Performance Evaluation of XSB, Clingo, Souffle, and Alda via Transitive Closure Rules

John Owolabi Idogun

May 16, 2024

Contents

List of Tables

List of Figures

Glossary

Graph Size The number of distinct nodes or vertices n in a graph. Here, given a number n, the nodes or vertices are numbered from 1 to n. For example, a graph of size 3 will have nodes 1, 2, and 3.

Acronyms

DR Double Recursion

LR Left Recursion

RR Right Recursion

TC Transitive Closure

List of Algorithms

Project proposal and justification

1.1 Problem Description

This experiment evaluates the performance of four rule systems —Alda [12], Clingo [5, 6], Soufflé [9], and XSB [23] —via computing Transitive Closure (TC) using different rule variants—specifically LR, RR, and DR. The objective is to analyze and compare the running times of each of the rule systems for each variant for various Graph Sizes and shapes.

1.2 Input/Output

1.2.1 Input

The input for this study includes:

1. Rule systems: Alda [12], Clingo [5, 6], Soufflé [9], and XSB [23].
2. Rules: Variants of TC (LR, RR, and DR).
3. Shapes of data: Different graph structures including star and grid graphs, and those covered in *rbench* [2] —complete, cyclic, maximum acyclic, star (1 at its center), cyclic graph with shortcuts, path, multi-path, binary tree, reverse binary tree, y, w, and x graphs.

1.2.2 Output

The output will consist of performance plots detailing the running times for each variant of the TC rules in the four rules systems.

1.3 State of the Art

The efficiency and scalability of TC computations in various computational models and systems have been a subject of intense scrutiny due to their wide-ranging applications in **graph theory**, **database systems**, and **semantic web technologies** and this makes it perfect for evaluating the performance of rule systems. This review aims to synthesize contributions from seminal works in the field, identifying gaps that this experiment intends to address.

OpenRuleBench: An Analysis of the Performance of Rule Engines by Liang et al. (2009) marks a significant step towards understanding the performance landscape of rule-based systems at the web-scale [11]. By benchmarking eleven systems across five technologies, OpenRuleBench highlighted performance variations in handling semantic information, including TC computations on cyclic and acyclic graphs. However, its analyses were constrained by the singular execution of benchmarks and a focus on graph sizes of 6000 and 24000, limiting the breadth of performance insights garnered.

In parallel, Performance Analysis and Comparison of Deductive Systems and SQL Databases by Brass and Wenzel (2019) provided a comparative lens through which the performances of logic programming systems and relational databases were examined for TC computations [2]. Their methodology, which involved running tests multiple times to calculate average performance metrics, offered a more reliable performance evaluation. Furthermore, the

inclusion of a diverse set of graph shapes, such as complete graphs, acyclic graphs, cycle graphs, and more, presented a comprehensive view of system capabilities. Despite these advancements, the study's coverage of rule systems and TC variants was not exhaustive, omitting considerations for star and grid graphs and other TC forms. Also, there was no separation between the different stages of the TC computation, which could have provided more insights into the performance of the rule systems.

Regarding general TC computations, Dar and Ramakrishnan (1994) delivered a foundational performance evaluation of TC algorithms, providing a broad analysis across acyclic graphs with varying densities [4]. This study illuminated the cost trade-offs between different TC algorithms, setting a precedent for subsequent research. Agrawal, Dar, and Jagadish (1990) introduced direct TC algorithms that significantly deviated from iterative approaches, offering a predictable performance model that furthered the discourse on TC computations [1]. Cacace et al. (1993) explored the untapped potential of parallel execution strategies for TC computations, advocating for the leverage of parallelism in enhancing computational efficiency [3]. Lemström and Hella (2003) ventured into the domain of approximate pattern matching integrated with TC logics, extending the applicability of TC operations to complex data analysis tasks [10].

This experiment aims to build upon these insights, introducing a broader spectrum of rule systems and graph shapes into TC performance analysis. By examining three distinct variants of TC—LR, RR, and DR—across a more extensive corpus of rule systems and graph structures, this experiment endeavours to unveil and compare dimensions of efficiency and scalability of rule systems. It also seeks to unravel the performance intricacies of rule systems at each stage of computation from rules and data loading to writing query results to output files. This will provide a more nuanced understanding of the performance characteristics of rule systems in TC computations, thereby contributing to the broader discourse on rule-based systems and their applications. This meticulous methodology and the inclusion of more rule systems and graph shapes will address existing gaps and extend the current understanding of rule system performance in TC computations across various computational models.

1.4 Tasks and Subtasks

The project is organized into a series of tasks and subtasks aimed at a thorough evaluation of TC rule performance:

1. **Experimental Setup**:
   - Preparation of graph datasets by generating different shapes of data/graphs.
   - Configuration of rule systems —Alda [12], Clingo [5, 6], Soufflé [9], and XSB [23].
   - Implementation of TC rule variants —LR, right or tail recursion, and DR.
2. **Data Collection and Analysis**: Execution of experiments to collect running times and subsequent analysis to determine the efficiency and scalability of each rule variant in Alda [12], Clingo [5, 6], Soufflé [9], and XSB [23].
3. **Result Synthesis**: Compilation of findings into a coherent analysis, highlighting the performance characteristics of LR, RR, and DR in the context of TC computations.

System Design and API

2.1 Introduction

This chapter delves into the intricacies of the architecture and APIs of the system designed for the automated evaluation of transitive closure rule variants across multiple rule systems. Emphasis is placed on the system's flexibility in incorporating and analyzing new rule files without necessitating adjustments to the existing codebase.

2.2 System Design Overview

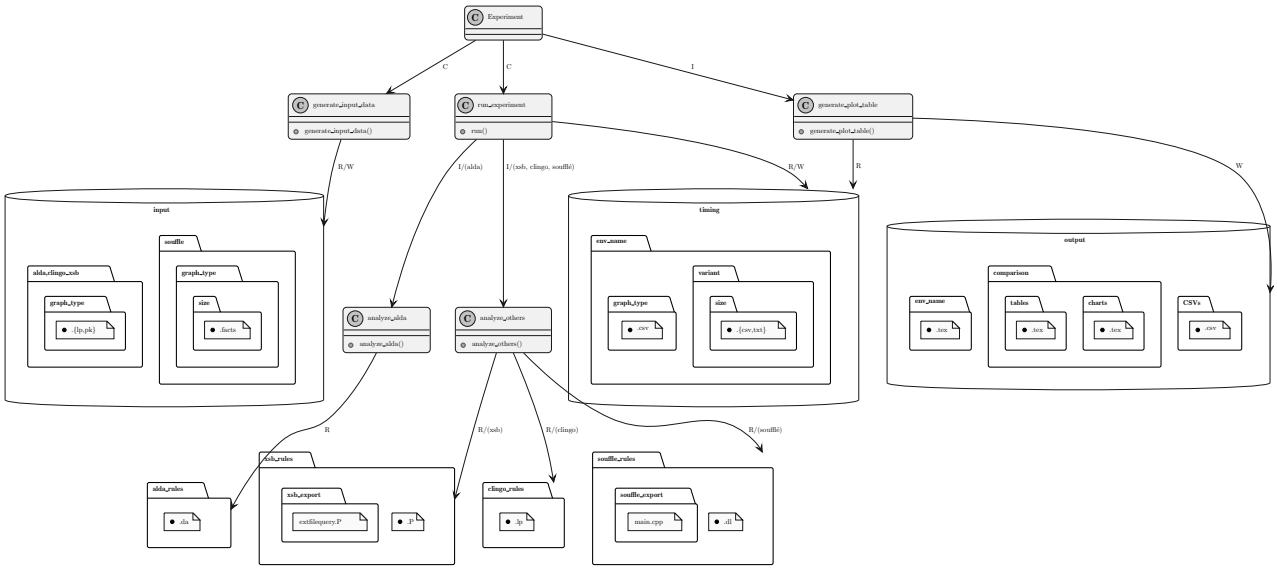


Figure 2.1: General Overview of the system

Figure 2.1 represents the architecture of the system developed. The main orchestrator, the `transitive` file, coordinates the overall workflow. It calls the `generate_input_data` method to prepare necessary input files, the `run_experiment` method to execute experiments based on specific parameters, and invokes the `generate_plot_table` file to produce detailed output plots and tables in LaTeX. The `run_experiment` method conditionally triggers `analyze_alda.da` or `analyze_others.py`, depending on the environment setting, which in turn read their respective rule files from designated directories (`alda_rules`, `xsb_rules`, `clingo_rules`, `souffle_rules`). Each interaction within the system is defined to ensure precise execution and documentation of experiments, tailoring operations to the parameters and environment specified.

