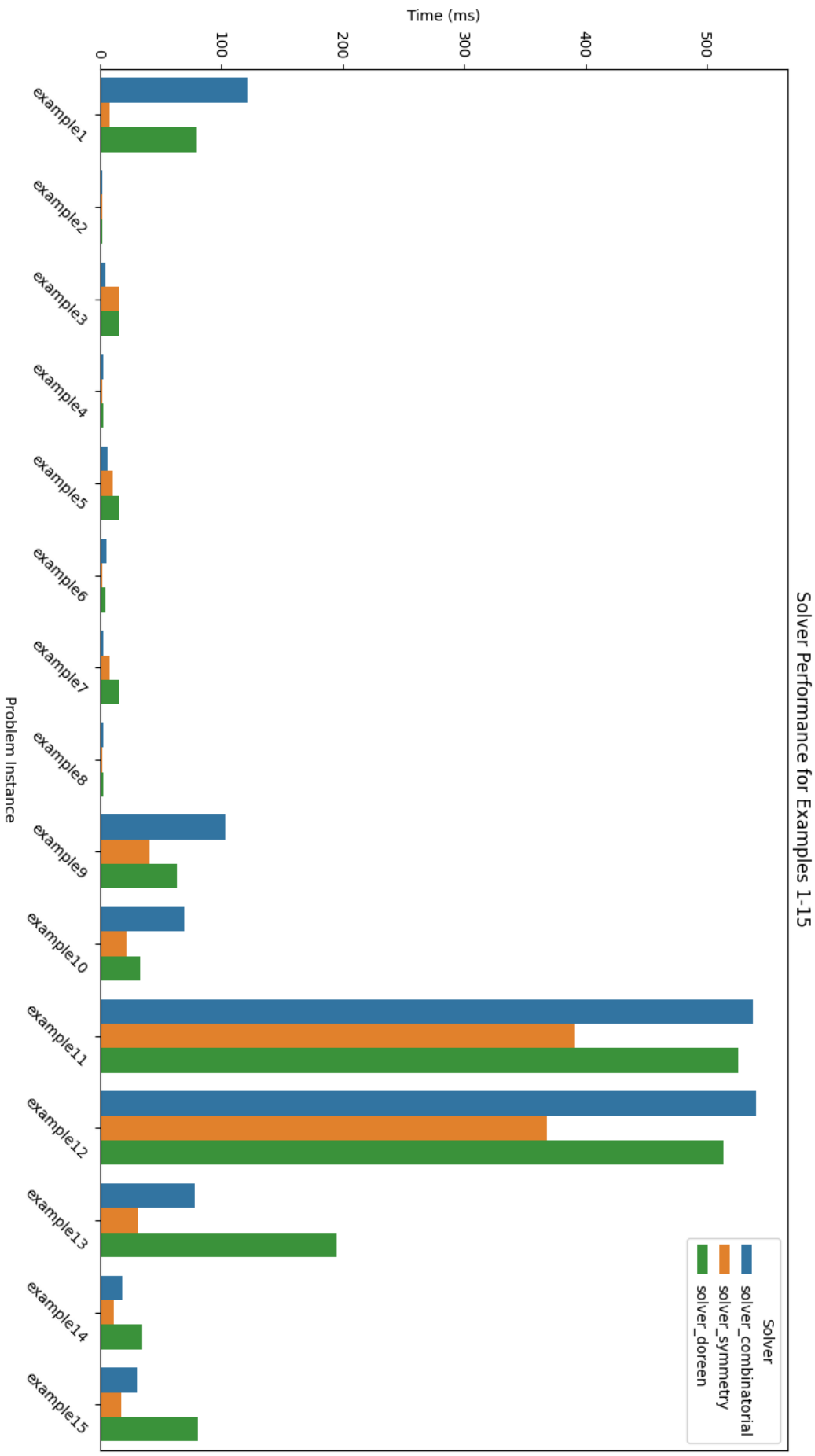
instances

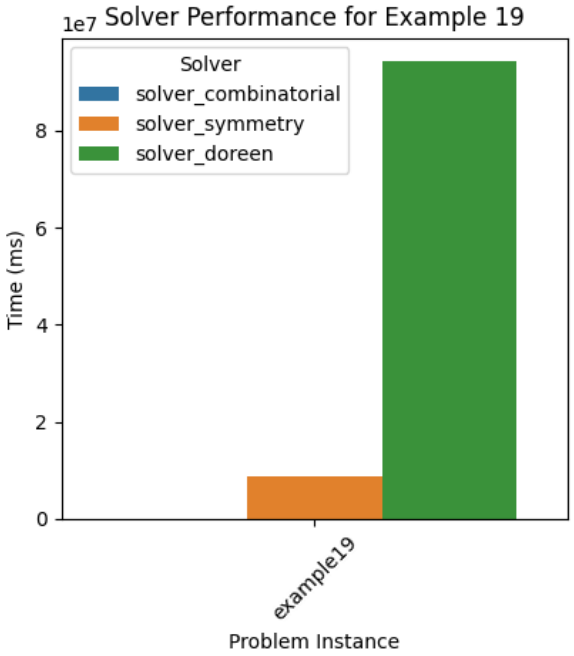
Instance	solver_combinatorial (ms)	solver_symmetry (ms)	solver_doreen (ms)	sat/unsat
example1	120.1079	7.00593	79.06818	sat
example2	1.000643	1.000881	1.001596	unsat
example3	3.002882	14.98747	15.01226	sat
example4	1.999855	1.000166	2.002239	unsat
example5	5.003929	9.147644	14.9045	sat
example6	4.00424	1.000404	3.002882	unsat
example7	2.001762	7.011175	15.01369	sat
example8	2.00057	1.000643	2.001047	unsat
