Certainly! Below is a detailed analysis of each of the four solvers you've provided:

1. `solver\_mine\_combinatorial.py`

2. `solver\_og\_symmetryk.py`

3. `solver\_symmetryk\_simplified\_uservars.py`

4. `solver\_doreen.py`

For each solver, I'll cover:

- Formulation: How the solver models the problem and implements the constraints.

- Advantages and Disadvantages: Strengths and potential weaknesses.

- Best Use Cases: Scenarios where the solver performs well.

- Problem Formulation: Mathematical representation of the constraints and variables.

---

1. `solver\_mine\_combinatorial.py`

Formulation

Variables:

- Assignment Variables: Creates an integer variable `assignments[i]` for each step `i`, where `assignments[i] ∈ {1, 2, ..., users\_count}`.

Constraints:

1. Authorisations:

- For each user `u` with specified authorised steps, the solver adds constraints to ensure that `assignments[step] != u` for all steps not in `allowed\_steps[u]`.

2. Separation-of-Duty (SoD):

- For each SoD constraint between steps `s1` and `s2`, the solver adds `assignments[s1] != assignments[s2]`.

3. Binding-of-Duty (BoD):

- For each BoD constraint between steps `s1` and `s2`, the solver adds `assignments[s1] == assignments[s2]`.

4. At-Most-k:

- Optimized Combinatorial Approach:

- If `k < len(steps\_T)` (number of steps in the group), the solver generates all combinations of `k+1` steps.

- For each combination, it ensures that at least one pair of steps in the combination is assigned to the same user.

- This is done by introducing boolean variables `same\_user` for each pair in the combination and adding constraints that at least one `same\_user` is `True`.

- If `k >= len(steps\_T)`, the constraint is ignored because it is always satisfied.

5. One-Team:

- Parses the steps and the list of possible teams (each team is a group of users).

- For each possible combination of users in the teams and steps, it generates allowed assignments.

- Uses `AddAllowedAssignments` to restrict the assignments of the steps to only those combinations where the steps are assigned to users within the same team.

6. Precedence Constraints:

- Adds constraints `assignments[s1] < assignments[s2]` to enforce that the user assigned to `s1` has a lower index than the user assigned to `s2`.

Advantages and Disadvantages

Advantages:

- Optimized At-Most-k Constraint:

- The combinatorial approach can be efficient for small values of `k` and small groups of steps.

- It avoids introducing additional variables for user assignments.

- Direct Modeling of Constraints:

- Constraints are added directly without extra layers of abstraction, which can be straightforward for the solver.

Disadvantages:

- Scalability Issues:

- The combinatorial approach for the At-Most-k constraint can lead to a combinatorial explosion as the number of steps or `k` increases.

- Generating all combinations can become computationally intensive for larger instances.

- Limited Symmetry Breaking:

- Does not employ symmetry-breaking techniques extensively, which can affect solver performance due to redundant search spaces.

Best Use Cases

- Small to Medium-Sized Problems:

- Instances where the number of steps and users is moderate.

- Situations where the At-Most-k groups are small and `k` is significantly less than the number of steps.

- Problems with Few At-Most-k Constraints:

- When the number of At-Most-k constraints is low, the combinatorial overhead is manageable.

Problem Formulation

Let:

- \( S = \{1, 2, ..., n\} \) be the set of steps.

- \( U = \{1, 2, ..., m\} \) be the set of users.

- \( assignments: S \rightarrow U \) is the assignment function.

Variables:

- \( assignments\_i \in U \) for each step \( i \in S \).

Constraints:

1. Authorisations:

\[

\forall u \in U, \forall i \notin allowed\\_steps[u], assignments\_i \neq u

\]

2. Separation-of-Duty (SoD):

\[

\forall (s1, s2) \in SoD, assignments\_{s1} \neq assignments\_{s2}

\]

3. Binding-of-Duty (BoD):

\[

\forall (s1, s2) \in BoD, assignments\_{s1} = assignments\_{s2}

\]

4. At-Most-k (Optimized for \( k < |steps\\_T| \)):

- For each group of steps \( steps\\_T \) and \( k \):

- For all combinations \( C \subset steps\\_T \) where \( |C| = k+1 \):

\[

\exists i, j \in C, i \neq j : assignments\_i = assignments\_j

\]

- Implemented by ensuring that in each combination of \( k+1 \) steps, at least one pair shares the same user.

5. One-Team:

- For each group of steps \( G \) and teams \( T \):

- The assignments of steps in \( G \) must be such that all steps are assigned to users within the same team in \( T \).