2.3 Component Design and API

2.3.1 transitive.py: Central Experiment Orchestrator

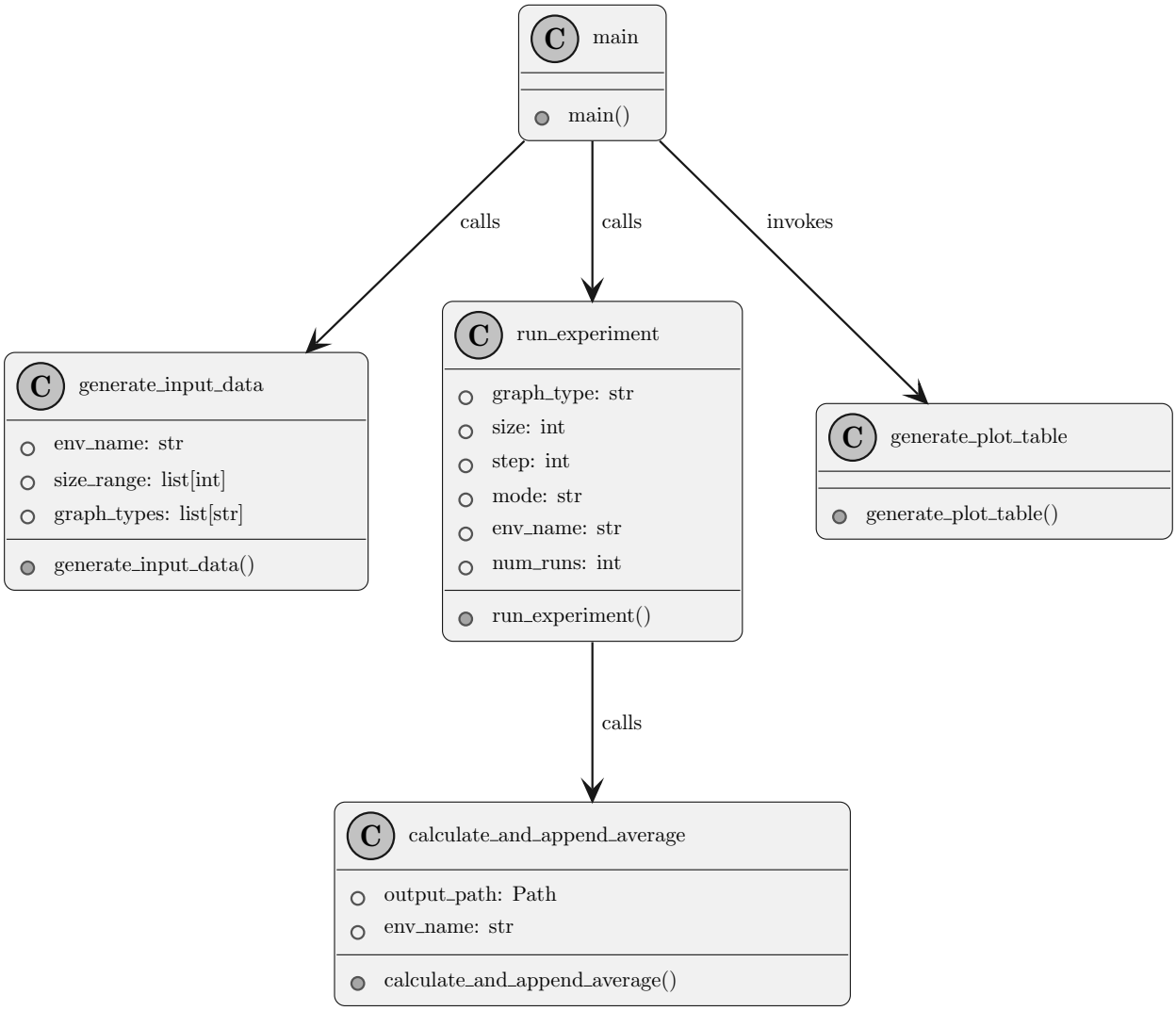


Figure 2.2: Central Experiment Orchestrator overview

The cornerstone of the experiment's workflow, `transitive.py`, manages the entire lifecycle of the experiment as . It acts as the main entry point for initiating experiments, orchestrating the data generation, execution of analysis scripts, and the aggregation and visualization of results.

Key Functionalities

generate_input_data Prepares input data for a given rule system. Suppose the input directory for a rule system is empty or without some data for the specified size and graph types. In that case, it generates new graph datasets of varying shapes and sizes using `generate_db.py` (see 2.3.2). XSB and Clingo use the same data format hence their data generation was combined. For Soufflé, tab-separated files were generated. Alda uses a set of Python tuples to represent the graph data. These differences in data formats made it necessary to have separate folders for Soufflé and Alda.

run_experiment Orchestrates the experiment for a specified graph type, size, mode (recursion variant), and rule system. It determines the correct command to run based on the rule system. For Alda, it uses the analysis script `analyze_alda.da` while a generic script `analyze_others.py` is used for Clingo, Soufflé and XSB. This script performs environment-specific tasks to prepare the analysis. `run_experiment` then executes the analysis script the specified number of times (10 times by default) to account for potential variations in performance.

calculate_and_append_average After executing the experiments for a specific graph type, size, and mode, this function retrieves the collected timings data from a CSV file. It then calculates the average elapsed time and CPU time across all runs, ensuring outliers don't significantly skew the results. Finally, it appends these averages to their respective output CSV files for further analysis.

Plot and table generation `transitive.py` concludes the experiment cycle by invoking `generate_plot_table.py`, a script dedicated to generating LaTeX files for data visualization and tabulation. This process is automated and triggered after all experiments are completed, ensuring that performance data across various configurations are visualized and tabulated consistently.

Command-Line Arguments and Libraries

`transitive.py` leverages the `argparse` library to manage command-line arguments. This allows users to customize the experiment's behaviour through various options:

`--size-range` Defines the range of Graph Sizes to be evaluated (start stop step). Defaults to `10 101 10`. Because of Python's `range`, endeavour to add 1 to the intended stop. For the defaults, the intended stop was `100`.

`--num-runs` Specifies the number of times to repeat each experiment for increased confidence (defaults to `10`).

`--modes` Selects the Transitive Closure rule variants to be compared (e.g., left, right, double recursion).
Defaults to `right_recursion left_recursion double_recursion`.

`--environments` Chooses the rule system to be included in the experiment (defaults to `clingo xsb souffle alda`).

`--graph-types` Selects the graph shapes to be used for data generation (defaults to all the 12 graph shapes, see item 3 in 1.2.1).

`--souffle-include-dir` Specifies the directory containing the Soufflé C++ headers and libraries. This is necessary for executing Soufflé programs.

For file system operations, `transitive.py` utilizes the `pathlib`[20] library for path manipulation and the `subprocess`[22] module to execute external commands like invoking the logic program solvers and the data generation script (`generate_db.py`).

By providing these arguments and leveraging these libraries, `transitive.py` offers a flexible and configurable approach to experiment execution.

2.3.2 generate_db.py: Graph Database Generator

This script generates graph data for the experiments. It is designed to create graph datasets of varying sizes and shapes.