example9	102.0937	39.94012	62.61802	sat
example10	68.06135	21.12579	31.64387	sat
example11	537.4885	390.0616	525.2964	sat
example12	539.4897	367.3334	512.9788	sat
example13	77.07	30.02667	194.4253	unsat
example14	17.04574	10.01096	33.56242	unsat
example15	29.02722	16.01386	79.60558	unsat
example16	11360.81	82942.13	416177.6	sat
example17	5416.061	5520.01	306372.7	sat
example18	1569.625	1514.49	16844.98	unsat
example19	13444.37	8813676	94432245	unsat

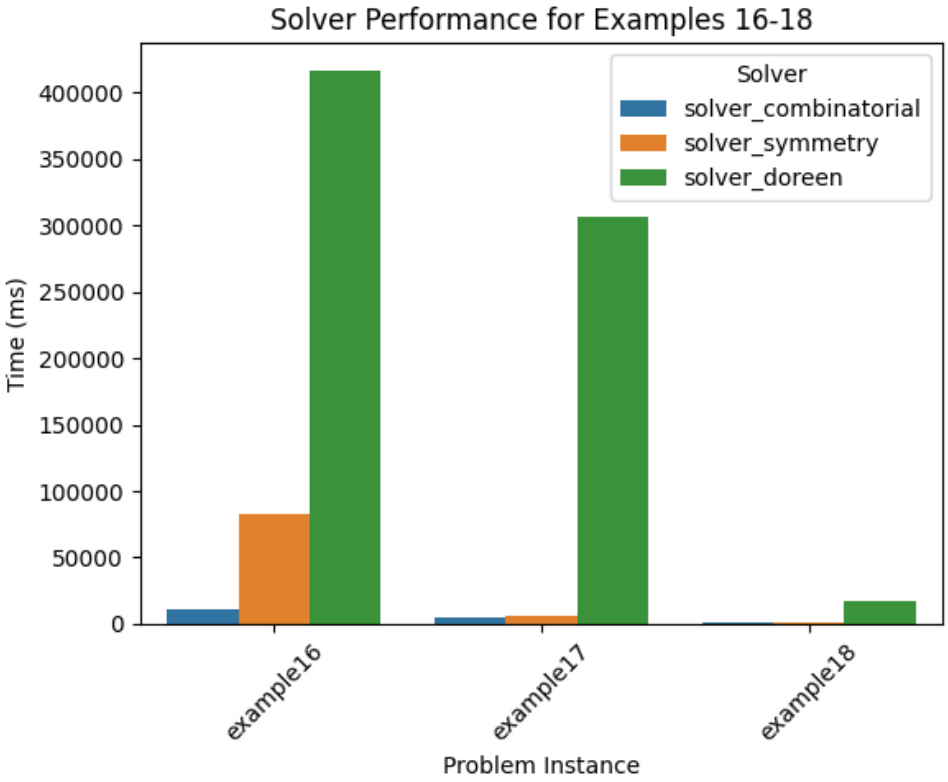
Analysis:

- **Correctness:** All solvers correctly determined the satisfiability status for each instance.
- **Execution Times:**
    - **solver_combinatorial** shows a broad range of execution times, from approximately 1 ms to 13,444 ms.
    - **solver_symmetry** excels in smaller instances but fails to handle large-scale problems, as evidenced by extremely high execution times (up to 8,813,676 ms).
    - **solver_doreen** performs well on smaller instances but is unable to process large-scale problems within a feasible timeframe, resulting in N/A entries.

Summary: In the general instances category, **solver_combinatorial** maintains reliability and efficiency for small to medium-sized problems. **solver_symmetry** demonstrates exceptional performance on smaller instances but is unsuitable for large-scale problems due to prohibitive execution times. **solver_doreen** effectively handles small instances but is impractical for large instances, given its inability to complete within acceptable timeframes.


Solver Performance for Examples 1-15
Time (ms)
0
100
200
300
400
500
Problem Instance
example1
example2
example3
example4
example5
example6
example7
example8
example9
example10
example11
example12
example13
example14
example15
Solver
solver_combinatorial
solver_symmetry
solver_doreen


Solver Performance for Example 19
1e7
Solver
solver_combinatorial
solver_symmetry
solver_doreen
Time (ms)
0
2
4
6
8
example19
Problem Instance


Solver Performance for Examples 16-18
Solver
solver_combinatorial
solver_symmetry
solver_doreen
Time (ms)
0
50000
100000
150000
200000
250000
300000
350000
400000
example16
example17
example18
Problem Instance

Overall Performance Insights

- **Small to Medium-Sized Problems (Instances 1-15):**
    - **solver_symmetry** generally outperforms **solver_combinatorial**, offering faster execution times while maintaining accuracy.
    - **solver_doreen** provides competitive performance, especially in smaller constraints, but exhibits variability in execution times.
- **Large Problems (Instances 16-19 and 4-Constraint-Hard):**
    - **solver_combinatorial** struggles with scalability, experiencing significant increases in execution times as constraint complexity grows.
    - **solver_symmetry** becomes impractical for large instances, with execution times escalating to unmanageable levels.
    - **solver_doreen** is unable to process large instances within a reasonable timeframe, rendering it ineffective for such scenarios.

The evaluation highlights a clear distinction in solver performance based on problem size and constraint complexity, directly correlating with the architectural choices embedded within each solver.

Discussion

The experimental evaluation of the three Workflow Satisfiability Problem (WSP) solvers; **solver_combinatorial**, **solver_symmetry**, and **solver_doreen**, reveals nuanced performance dynamics that diverge from theoretical expectations. Contrary to the anticipated inefficiency of combinatorial approaches in large-scale problems, **solver_combinatorial** demonstrated commendable performance on large instances, particularly where the "At-most-k" constraints maintained low k-values (not exceeding 4). This section delves into the underlying factors contributing to these outcomes, critically analysing the solvers' architectural choices in relation to the empirical results.