- Implemented via allowed assignments:

\[

assignments\_G \in AllowedAssignments

\]

6. Precedence:

\[

\forall (s1, s2) \in Precedence, assignments\_{s1} < assignments\_{s2}

\]

---

2. `solver\_og\_symmetryk.py`

Formulation

Variables:

- Assignment Variables: Integer variables `assignments[i]` for each step `i`, where `assignments[i] ∈ {1, 2, ..., users\_count}`.

- At-Most-k User Variables: For each At-Most-k constraint, creates `k` integer variables `user\_vars[i]` representing the unique users assigned to the group of steps.

Constraints:

1. Authorisations, Separation-of-Duty, Binding-of-Duty:

- Same as in `solver\_mine\_combinatorial.py`.

2. At-Most-k:

- Symmetry-Breaking Approach:

- Creates `k` integer variables `user\_vars[i]`, each representing a unique user.

- Enforces ordering on `user\_vars` to break symmetry:

\[

user\\_vars[i] \leq user\\_vars[i+1]

\]

- For each step in the group, it must be assigned to one of the `user\_vars`:

\[

assignments[step] = user\\_vars[i], \text{for some } i \in \{1, ..., k\}

\]

- Introduces binary variables to indicate which `user\_var` a step is assigned to and ensures exactly one is selected.

3. One-Team:

- For each One-Team constraint:

- Team Selection Variables: Boolean variables representing whether a team is selected.

- Ensures exactly one team is selected.

- For each step in the group, the assigned user must be in the selected team.

- Overlapping Steps Handling:

- For steps appearing in multiple One-Team constraints, enforces compatibility between team selections:

- If two teams for overlapping constraints have no common users, they cannot both be selected.

Advantages and Disadvantages

Advantages:

- Symmetry Breaking:

- Ordering `user\_vars` reduces the search space by eliminating equivalent solutions arising from permutations of users.

- This can significantly improve solver performance on instances with many users and steps.

- Efficient Handling of At-Most-k:

- Avoids combinatorial explosion by not generating all possible combinations.

- Uses fewer variables compared to the combinatorial approach.

- Conflict Resolution in One-Team Constraints:

- Handles overlapping steps in multiple One-Team constraints by ensuring team selections are compatible.

- Reduces infeasibility due to conflicting assignments.

Disadvantages:

- Increased Complexity:

- Introduces additional variables and constraints, which can increase the problem size.

- The handling of overlapping One-Team constraints can be complex and may not scale well for a large number of overlapping constraints.

- Potentially Larger Number of Variables:

- For each At-Most-k constraint, `k` additional integer variables are introduced.

- For steps in At-Most-k groups, additional binary variables are created.

Best Use Cases

- Large-Scale Problems with Many Users:

- Instances where symmetry can significantly impact solver performance.

- Problems with large At-Most-k groups and where `k` is relatively small compared to the number of steps.

- Complex Constraints:

- Scenarios where steps appear in multiple One-Team constraints.

- Problems requiring careful handling of overlapping constraints.

Problem Formulation

Let:

- \( S = \{1, 2, ..., n\} \) be the set of steps.

- \( U = \{1, 2, ..., m\} \) be the set of users.

- \( assignments\_i \in U \) for each step \( i \in S \).

Variables:

- Assignments:

- \( assignments\_i \in U \) for each step \( i \in S \).

- At-Most-k User Variables:

- For each At-Most-k constraint with group \( G \) and \( k \):

- \( user\\_vars\_j \in U \) for \( j = 1 \) to \( k \).

Constraints:

1. Authorisations, SoD, BoD:

- Same as in the first solver.

2. At-Most-k:

- Symmetry Breaking:

\[

user\\_vars\_j \leq user\\_vars\_{j+1}, \forall j = 1 \text{ to } k-1

\]

- Assignment Constraints:

- For each step \( i \in G \):

\[

assignments\_i = user\\_vars\_j, \text{for some } j \in \{1, ..., k\}

\]

- Implemented by:

- Introducing binary variables \( is\\_assigned\_{i,j} \in \{0,1\} \).

- Adding constraints:

\[

\sum\_{j=1}^{k} is\\_assigned\_{i,j} = 1

\]

\[

assignments\_i = user\\_vars\_j, \text{if } is\\_assigned\_{i,j} = 1

\]

3. One-Team:

- Team Selection:

- For each One-Team constraint:

- Binary variables \( team\\_selected\_t \in \{0,1\} \) for each team \( t \).