Graph Generation and Serialization

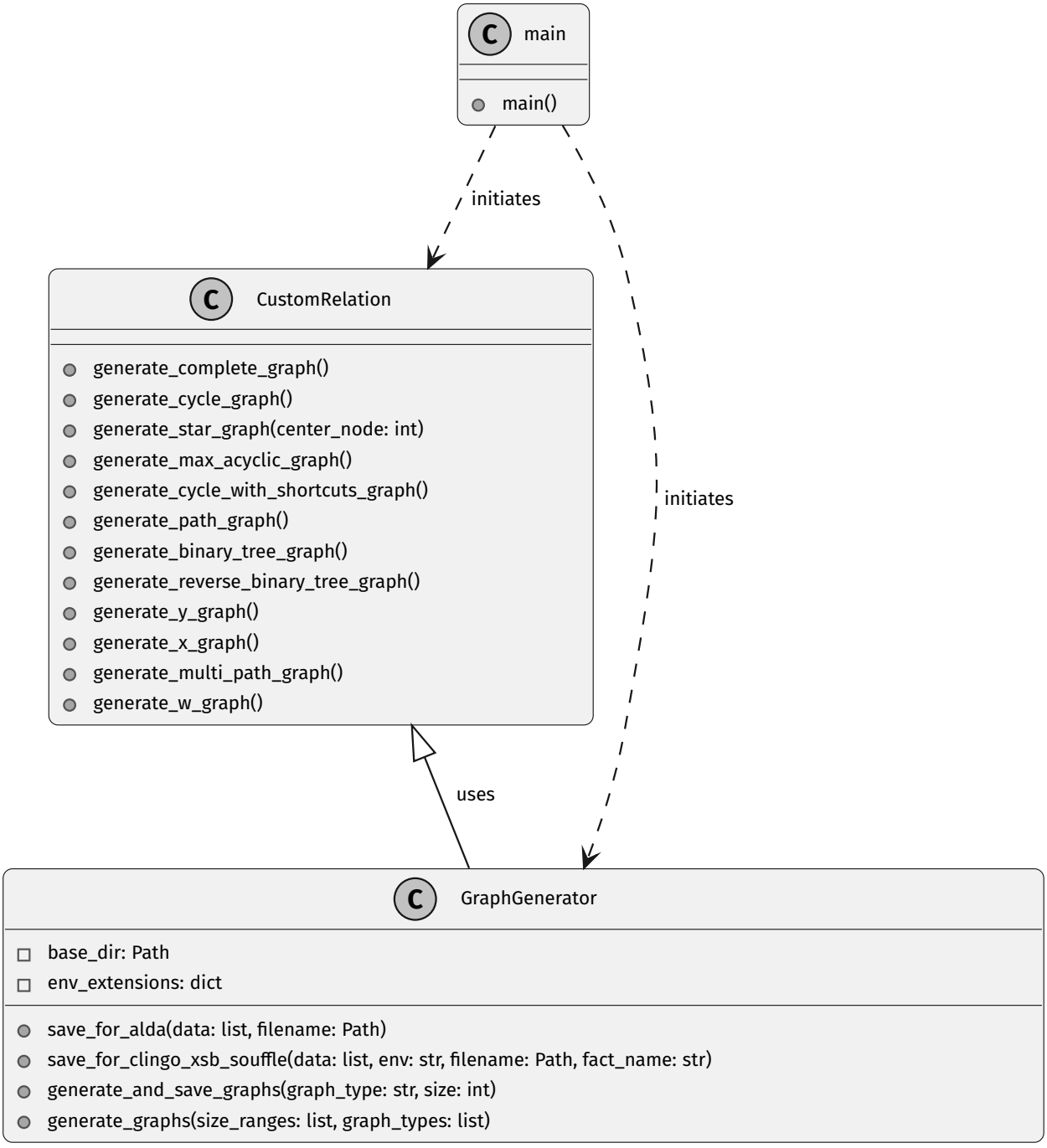


Figure 2.3: Graph Database Generator overview

DataGenerator Class (`DataGenerator`): Responsible for generating graph data of varying sizes and shapes. The implementations follow the specifications in rbench [2] with the following modifications:

- **Binary Tree and Reverse Binary Tree:** To ease computational complexity, h was set to $\lfloor \log_2(n) \rfloor$ where n is the number of nodes.
- **Circle with shortcut, W-, Y-, X- and Multipath-graphs:** The maximum value of k was set to 10.
- **Star:** The Star-graph S_n consists of $n-1$ vertices each pointing to a central vertex. There are no vertices that the central vertex points to, effectively making the star graph a simpler structure compared to the X-graph. The set of edges is $E = (i, 1) | i = 2, ..., n$.
- **Grid:** The Grid-graph G_h consists of $h \times h$ vertices arranged in a grid. Each vertex is connected to its immediate right neighbor and its immediate bottom neighbor, forming a regular grid pattern. The set of edges is defined as follows:
  - Horizontal edges: $E_H = \{(j, j+1) \mid j \in [(i-1)h+1, (i-1)h+h-1], i = 1, 2, \ldots, h\}$
  - Vertical edges: $E_V = \{(j, j+h) \mid j \in [(i-1)h+1, (i-1)h+h], i = 1, 2, \ldots, h-1\}$

  where $h = \lfloor \sqrt{n} \rfloor$ and n is the number of nodes. The complete set of edges is $E = E_H \cup E_V$ [14].

GraphGenerator Class (`GraphGenerator`): Oversees the generation and formatting of graph data for each rule system. It takes specifications for graph types and sizes, and performs the following tasks:

- Generates graph data using the appropriate methods from the `CustomRelation` class.
- Saves the generated graph data in a format compatible with the target environment (e.g., Alda, Clingo, Soufflé, XSB).

By providing this functionality, `GraphGenerator` ensures compatibility and facilitates easy ingestion of graph data by the analysis scripts in `transitive.py`.

Command-Line Arguments and Libraries

While `generate_db.py` can be invoked directly, it's typically called by `transitive.py`. However, it still offers some command-line arguments for customization:

`–sizes` : Defines the range of Graph Sizes to be generated (start, stop, step).

`–graph-types` : Selects the graph shapes to be used for data generation (e.g., complete, cycle, star).

Internally, `generate_db.py` utilizes the following libraries:

`argparse` [16] For parsing command-line arguments.

`logging` [17] For logging messages about the generation process (e.g., errors, warnings).

`math` [18] For mathematical operations used in graph generation algorithms (e.g., calculating the number of edges in a complete graph).

`pickle` [21] For serializing graph data in a format suitable for Alda (using `.pickle` extension).

pathlib [20] For path manipulation tasks (e.g., creating directories for storing generated graphs).

By leveraging these arguments and libraries, `generate_db.py` provides a flexible and efficient way to generate and store graph datasets for experiments in transitive closure computations.

2.3.3 analyze_others.py and analyze_alda.da: Core Analysis

These scripts perform the computational heavy lifting, analyzing the efficiency of transitive closure rules within their respective rule system. However, they take different approaches to rule management and environment interaction.

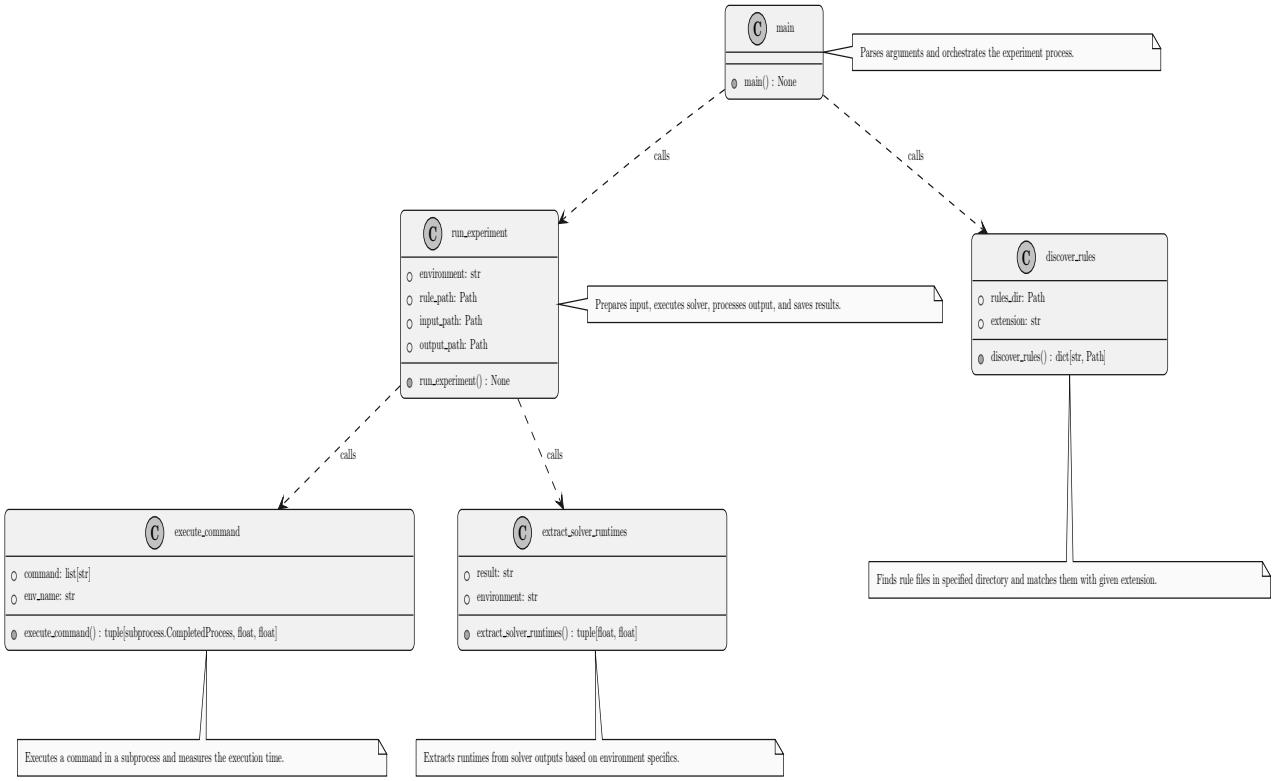


Figure 2.4: `analyze_others.py` Core Analysis overview

analyze_others.py

This script focuses on analyzing rule files written in different logic programming languages (clingo, XSB, Soufflé) and integrates with external solvers. Here are the key design aspects:

- **Automatic Rule File Analysis:** It employs a naming convention-based approach to automatically discover rule files in a designated directory (`xsb_rules` for rules particular to xsb). The file extension (e.g., `.lp` for Clingo) helps determine the target environment. This allows for easy addition of new rule sets without modifying the core analysis logic.
- **Environment-Specific Execution:** The script leverages the `ENVIRONMENT_EXTENSIONS` dictionary to map file extensions to their corresponding rule systems. Based on the chosen environment (`'clingo'`, `'xsb'`, or `'souffle'`), it constructs the appropriate command to execute the rule file along with the generated graph data.