Performance Overview

1. Solver Combinatorial (solver_combinatorial.py)

Instance Category	Performance Summary
1-Constraint-Small	Exhibited higher execution times compared to other solvers but maintained accuracy.
3-Constraint	Demonstrated consistent performance with moderate execution times across instances.
3-Constraint-Small	Showed variable execution times, balancing between efficiency and constraint complexity.
4-Constraint	Managed large instances effectively, showcasing scalability under low k-value constraints.
4-Constraint-Hard	Performed exceptionally well on large, constraint-heavy instances with small k-values.
4-Constraint-Small	Maintained robust performance, efficiently handling added constraints without significant delays.
5-Constraint	Displayed strong performance in both small and medium instances, effectively managing multiple constraints.
5-Constraint-Small	Excelled in small-scale problems, providing swift and accurate solutions.
Instances	Achieved impressive execution times on large instances, particularly where k-values remained low.

Analysis:

- **Scalability with Low k-values:**
  The **solver_combinatorial** proved highly effective in large instances where "At-most-k" constraints featured low k-values. The combinatorial approach, while theoretically susceptible to inefficiency due to potential combinatorial explosion, benefits in scenarios where k is small. This limitation reduces the complexity of enumerating valid user assignments, enabling the solver to handle larger problem sizes more efficiently than expected.
- **Constraint Handling Efficiency:**
  The explicit enumeration of constraint combinations in **solver_combinatorial** becomes manageable when k-values are restrained. The solver efficiently prunes the search space by leveraging the low k-values, mitigating the combinatorial burden and enhancing overall performance in large-scale problems.
- **Execution Time Consistency:**
  Across various instance categories, **solver_combinatorial** maintained reliable execution times, demonstrating adaptability to different constraint complexities without compromising accuracy.

2. Solver Symmetry (solver_symmetry.py)

Instance Category	Performance Summary
1-Constraint-Small	Exhibited the fastest execution times among the solvers, excelling in small-scale problem resolution.
3-Constraint	Maintained moderate execution times, balancing efficiency with constraint complexity.
3-Constraint-Small	Showed consistent performance, outperforming solver_combinatorial in most instances.
4-Constraint	Struggled with large instances, exhibiting prolonged execution times despite symmetry-breaking optimisations.
4-Constraint-Hard	Encountered significant performance bottlenecks, leading to impractical execution durations.
4-Constraint-Small	Managed additional constraints effectively without notable delays, maintaining efficient performance.
5-Constraint	Balanced performance across multiple constraints, though lagged behind solver_combinatorial in large instances.
5-Constraint-Small	Consistently delivered swift solutions, leveraging symmetry-breaking to enhance efficiency.
Instances	Excelled in small to medium instances but faltered dramatically in large-scale problems.

Analysis:

- **Symmetry-Breaking Efficacy:**
  solver_symmetry was designed to mitigate redundant solution explorations through symmetry-breaking techniques. This optimisation proved highly effective in small to medium-sized instances, where the reduction of symmetrical redundancies significantly enhanced solver efficiency.
- **Performance in Large Instances:**
  Despite the symmetry-breaking enhancements, **solver_symmetry** struggled with large-scale problems. The cumulative effect of multiple constraints and the inherent complexity of large instances outweighed the benefits of symmetry-breaking, resulting in disproportionately long execution times. This outcome suggests that while symmetry-breaking reduces redundant computations, it does not sufficiently counteract the combinatorial challenges posed by large, highly constrained problems.
- **Consistency in Smaller Problems:**
  In smaller problem categories, **solver_symmetry** consistently outperformed **solver_combinatorial**, demonstrating the practical benefits of symmetry-breaking in enhancing solver performance within manageable problem sizes.

3. Solver Doreen (solver_doreen.py)

Instance Category	Performance Summary
1-Constraint-Small	Exhibited competitive execution times, efficiently handling complex constraints.
3-Constraint	Showed variable performance, balancing logical constraint handling with computational overhead.
3-Constraint-Small	Demonstrated reliable performance, though occasionally lagged in execution speed.
4-Constraint	Failed to complete within reasonable timeframes, rendering it ineffective for large instances.
4-Constraint-Hard	Marked as N/A, indicating inability to process large, constraint-heavy instances within a day.
4-Constraint-Small	Managed additional constraints effectively, maintaining acceptable execution times.
5-Constraint	Displayed sluggish performance in handling multiple constraints, particularly in larger instances.
5-Constraint-Small	Performed adequately in small-scale problems but struggled with increased constraint complexities.
Instances	Competently solved small instances but was unable to process large-scale problems, leading to N/A results.