- Constraint:

\[

\sum\_{t} team\\_selected\_t = 1

\]

- Assignment Constraints:

- For each step \( i \) in the group and team \( t \):

- If \( team\\_selected\_t = 1 \), then \( assignments\_i \in team\\_t \).

- Overlapping Steps:

- For overlapping steps between constraints, ensure that incompatible team selections are not both selected.

---

3. `solver\_symmetryk\_simplified\_uservars.py`

Formulation

This solver is similar to `solver\_og\_symmetryk.py` but simplifies the handling of user variables.

Variables:

- Assignment Variables: Integer variables `assignments[i]` for each step `i`.

- At-Most-k User Variables: For each At-Most-k constraint, creates `k` integer variables `user\_vars[i]`.

- Assignment Indicator Variables: Binary variables indicating whether a step is assigned to a particular `user\_var`.

Constraints:

1. Authorisations, Separation-of-Duty, Binding-of-Duty:

- Same as previous solvers.

2. At-Most-k:

- Symmetry Breaking:

\[

user\\_vars[i] < user\\_vars[i+1]

\]

- Assignment Constraints:

- For each step in the group:

- Binary variables \( is\\_assigned\_{i,j} \in \{0,1\} \) for each `user\_var`.

- Constraints:

\[

\sum\_{j=1}^{k} is\\_assigned\_{i,j} = 1

\]

\[

assignments\_i = user\\_vars\_j, \text{if } is\\_assigned\_{i,j} = 1

\]

- This approach reduces the number of variables and constraints compared to previous methods.

3. One-Team:

- Similar to `solver\_og\_symmetryk.py`.

Advantages and Disadvantages

Advantages:

- Reduced Complexity:

- Simplifies the At-Most-k constraint implementation.

- Fewer variables and constraints may lead to faster solving times.

- Symmetry Breaking:

- Still employs symmetry breaking through ordering of `user\_vars`.

Disadvantages:

- Potentially Less Flexible:

- The simplification may limit the solver's ability to handle certain complex scenarios.

- May not handle overlapping constraints as efficiently as more complex formulations.

Best Use Cases

- Medium to Large Problems:

- When a balance between complexity and performance is needed.

- Suitable for problems where At-Most-k groups are not overly large.

- Simpler Constraints:

- Instances where overlapping One-Team constraints are minimal or non-existent.

Problem Formulation

Similar to `solver\_og\_symmetryk.py`, but with simplified assignment indicators.

---

4. `solver\_doreen.py`

Formulation

Variables:

- Assignment Variables: Binary variables `user\_assignment[s][u]` indicating whether step `s` is assigned to user `u`.

Constraints:

1. Each Step Assigned to Exactly One User:

\[

\forall s, \sum\_{u} user\\_assignment[s][u] = 1

\]

2. Authorisations:

- For users with specified authorisations:

\[

\text{If } s \notin allowed\\_steps[u], \quad user\\_assignment[s][u] = 0

\]

3. Separation-of-Duty (SoD):

- For each SoD pair `(s1, s2)`:

\[

user\\_assignment[s2][u] = 0 \text{ if } user\\_assignment[s1][u] = 1, \forall u

\]

- Implemented using `OnlyEnforceIf`.

4. Binding-of-Duty (BoD):

- For each BoD pair `(s1, s2)`:

\[

user\\_assignment[s2][u] = 1 \text{ if } user\\_assignment[s1][u] = 1, \forall u

\]

5. At-Most-k:

- Introduces binary variables `user\_assignment\_flag[u]` indicating whether user `u` is assigned to any of the steps in the group.

- Constraints:

\[

\sum\_{u} user\\_assignment\\_flag[u] \leq k

\]

\[

user\\_assignment\\_flag[u] \geq user\\_assignment[s][u], \forall s \text{ in group}, \forall u

\]

6. One-Team:

- Introduces binary variables `team\_flag[t]` for each team `t`.

- Constraints:

\[

\sum\_{t} team\\_flag[t] = 1

\]

- For each team:

- If `team\_flag[t] = 1`, steps in the group must be assigned to users in team `t`.

- Ensures that steps are not assigned to users outside of any team.

Advantages and Disadvantages

Advantages:

- Simple and Direct Modeling:

- Uses binary variables for assignments, which can be more intuitive.

- Constraints are modeled directly, making it easier to understand and modify.

- Effective for Small to Medium Problems:

- The approach can be efficient when the number of steps and users is not too large.

Disadvantages:

- Scalability Issues:

- The number of variables is \( O(steps \times users) \), which can become large.

- May not perform well for instances with a large number of users and steps.