- **Timing and Result Collection:** The script measures the real (wallclock) time and CPU (user and system) time by using different methods for each environment, see section 3.2.3. It then appends these timings to a CSV file for further analysis.

analyze_alda.da

This script is specific to the Alda logic programming environment. It relies on Alda's built-in mechanisms for loading and executing rules, drawing inspiration from the methodologies proposed by Liu et al [13], particularly in [25]:

- **Rule Discovery:** The `discover_rules` function is instrumental in automatically identifying rule files within the `alda_rules` directory based on a naming convention. It parses the file name to extract the rule mode (e.g., `right_recursion`) and employs `importlib` to dynamically construct corresponding module and class names, facilitating modular rule management and the seamless integration of new rule sets.
- **Data Loading:** Utilizes Python's `pickle` module to deserialize pre-generated graph data from a `.pickle` file, making the data available for analysis.
- **Rule Class Instantiation:** Dynamically imports the rule class corresponding to the identified rule mode using `importlib`, instantiates it with the loaded graph data and a designated output path for results. This step integrates the rule logic with the experiment's data flow.
- **Execution and Timing:** While Alda internally manages rule execution, the script extends its functionality by leveraging the `os.times()`[19] function from Python's standard library to capture detailed timing information (user CPU time, system CPU time, and real elapsed time). This approach ensures a comprehensive measurement of execution performance, aligning with the methodologies utilized in the non-Alda rule systems for consistency in performance evaluation.
- **Limited Environment Interaction:** Exhibits a more abstracted interaction with the logic programming environment compared to `analyze_others.py`, leveraging Alda's high-level functionalities for rule management and execution, thus simplifying the experimental setup.

In essence, `analyze_alda.da` and `analyze_others.py` pursue the analysis of the rule systems via transitive closure rules through distinct methodologies tailored to the respective rule systems. While `analyze_others.py` is designed for flexibility across various logic programming languages by interfacing with external solvers, `analyze_alda.da` optimizes for the Alda environment, utilizing its native features and Python's `os.times()` for performance measurement.

2.3.4 generate_plot_table.py: Plot & Table Generation

`generate_plot_table.py` is designed to automate the generation of visual plots and detailed tables representing timing data for experiments. It employs LaTeX for high-quality output, utilizing the `pgfplots` package for plots and the `tabularx` and `siunitx` packages for tables. The script processes CSV files containing timing data across different environments, graph types, and modes, and then organizes this data into LaTeX files specific to each combination, facilitating detailed performance analysis and comparison.

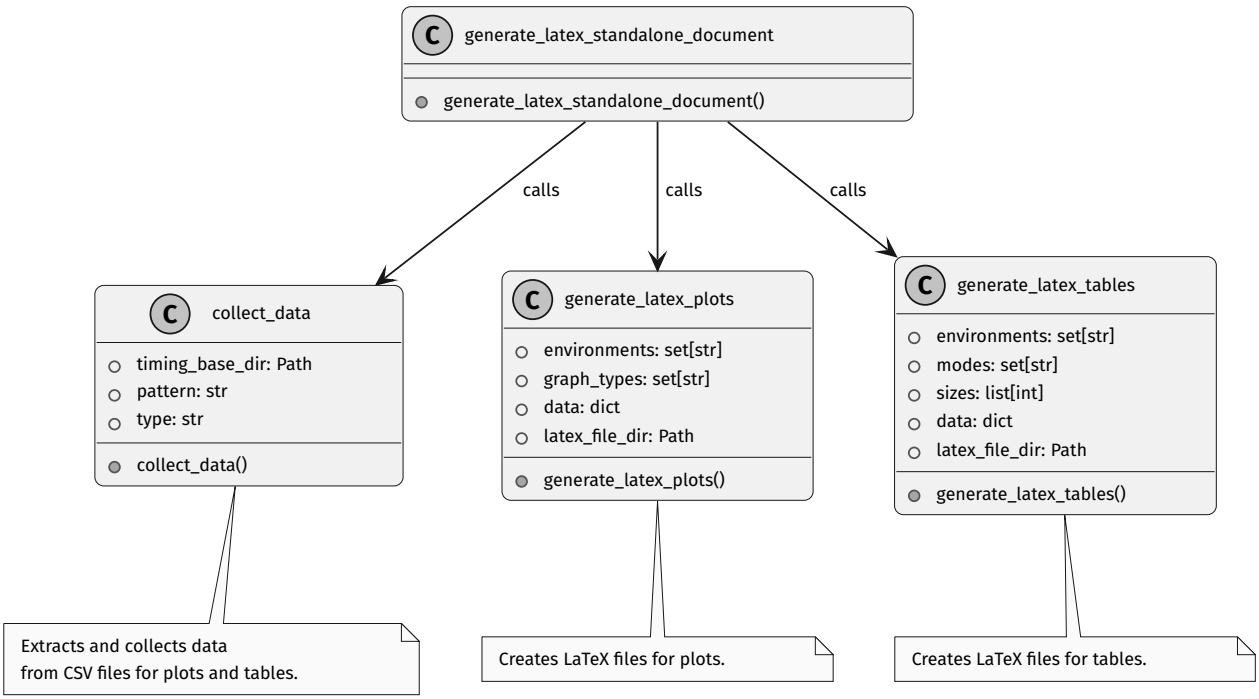


Figure 2.5: Plot & Table Generation overview

Data Collection and Processing

The script functions to:

1. Scan the specified `timing` directory for CSV files that adhere to a predefined naming convention.
2. Extract and process timing data from these files, focusing on average timings (execution time and CPU time) from the last line of each file.
3. Organize and store this data systematically according to the environment, graph type, mode, and Graph Size which aids in generating targeted plots and tables.

Plot Generation

It generates varieties of complex plots using the `pgfplots` package in LaTeX, ensuring high-quality visualizations that are consistent across different environments and graph types. The plots are designed to provide a clear and detailed representation of the performance data, facilitating easy comparison and analysis.

Table Generation

The script generates tables in LaTeX format, providing a structured and detailed representation of the performance data. It employs the `tabularx` package to create tables with adjustable column widths, ensuring optimal readability. The `siunitx` package is used to format numerical data consistently, enhancing the tables' clarity and precision.

Usage and Customization

The script is flexible and can be easily adapted to different data structures or presentation styles. Adjustments can be made to accommodate additional data metrics, alter the visual

style of plots and tables, or extend the script's functionality to include additional types of analyses.

2.3.5 Adding New Rule Files

To add a new rule file for analysis, a user simply needs to place the file in the appropriate directory (`alda_rules`, `clingo_rules`, etc.) following the naming convention (`transitive_<variant>.<extension>`). The system's design ensures that the new file if properly written and conforms with the required convention, is automatically discovered and included in the analysis without requiring any changes to the scripts. By adhering to established naming conventions and placing these files in the correct directories, researchers can extend the scope of the analysis without modifying the core scripts. This design choice underscores the system's emphasis on modularity and extensibility.

2.3.6 Design Choices

`transitive.py`

- **Language Choice:** Initially implemented as a Bash script, `transitive.py` was migrated to *Python* due to limitations inherent to shell scripting. These limitations included:
  - Requirement for permission changes before execution.
  - Increased complexity when writing the script itself.
  - Environmental restrictions.
- **Enhanced User Experience:** The script exposes *command-line arguments*, allowing users to interact with the program without modifying the code directly. Additionally, default values are provided for all arguments, ensuring seamless execution even without user-specified data.
- **Efficiency:** Graph data generation is skipped if the input folder is not empty. Also, it deletes empty subdirectories within the timing directory and skips running experiments for sizes that already have corresponding .csv files. This is to optimize the execution process.
- **Rule System specific execution:** For fairness, the script utilized the best practices for each rule system. Clingo's Python API was used, Soufflé's C++ interface was employed, and XSB's timing was done using a script written in Prolog.

`generate_db.py`

- **Extensibility:** The `generate_db.py` script was created as an extension of `gen_db.py` to overcome limitations in the latter's functionalities. These limitations included:
  - Difficulty in serializing data for Clingo and Soufflé.
  - Challenges in generating data with the required shapes.
- **Improved Script Interaction:** Similar to `transitive.py`, command-line arguments were implemented within `generate_db.py`. This facilitates effortless data generation without requiring script modifications.

`generate_plot_table.py`

- **Visualization Choice:** *LaTeX* was chosen over Matplotlib for generating plot files due to the superior quality and clarity of visualizations produced by LaTeX.
- **Informative Output:** The generated LaTeX files create comparative plots and tables showcasing the performance of different TC variants for each graph type used within a specific rule system. These visual and tabular representations allow for easy performance evaluation across different scenarios.

`analyze_alda.da`

- **Rule File Naming Convention and Dynamic Loading:** A critical aspect of `analyze_alda.da`'s design is its reliance on a naming convention to dynamically discover and load rule files. This facilitates easy expansion of the rule set without the need to alter the core analysis logic. For instance, adding a new file named `transitive_right_recursion.da` automatically integrates it into the analysis process, assuming the class within adheres to the naming convention, being `TransitiveRightRecursion` in this case. This convention not only streamlines the management of rule files but also enhances the system's modularity and scalability.
- **Internal Execution and Timing Measurement:** Both the elapsed time and CPU time measurements are intricately handled within the respective rule file for Alda, adhering to methodologies borrowed from Liu et al [13].

Project Report

3.1 Introduction

In computing, accurate performance evaluations are crucial for optimizing algorithms and systems. Benchmarking systematically measures system performance, helping to understand how algorithms perform under various conditions and in different rule systems. This project focuses on the performance evaluation of rule systems in computing transitive closure (TC) operations, a fundamental operation in graph theory and databases that assesses node reachability.