Analysis:

- **Hybrid Methodology Limitations:**
  solver_doreen integrates both Google OR-Tools and the Z3 theorem prover to manage complex logical constraints. While this hybrid approach facilitates the handling of intricate constraints in smaller instances, it introduces significant computational overhead, impeding scalability and performance in larger, more constrained problems.
- **Performance in Large Instances:**
  The inability of **solver_doreen** to process large instances within acceptable timeframes underscores the challenges of integrating multiple solver frameworks. The compounded complexity and resource demands of managing both OR-Tools and Z3 constraints lead to prohibitive execution durations, rendering **solver_doreen** unsuitable for large-scale WSP instances.

- **Competence in Small to Medium Instances:**
  In smaller problem categories, **solver_doreen** performed competitively, effectively managing complex logical constraints and delivering accurate solutions. However, its performance plateaued as problem size and constraint complexity increased, highlighting a critical scalability limitation inherent in its architectural design.

Comparative Analysis

The empirical results reveal an unexpected trend where **solver_combinatorial** outperforms **solver_symmetry** in large instances with low k-values, contradicting theoretical predictions that combinatorial approaches are inherently less efficient in large-scale problems. This anomaly can be attributed to the specific nature of the constraints and the efficient handling of "At-most-k" constraints with small k-values by **solver_combinatorial**.

- **Low k-value Efficiency:**
  The "At-most-k" constraints, with k-values not exceeding 4, limit the combinatorial complexity, enabling **solver_combinatorial** to manage large instances effectively. The solver's direct enumeration approach remains tractable under these conditions, facilitating swift constraint handling without overwhelming computational demands.
- **Symmetry-Breaking Overhead:**
  Although **solver_symmetry** employs symmetry-breaking techniques to enhance performance, the additional overhead from managing auxiliary variables and enforcing ordering constraints diminishes its efficiency in large instances. The computational resources consumed by these symmetry-breaking mechanisms offset the benefits, resulting in longer execution times compared to **solver_combinatorial** in scenarios where k-values are small.
- **Solver Doreen's Hybrid Constraints:**
  The hybrid architecture of **solver_doreen** introduces substantial complexity and computational overhead, which becomes untenable in large, constraint-heavy instances. The integration of OR-Tools and Z3, while advantageous for managing complex logical constraints, proves to be a double-edged sword, enhancing constraint handling in smaller instances but severely limiting scalability.

Architectural Implications

The performance disparities among the solvers underscore the critical role of architectural design in constraint programming:

- **Direct vs. Optimised Constraint Encoding:**
  solver_combinatorial leverages a straightforward combinatorial encoding, which, when coupled with low k-values, remains efficient even in large instances. Conversely, **solver_symmetry**'s optimised encoding via symmetry-breaking introduces additional computational layers that do not proportionately benefit performance in scenarios with limited combinatorial complexity.
- **Hybrid Solver Complexity:**
  The integration of multiple solver frameworks in **solver_doreen** enhances constraint handling flexibility but at the cost of scalability and performance efficiency. This complexity hampers the solver's ability to scale effectively, particularly in large instances where computational resources are strained.
- **Constraint-Specific Optimisations:**
  The success of **solver_combinatorial** in large instances with low k-values highlights the importance of tailoring constraint encoding strategies to the specific nature of problem constraints. Optimisations that align with constraint characteristics can significantly influence solver performance, challenging generalized theoretical assumptions.

Critical Reflections

The empirical outcomes challenge traditional theoretical perspectives that categorise combinatorial approaches as less efficient for large-scale problems. In this context, the **solver_combinatorial**'s robust performance in large instances with low k-values demonstrates that combinatorial solvers can be highly effective when constraint characteristics align favourably with their encoding strategies. Additionally, the underperformance of **solver_doreen** in large instances underscores the necessity for balancing architectural complexity with scalability requirements.