- Lack of Symmetry Breaking:

- Does not employ techniques to reduce the search space caused by symmetries.

- Can lead to longer solving times due to redundant exploration.

Best Use Cases

- Small Instances:

- Problems with a small number of steps and users.

- Situations where an easy-to-understand model is preferred.

- Educational Purposes:

- Useful for learning how constraints can be modeled using binary variables.

Problem Formulation

Let:

- \( S = \{1, 2, ..., n\} \), steps.

- \( U = \{1, 2, ..., m\} \), users.

- Binary variables \( user\\_assignment[s][u] \in \{0,1\} \).

Constraints:

1. Each Step Assigned to Exactly One User:

\[

\forall s \in S, \sum\_{u \in U} user\\_assignment[s][u] = 1

\]

2. Authorisations:

\[

user\\_assignment[s][u] = 0, \text{ if } s \notin allowed\\_steps[u]

\]

3. Separation-of-Duty (SoD):

\[

user\\_assignment[s2][u] = 0, \text{ if } user\\_assignment[s1][u] = 1, \forall u, \forall (s1, s2) \in SoD

\]

4. Binding-of-Duty (BoD):

\[

user\\_assignment[s2][u] = 1, \text{ if } user\\_assignment[s1][u] = 1, \forall u, \forall (s1, s2) \in BoD

\]

5. At-Most-k:

- Binary variables \( user\\_assignment\\_flag[u] \in \{0,1\} \).

- Constraints:

\[

\sum\_{u \in U} user\\_assignment\\_flag[u] \leq k

\]

\[

user\\_assignment\\_flag[u] \geq user\\_assignment[s][u], \forall s \in G

\]

6. One-Team:

- Binary variables \( team\\_flag[t] \in \{0,1\} \) for each team \( t \).

- Constraints:

\[

\sum\_{t} team\\_flag[t] = 1

\]

\[

user\\_assignment[s][u] = 0, \text{ if } u \notin team\_t, \forall s \in G, \forall t, \text{ when } team\\_flag[t] = 1

\]

---

Summary and Recommendations

Comparative Analysis

- `solver\_mine\_combinatorial.py`:

- Pros: Direct modeling, optimized At-Most-k for small `k`.

- Cons: Scalability issues due to combinatorial explosion.

- Best For: Small problems with small At-Most-k groups.

- `solver\_og\_symmetryk.py`:

- Pros: Effective symmetry breaking, efficient handling of At-Most-k.

- Cons: Increased complexity, more variables.

- Best For: Large problems where symmetry reduction is crucial.

- `solver\_symmetryk\_simplified\_uservars.py`:

- Pros: Simplified variable usage, balance between complexity and performance.

- Cons: May not handle complex overlapping constraints as efficiently.

- Best For: Medium to large problems with moderate complexity.

- `solver\_doreen.py`:

- Pros: Simple and intuitive modeling.

- Cons: Not scalable due to the large number of variables.

- Best For: Small instances or educational purposes.

Recommendations

- For Large and Complex Problems:

- Use `solver\_og\_symmetryk.py` or `solver\_symmetryk\_simplified\_uservars.py` to benefit from symmetry breaking and efficient constraint handling.

- For Small to Medium Problems:

- `solver\_mine\_combinatorial.py` can be effective if At-Most-k constraints are manageable.

- `solver\_doreen.py` is suitable if simplicity is preferred over performance.

- When Overlapping One-Team Constraints Exist:

- `solver\_og\_symmetryk.py` provides mechanisms to handle overlapping steps effectively.

- For Educational Understanding:

- `solver\_doreen.py` offers a clear view of how constraints can be modeled using binary variables.

---

Final Notes

Each solver has been designed with different priorities in mind, balancing between performance, complexity, and simplicity. The best solver for a given problem depends on the specific characteristics of the problem instance, such as the number of steps, users, and the nature of the constraints.

When selecting a solver:

- Analyze the Problem Size: Larger problems benefit from symmetry breaking and efficient constraint modeling.

- Consider the Constraint Types: Complex constraints with overlaps require more sophisticated handling.

- Balance Complexity and Performance: Additional variables and constraints can improve solver performance but may increase the model's complexity.

By understanding the formulations and trade-offs of each solver, you can choose the one that best fits your problem and performance requirements.  
  
19 instances named; 0.txt,1.txt,...,19.txt and its solutions named; 0-solution.txt,1- solution.txt,...,19- solution.txt  
19 instances

First-order logic, Knowledge-Based Systems, Propositional Resolution, FOL Resolution, Horn Clauses, Prolog, Constraint satisfaction