Implementing these algorithms in logic-based programming frameworks like Prolog and Answer Set Programming (ASP) allows for concise, readable code, enhancing understandability and maintainability. However, performance in these systems can be unpredictable, underscoring the need for rigorous benchmarking.

Existing benchmark suites like rbench [2] and openrulebench [11] have explored transitive closure benchmarking but have not addressed the implementation variants — such as RR, LR, and DR — explored in this project. These variants represent different logical approaches, each with unique performance profiles that vary by the logic programming environment.

Unlike rbench, this project does not use a dedicated database for result storage but manages results directly within the file system, simplifying the architecture and enhancing flexibility to include a wider variety of graph shapes. The tested rule systems — XSB, clingo, alda, and souffle — are significant in the logic world. It not only measures the real-time performance of these systems but also the CPU time, from various stages of the process, providing a comprehensive view of their efficiency.

By conducting multiple iterations of experiments in a configurable manner, it also automates data gathering and analysis. This ensures repeatability and reliability in benchmark tests, offering detailed insights crucial for performance tuning and improving the effectiveness of logic programming techniques.

It bridges the theoretical advantages of logic programming — expressive power and ease of use — with the practical necessity for performance optimization, shedding light on the performance dynamics of various rule systems via transitive closure variants.

3.2 Implementations

3.2.1 Transitive Closure Rules

We outline core transitive closure rules implemented across various programming environments—RR, LR, and DR. The complete source code, including utility functions and integration with the benchmarking scripts, is available on GitHub [7]. Access the rules in their respective directories for XSB `xsb_rules`, Soufflé `souffle_rules`, Clingo `clingo_rules`, and Alda `alda_rules`.

XSB Prolog

XSB Prolog uses a logic programming paradigm that directly supports recursion. The `auto_table` directive is crucial as it enables tabling to avoid redundant computations and ensures that recursive queries terminate properly. Here are the transitive closure rule variants:

RR The recursive call occurs after traversing an edge, making this approach straightforward but potentially less efficient on large graphs.

```
:- auto_table.
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
```

LR By initiating the recursive call before traversing the final edge, this method can leverage previously computed paths, potentially increasing efficiency.

```
:- auto_table.
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), edge(Z, Y).
```

DR This method recursively searches for paths to an intermediate node and from that node to the target, which can be computationally expensive.

```
:- auto_table.
path(X, Y) :- edge(X, Y).
path(X, Y) :- path(X, Z), path(Z, Y).
```

Comparison with Clingo and Soufflé

XSB Prolog uses explicit predicates and clauses, while Clingo and Soufflé handle recursion implicitly. Clingo's equivalent RR is:

```
path(X, Y) :- edge(X, Y).
path(X, Y) :- edge(X, Z), path(Z, Y).
#show path/2.
```

Soufflé requires explicit type declarations, shown in LR implementation:

```
.decl edge(x:number, y:number)
.input edge

.decl path(x:number, y:number)
.output path
path(x,y) :- edge(x,y).
path(x,y) :- path(x,z), edge(z,y).
```

Alda TC Implementation

Alda combines Python's readability with rule-based logic:

```
class TransitiveRightRecursion(Transitive, process):
    def rules_TransitiveRightRecursion():
        path(x, y), if_(edge(x, y))
        path(x, y), if_(edge(x, z), path(z, y))
```

This syntax highlights how Alda combines declarative logic with Python's procedural style, making it potentially more accessible to developers familiar with traditional programming languages but less concise in expressing purely logical constructs compared to XSB or Clingo.

3.2.2 Benchmarking Scripts

Brief discussion on the benchmarking scripts are written here. A more comprehensive detail can be found in Chapter 2.

Main Orchestrator: transitive.py The `transitive.py` script automates the generation of input data, execution of experiments, and visualization of results.

Graph Database Generator: generate_db.py `generate_db.py` handles graph creation and outputs them for experiments.

Data Analysis and Execution: analyze_others.py `analyze_others.py` was designed for executing and analyzing performance in rule systems like Clingo, XSB, and Soufflé. It shares certain similarities with analyze_alda.da.

Table and Plot Generation: generate_plot_table.py The `generate_plot_table.py` script processes CSV data and constructs plots and tables in LaTeX.

3.2.3 Timing

The description of the timing mechanisms for each rule system is provided below. It should be noted that the query that was timed is `path(X, Y).` for all the rule systems.

3.2.4 XSB

XSB timing was done in three parts:

Loading the facts and rules: Rules were loaded using prolog's `consult` predicate, while facts were loaded using the fast `load_dync` predicate. Using the `statistics` predicate with `runtime` and `walltime` options, the times taken to load facts and rules were measured separately.

Executing the query: The time taken to execute the query was measured using the the same `statistics` predicate with `runtime` and `walltime` options. The time measured here was only for the query execution. This was recorded as the query time.

Writing the results: The query was executed again but this time the results were written to a text file using the `writeq` predicate. Then the difference in time between the time taken to execute only the query and the time taken to execute the query and write the results was calculated. This difference was taken as the time taken to write the results, writing time.

To achieve this, an XSB script, adapted from [26], was used. The script is available in the `extfilequery.P` file. The outputs were then parsed to extract the times. In total, four phases were timed for XSB and each phase had both the real time and CPU time measured.

3.2.5 Clingo

Clingo timing was done in five phases. Using its Python interface [15], the facts and rules were loaded using the `load` method on the `Control` object. The times to do this were measured separately using Python's `os` module. Next, the object was grounded using the `ground` method and time taken was measure. Then, query was solved with `solve` method. In this phase, the `solve` method did not take any arguments. This time was also measured using Python's `os` module. Then, the query was solved again but this time the `yield` was

set to yield results. These outputs were iterated over and all atoms and terms as outputted by clingo were selected and written to a text file. The time taken to do this entire process was measured and the difference between this time and that of the previous phase was taken as the time taken to write the results. The script used for this is available in the `analyze_others.py` file. Five phases were timed for Clingo and each phase had both the real time and CPU time measured.

3.2.6 Soufflé

Soufflé's phases were a little different from the other rule systems. Its C++ interface [24] was used extensively here in the following phases:

Converting the Datalog file to a C++ file: The C++ equivalent of the Datalog file alongside the paths to the fact file and the output file was generated by Soufflé Datalog compiler. The time taken to do this was measured using Python's `os` module.

Compiling the C++ file: A driver code in C++, see `main.cpp`, was used in compiling the generated C++ file using `g++` with `C++17` standard. Compilation time was measured using Python's `os` module.

Loading instance and facts, executing (running) the query and writing the results: The driver code written measured the time taken to load Soufflé's instance from the generated C++ code and the facts, execute the query and write the results using C++'s `std::chrono` and `sys/resource.h` libraries. Facts were loaded using `loadAll`, the query was executed using `run` and the results were written to a `.csv` file. Each of these stages was timed separately and the results were parsed to extract the times.

In total, six phases were timed for Soufflé and each phase had both the real time and CPU time measured.

3.2.7 Alda

There was no separation of the time taken to load the facts and rules and the time taken to execute the query in Alda. The time taken to execute the query was measured using Python's `os` module.

3.2.8 Visualization and Table Generation

A dynamic Python script, `generate_plot_table.py` was developed to process the timing data and generate plots, tables and CSV files. For tables and plots, the generated files were in LaTeX formats. Provided that the system has a LaTeX distribution installed, the script can generate the tables and plots in PDF format.

3.2.9 Execution commands

The commands used in each analysis for each solver are as follows:

XSB

The implemented command for executing queries in XSB is:

```
...
xsb_query =
↪  f"extfilequery:external_file_query('{rule_file}','{fact_file}',{queries},'{results
xsb_command = [
    'xsb',
    '--nobanner',
    '--quietload',
    '--noprompt',
    '-e',
    f"add_lib_dir('{xsb_export_path}').",
    '-e',
    xsb_query,
]
...
```

The command consists of the following parts:

1. **`xsb_query`**: A formatted string that constructs the Prolog query to be executed. It uses the `external_file_query` predicate from the `extfilequery` module, with the rule file, fact file, a string of list of list of queries, and the results path as arguments. A typical `queries` string would be "[[UniqueQueryIdentifier, ActualQuery],[UniqueQuery
ActualQuery],...]". For example, "[[query1, path(X, Y)]]" was used.

2. **`xsb_command`**: A list that constructs the full command to be executed.

    - **`'xsb'`**: Starts the XSB Prolog interpreter.
    - **`'--nobanner', '--quietload', '--noprompt'`**: Flags to suppress the startup banner, quiet the loading of files, and suppress the prompt respectively.
    - **`'-e', f"add_lib_dir('xsb_export_path')."`**: Executes the `add_lib_dir` predicate to add the specified library directory to the library search paths.
    - **`'-e', xsb_query`**: Executes the constructed Prolog query.