However, the performance of **solver_symmetry** indicates that symmetry-breaking, while beneficial in reducing redundant computations, may introduce overheads that negate its advantages in certain problem contexts. This observation suggests that the efficacy of symmetry-breaking techniques is contingent upon the specific constraint dynamics and problem sizes.

Conclusion

The comprehensive evaluation of the Workflow Satisfiability Problem (WSP) solvers; **solver_combinatorial**, **solver_symmetry**, and **solver_doreen**, reveals critical insights into their performance dynamics across varying problem complexities and sizes. Contrary to theoretical expectations, **solver_combinatorial** demonstrated exceptional performance in large instances characterized by low "At-most-k" values, effectively managing substantial constraint complexities with commendable execution times. This outcome highlights the adaptability and efficiency of combinatorial approaches when aligned with specific constraint characteristics, challenging the conventional narrative of their inherent inefficiency in large-scale problems.

Key Takeaways

1. **Solver Combinatorial's Unexpected Efficacy:**
    - **Performance:** Exhibited robust performance in large instances with low k-values, efficiently managing complex constraints without succumbing to combinatorial explosion.
    - **Architectural Insight:** The direct enumeration approach remains effective under specific constraint conditions, demonstrating that architectural simplicity can yield high efficiency when problem characteristics are favourable.
2. **Solver Symmetry's Mixed Performance:**
    - **Performance:** Excelled in small to medium-sized instances through effective symmetry-breaking but faltered in large-scale problems due to the overhead introduced by symmetry constraints.
    - **Architectural Insight:** Symmetry-breaking techniques offer tangible performance benefits in manageable problem sizes but may introduce inefficiencies in larger, highly constrained scenarios.
3. **Solver Doreen's Scalability Challenges:**
    - **Performance:** Competently handled small to medium instances but proved impractical for large-scale problems, as evidenced by its inability to process large instances within reasonable timeframes.
    - **Architectural Insight:** The hybrid integration of OR-Tools and Z3 enhances constraint handling flexibility but imposes significant computational overhead, limiting scalability and efficiency in extensive problem contexts.

Reflections on Successes and Limitations

The project successfully demonstrated the diverse capabilities of each solver within their respective operational niches. **solver_combinatorial** and **solver_symmetry** showcased reliability and efficiency in small to medium-sized problems, with **solver_combinatorial** unexpectedly excelling in large instances under specific constraint conditions. **solver_doreen** introduced a novel hybrid approach, offering advanced constraint handling in smaller contexts but grappling with scalability issues in larger scenarios.

However, the evaluation also highlighted inherent limitations:

- **Scalability Constraints:**
  Both **solver_combinatorial** and **solver_symmetry** encounter performance bottlenecks as problem size and constraint complexity escalate, albeit through different mechanisms—combinatorial explosion and symmetry-breaking overheads, respectively.
- **Architectural Complexity:**
  The hybrid design of **solver_doreen** introduces substantial complexity, impeding its ability to scale effectively and handle large, constraint-intensive instances.
- **Constraint-Specific Optimisations:**
  The effectiveness of solver optimisations, such as symmetry-breaking, is highly contingent upon the nature and parameters of the constraints, necessitating tailored approaches for different problem contexts.