Clingo

For clingo, its Python interface was used to load the facts and rules and solve the query as shown in the following code snippet:

```
ctl = clingo.Control()
ctl.load(str(fact_file))
ctl.load(str(rule_file))

ctl.ground([('base', [])])
ctl.solve()
```

Soufflé

Soufflé had a series of steps and the commands used are as follows:

```
# Convert the Datalog file to a C++ file
...
datalog_to_cpp_cmd = f'souffle {rule_file} -F {fact_file} -g
↪  {generated_cpp_filename} -D {output_folder}'
...

# Compile the C++ file
compile_cmd = f'g++ {souffle_export_file}.cpp {generated_cpp_filename}
↪  -std=c++17 -I {include_dir} -o {souffle_export_file}
↪  -D__EMBEDDED_SOUFFLE__'
...

# Execute the compiled file
run_cmd = f'./{souffle_export_file} {fact_file}'
...
```

The code consists of the following parts:

1. **`datalog_to_cpp_cmd`**: This command uses the Soufflé Datalog compiler to convert a Datalog program into a C++ file. The `-F` flag specifies the directory for fact files, `-g` generates a C++ file, `-D` specifies the directory where query results are stored.

2. **`compile_command`**: This command compiles the generated C++ file using `g++` with the `C++17` standard. The `-I` flag specifies the include directory for Soufflé's C++ headers, `-o` specifies the output file, and `-D__EMBEDDED_SOUFFLE__` is a macro definition.

3. **`execution_command`**: This command executes the compiled file with the fact file as an argument.

3.2.10 Code Sizes and Analysis

Transitive closure

The following table summarizes the code sizes for each implementation, measured in lines of code (LOC):

Table 3.1: Code Sizes of Transitive Closure Implementations

Environment	RR (LOC)	LR (LOC)	DR (LOC)	Total LOC
XSB Prolog	3	3	3	9
Clingo	3	3	3	9
Soufflé	6	6	6	18
Alda	7	7	7	21

Soufflé and Alda tend to have slightly more verbose implementations due to additional type declarations and Pythonic syntax, respectively. XSB and Clingo have concise implementations due to their direct support for recursion and implicit handling of types.

Benchmarking Scripts

Table 3.2: Code Sizes for Project Files.

File	Lines of Code (LOC)
`transitive.py`	404
`generate_db.py`	294
`analyze_others.py`	626
`analyze_alda.da`	86
`generate_plot_table.py`	911

Note that the line counts reflect not only the functional complexity of each script but also the use of docstrings, code style preferences, and various inline comments that can significantly inflate the line numbers.

3.2.11 Development Effort and Challenges

The project required several weeks of diligent effort, focused on developing, testing, and refining the code and its associated functionalities. The nuances of each rules system posed significant challenges. Not to mention handling everything from data generation to result visualization and interpretation on-the-fly without manual intervention.

Additionally, a significant portion of time was dedicated to documentation, which proved to be more time-consuming than the coding itself.

3.3 Tools and Technologies

This project utilizes a diverse array of tools and technologies for dataset generation, rule analysis, and data manipulation, detailed as follows.

3.3.1 Dataset Generation

gendb: The `generate_db.py` script automates graph dataset generation using the `gen_db`[8] library for various graph types like complete graphs, cycle graphs, and binary trees.

3.3.2 Rule Systems

Various rule systems are analyzed, each selected for its capabilities in transitive closure computations:

Alda: Supports Alda rule files with Python 3.12, integrating rule management and efficiency measurements.

Clingo v5.7.1: Utilized for answer set programming, managing rules in `.lp` format.

XSB v5.0.0: A Prolog-based system for tabled logic programming, suitable for Prolog rule files (`.P` files).

Soufflé v2.4.1-25-g57f104dea: A Datalog-based compiler for static analysis problems, used with Datalog rule files (`.dl` files).

3.3.3 Analysis and Development Tools

Python Libraries: Extensive use of Python 3.12 and libraries for scripting and data analysis:

- `argparse`, `logging`, `csv`, `pathlib`, and `subprocess` for command-line parsing, logging, file management, and process interaction.

C++ and C Libraries: Used for Soufflé's C++ interface and driver code:

- `chrono` and `sys/resource.h` for timing and resource management.

External Solvers: Interfaces with solvers like Clingo, XSB, and Soufflé for rule execution and data collection.

3.3.4 Version Control and Source Code

Git: Manages code versions and facilitates collaboration, accessible via [7].

3.3.5 Documentation and Reporting

LaTeX and PlantUML: LaTeXis used for document creation and PlantUML for UML diagram generation, ensuring detailed and structured project documentation. All the `.py` files have corresponding `.puml` files that help generate the UML diagrams.

3.4 Testing and Evaluation

3.4.1 System Configuration

Tests were conducted on a machine with a 2022 Apple M2 chip having a maximum clock rate of 3.50 GHz, 8 GB RAM, running macOS Sonoma 14.4.1, Python 3.12, XSB 5.0.0, Souffle 2.4.1-26-gc7ce22981, Clingo 5.7.1, and Alda 1.1.2rc16. The times reported were averaged over 10 runs.

3.4.2 Data and Test Cases

Data was systematically generated to cover a wide range of scenarios. Datasets incremented from 100 to 601 in steps of 100, resulting in six distinct data files for each graph type. These data can be found in the input. Testing began on 2024-05-08 at 00:57:25.957 and concluded on 2024-05-08 at 07:47:28.443. Both the timing and output contain the timing and results files, and the generated latex, CSV and PDF files for the tables and plots respectively.

3.4.3 Running the Analysis

Execute the `transitive.py` script:

```
python transitive.py --size-range 50 401 50
```

This generates all the data files needed and uses other defaults as discussed in section 2.3.1 of the design document. For this to be successful, your machine must have all the required tools installed as pointed out in the README.md.

You can specify the environment(s) you want test:

```
python transitive.py --size-range 50 401 50 --environments souffle
```

Or the number of runs, modes (variants), and graph types:

```
python transitive.py --size-range 50 401 50 --environments souffle xsb
↪  --num-runs 15 --modes right_recursion --graph-types complete cycle star
```

Since the script is designed to be flexible, you can specify any combination of the parameters to suit your needs. If you make a mistake in form of typos or wrong parameters, the script will guide you on the correct usage.

3.4.4 Results and Analysis

In the experimental setup, the performance of three variants of transitive closure algorithms—right recursion, left recursion, and double recursion—is compared across various graph types and sizes. In some cases, it may be seen that CPU times surpassed Real times. This is due to parallel processing, where the CPU time is the sum of the CPU times of all the cores used in the process.

Correctness checks

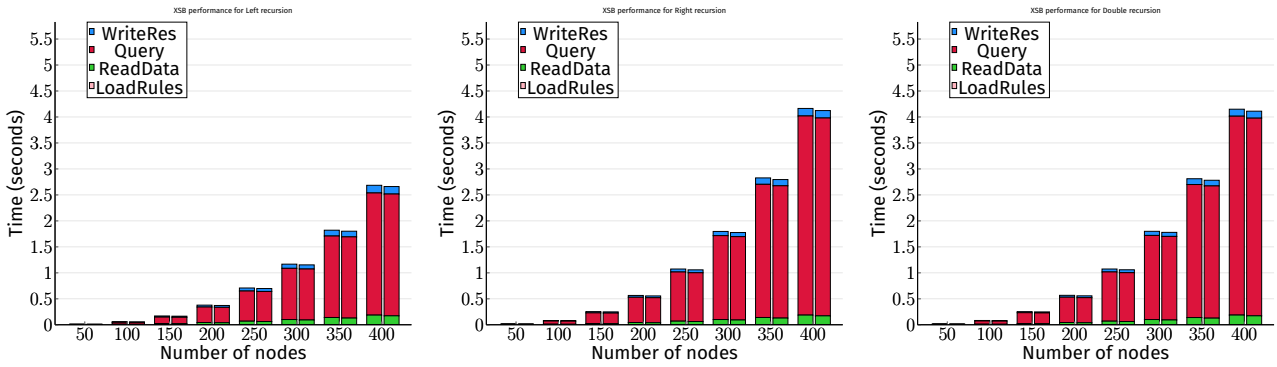
The correctness of the results was verified by comparing the outputs of each rule system. The results were consistent across all rule systems, confirming the accuracy of the implementations. They are in the `timing/<rule_system>/<graph_type>/<tc_variant>/<size>/` directory. For example, the results for XSB for the RR variant for a complete graph of size 100 can be found in the `timing/xsb/complete/right_recursion/100/xsb_results.txt` file. Due to size limit on GitHub, the results are not uploaded to github but were submitted.

XSB results

XSB results for each TC variant are provided below. The complete set of results is available in the output/xsb directory. As discused in section 3.2.3, the timing was done in four phases: loading rules (`Load Rules`), loading facts (`Read Data`), executing the query (`Query`), and writing the results (`Write Results`). In the following figures, the first bars in each graph represent the Real time, while the second bars represent the CPU time. This is consistent across all rule systems. Only a few graphs are shown here for brevity.