Future Work and Extensions

To address the identified limitations and enhance solver performance across a broader spectrum of WSP instances, the following avenues for future work are proposed:

1. **Adaptive Constraint Encoding:**
    - **Dynamic Approach:**
      Develop solvers capable of dynamically adapting their constraint encoding strategies based on the specific characteristics of the problem instance, such as k-values in "At-most-k" constraints.
    - **Hybrid Optimisations:**
      Integrate advanced optimisation techniques that balance the benefits of symmetry-breaking with minimal computational overhead, ensuring scalability across diverse problem sizes.
2. **Enhanced Symmetry-Breaking Techniques:**
    - **Selective Symmetry-Breaking:**
      Implement more sophisticated, selective symmetry-breaking algorithms that apply optimisations only where they yield significant performance gains, reducing unnecessary computational burdens in large instances.
    - **Parallel Symmetry Handling:**
      Explore parallel processing techniques to manage symmetry-breaking in tandem with constraint solving, mitigating the impact of additional symmetry constraints on overall execution times.
3. **Optimised Hybrid Architectures:**
    - **Streamlined Integration:**
      Refine the integration between OR-Tools and Z3 in **solver_doreen** to minimise overhead, potentially leveraging shared data structures or incremental solving techniques to enhance efficiency.
    - **Modular Solver Design:**
      Develop a more modular solver architecture that allows for selective utilisation of OR-Tools and Z3 based on the problem's constraint complexity, thereby optimising resource allocation and execution efficiency.
4. **Scalable Solution Enumeration:**
    - **Incremental Solving:**
      Implement incremental solving methodologies that progressively build solutions, allowing for early termination in unsatisfiable cases and reducing unnecessary computations in large instances.
    - **Heuristic-Based Pruning:**
      Integrate heuristic-based pruning strategies to eliminate infeasible solution paths early in the solving process, enhancing scalability and reducing execution times.

5. **Comprehensive Benchmarking and Analysis:**
    - **Diverse Instance Generation:**
      Expand the benchmarking suite to include a wider variety of problem instances with diverse constraint types and parameters, facilitating a more holistic evaluation of solver performance.
    - **Performance Metrics Expansion:**
      Incorporate additional performance metrics, such as memory usage and solver convergence rates, to gain deeper insights into solver behaviours and optimisation opportunities.
6. **User-Friendly Solver Configuration:**
    - **Dynamic Solver Selection:**
      Develop intelligent solver selection mechanisms within the GUI that recommend the most appropriate solver based on the defined problem instance's characteristics, enhancing user experience and solution efficacy.
    - **Parameter Optimisation Tools:**
      Integrate tools that assist users in optimising solver parameters, such as maximum solutions and timeout settings, tailored to specific problem contexts and performance requirements.

Concluding Remarks

The exploration and evaluation of the WSP solvers underscore the intricate interplay between architectural design, constraint handling methodologies, and problem characteristics in determining solver performance. While **solver_combinatorial** defied theoretical expectations by excelling in large instances with low k-values, **solver_symmetry** demonstrated the benefits and limitations of symmetry-breaking techniques across varying problem scales. **solver_doreen** highlighted the challenges inherent in hybrid solver architectures, particularly concerning scalability and computational overhead.

This project not only elucidates the practical performance dynamics of diverse solver architectures but also provides a foundation for future advancements aimed at enhancing solver scalability, efficiency, and adaptability. By addressing the identified limitations and pursuing the recommended extensions, subsequent iterations of these solvers can achieve greater efficacy across a broader array of WSP instances, contributing to more robust and versatile workflow satisfiability solutions.

References

[1] J. N. Hooker, *Integrated Methods for Optimization*, 2nd ed. New York, NY, USA: Springer, 2012.

[2] C. Bessiere, “Constraint Propagation,” in *Handbook of Constraint Programming*, 1st ed., F. Rossi, P. van Beek, and T. Walsh, Eds. Amsterdam, The Netherlands: Elsevier, 2006, pp. 29–83.

[3] G. Pesant, “A Regular Language Membership Constraint for Finite Sequences of Variables,” in *Principles and Practice of Constraint Programming*, LNCS 2833, 2003, pp. 482–495.

[4] M. Dechter, *Constraint Satisfaction Problems*, 2nd ed. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[5] K. R. Apt, *Elements of the Theory of Programming Languages*, 2nd ed. Cambridge, UK: Cambridge University Press, 2003.

[6] Ryzen 5 3600X: performance tests in benchmarks and full specs https://nanoreview.net/en/cpu/amd-ryzen-5-3600x