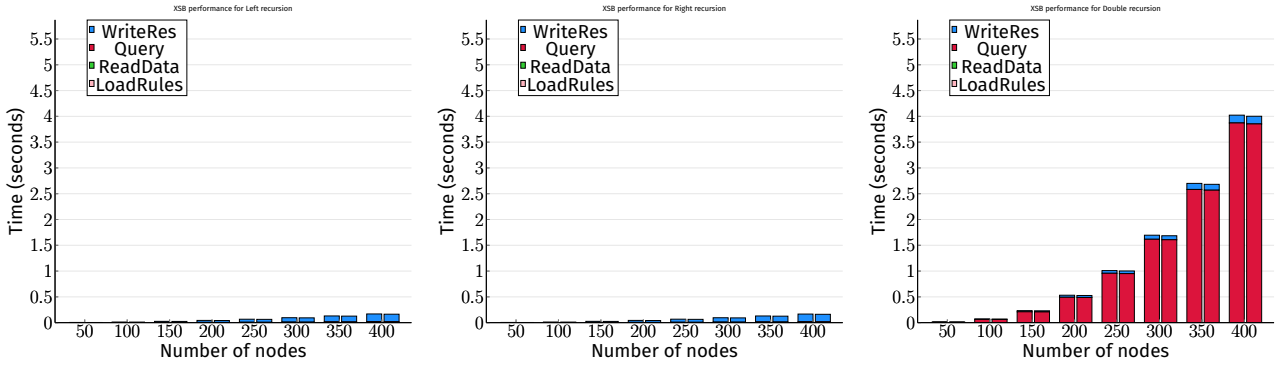
Left Recursion (LR) As depicted from Figure 3.1 to Figure 3.4, XSB's performance varies across different graph types and sizes. However, the general trend is that it exhibits linear or slightly superlinear time complexity, with data loading and query execution times increasing as the Graph Size grows.

Right Recursion (RR) The results indicate that XSB's performance is generally linear or slightly superlinear across various graph types, just like left recursion. However, right recursion tends to be slightly slower than left recursion as shown in Figure 3.1 to Figure 3.4.



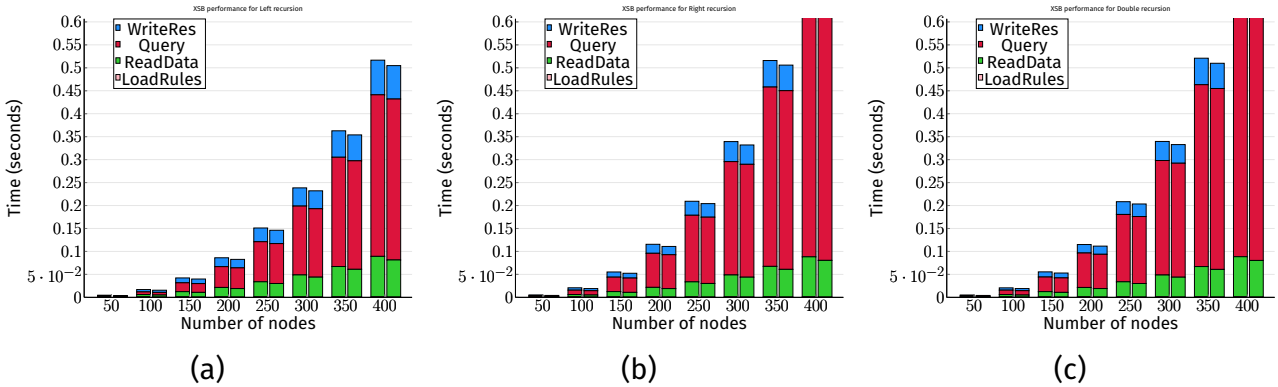
(a) (b) (c)

Figure 3.1: XSB performance for (a) LR, (b) RR, and (c) DR on Complete graphs.



(a) (b) (c)

Figure 3.2: XSB performance for (a) LR, (b) RR, and (c) DR on Cycle graphs.



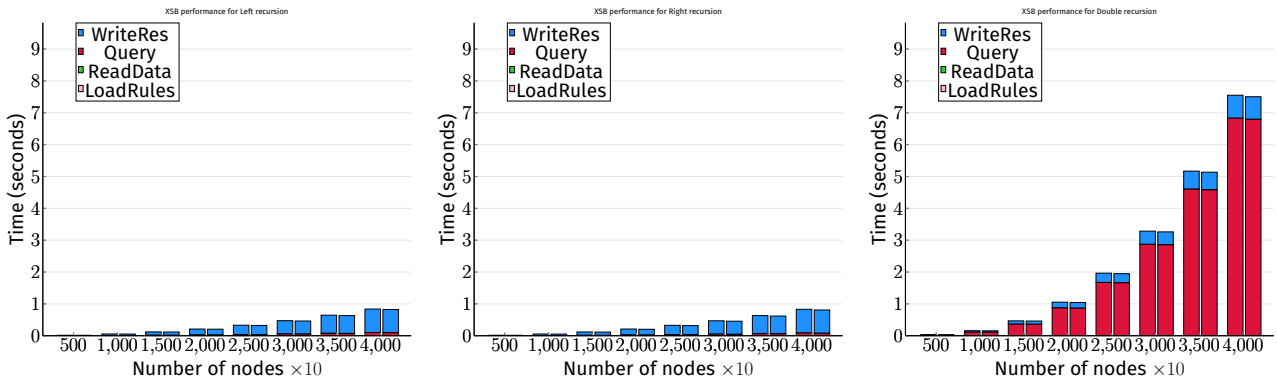
(a) (b) (c)

Figure 3.3: XSB performance for (a) LR, (b) RR, and (c) DR on Max Acyclic graphs.

Double Recursion (DR) For double recursion, the results show that XSB's performance is notably slower, with time complexity escalating quickly. This is likely due to redundant computations inherent in double recursion strategies.

Clingo results

The performance of Clingo for each transitive closure variant is as follows. The complete set of results is available in the output/clingo directory. Clingo's timing were done in five phases as discussed in section 3.2.3. The first phase was loading rules (`Load Rules`), the second phase was loading facts (`Read Data`), the third phase was grounding (`Ground`),

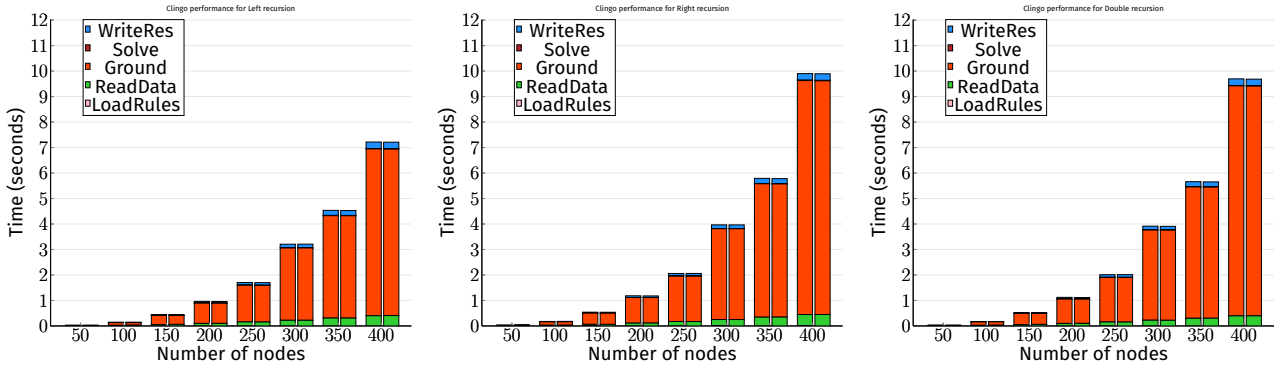


(a) (b) (c)

Figure 3.4: XSB performance for (a) LR, (b) RR, and (c) DR on Multi-Path graphs.

the fourth phase was solving the query (`Solve`), and the fifth phase was writing the results (`Write Results`). Grounding seems to be the most time-consuming phase in Clingo.

Left Recursion (LR) The results show that Clingo's performance is generally linear or slightly superlinear across various graph types, with data loading and query times mostly increasing as the Graph Size grows. This is consistent across all graph types, as shown in Figure 3.5 to Figure 3.8.

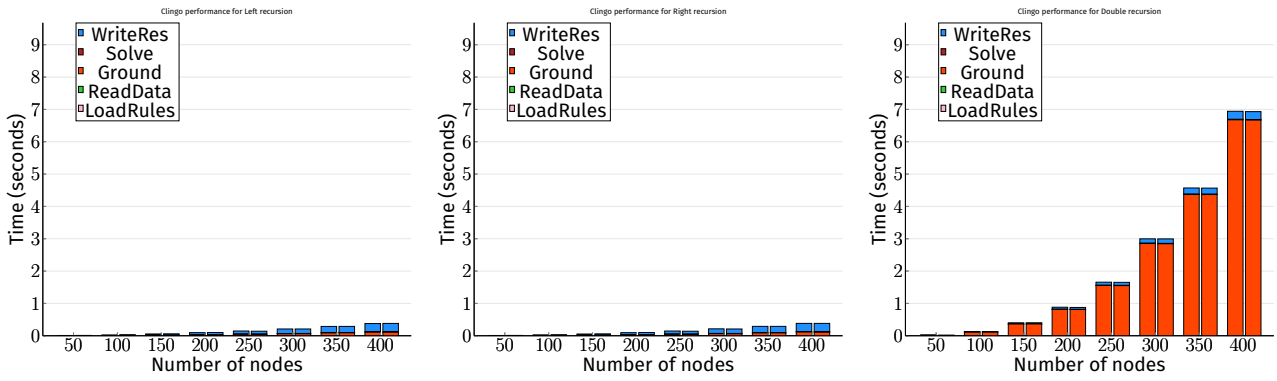


(a) (b) (c)

Figure 3.5: Clingo performance for (a) LR, (b) RR, and (c) DR on Complete graphs.

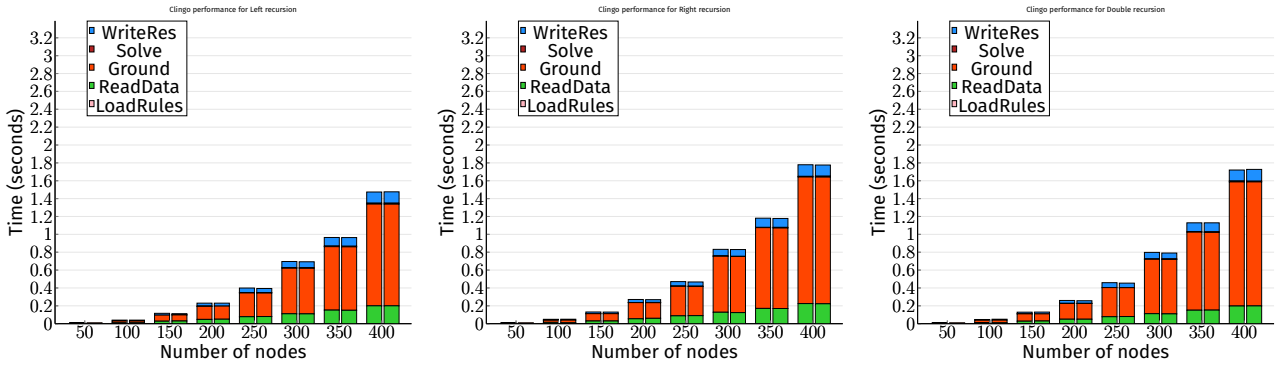
Right Recursion (RR) The results mirror those of left recursion. However, right recursion tends to be slightly slower than left recursion. The consistency of this trend is evident in Figure 3.5 to Figure 3.8.

Double Recursion (DR) Clingo's performance for double recursion is notably slower, with time complexity escalating quickly. Again, the redundancy in computations inherent in double recursion strategies is likely the cause. This is evident in Figure 3.5 to Figure 3.8 where results for complete, cycle, and multi-path graphs are shown by placing all the variants side by side.



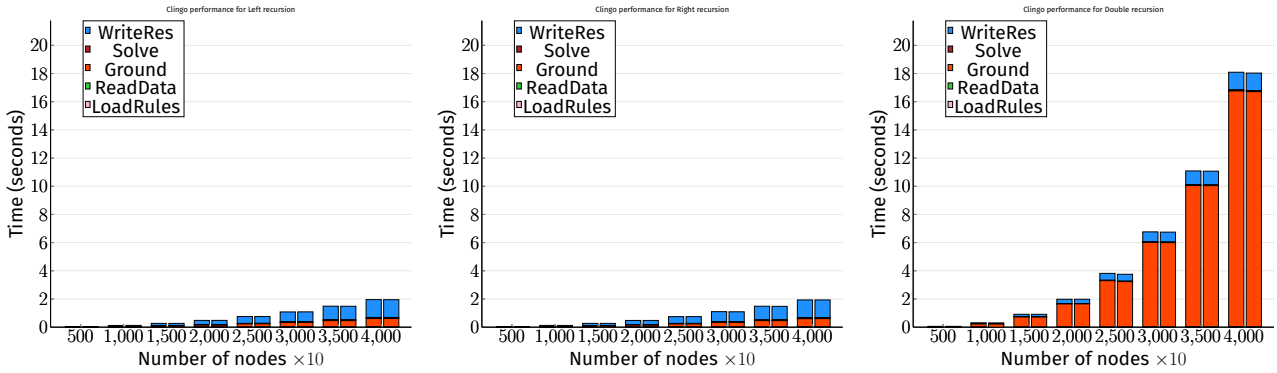
(a) (b) (c)

Figure 3.6: Clingo performance for (a) LR, (b) RR, and (c) DR on Cycle graphs.



(a) (b) (c)

Figure 3.7: Clingo performance for (a) LR, (b) RR, and (c) DR on Max Acyclic graphs.



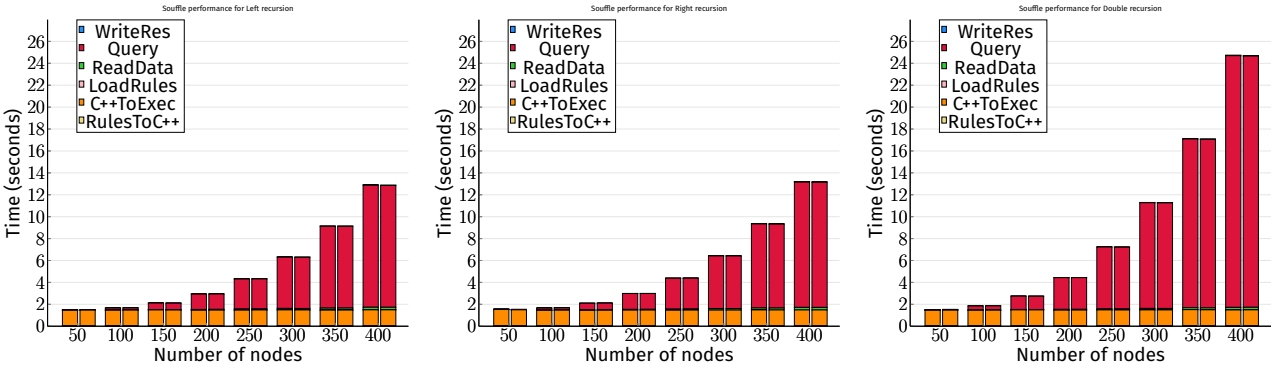
(a) (b) (c)

Figure 3.8: Clingo performance for (a) LR, (b) RR, and (c) DR on Multi-Path graphs.

Soufflé results

With reference to Soufflé, the performance of each transitive closure variant is as follows. The complete set of results is available in the output/souffle directory. Soufflé timing data is separated into program conversion from `Datalog` to `C++` (`Rules to C++`), compilation of the generated `C++` code to an executable (`C++ to Executable`), loading of Soufflé's instance (`Load Instance`), loading of facts (`Read Data`), running the query (`Run`), and writing the results to a file (`Writing`). The most resource-intensive step is the compilation of the generated `C++` code which is the most time-consuming.

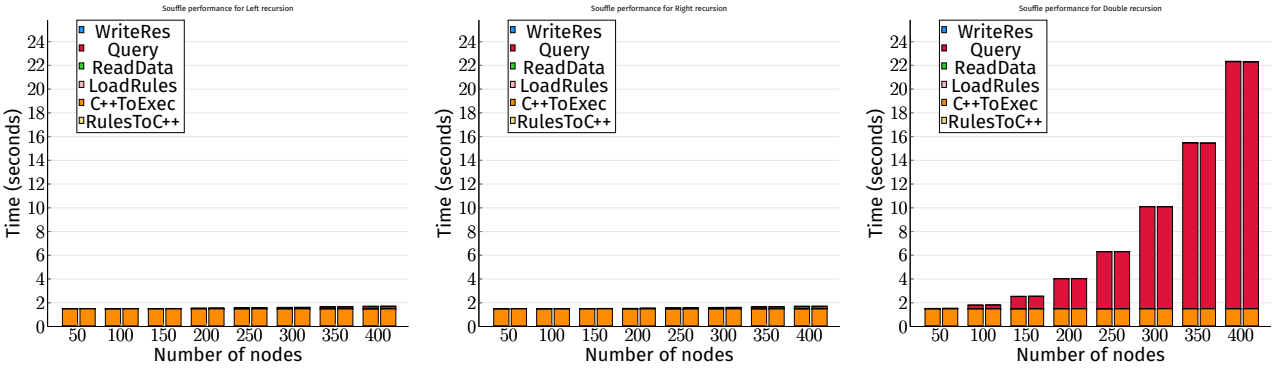
Left Recursion (LR) Soufflé's performance for all the variants is as shown in Figure 3.9 to Figure 3.12. Zooming in on the querying time, the results indicate that Soufflé's performance is generally linear or slightly superlinear across various graph types, with query (running) times increasing as the Graph Size grows.



(a) (b) (c)

Figure 3.9: Soufflé performance for (a) LR, (b) RR, and (c) DR on Complete graphs.

Right Recursion (RR) Soufflé's performance for right recursion mirror those of left recursion, with right recursion slightly slower than left recursion. Figure 3.9 to Figure 3.12 show the results for complete, cycle, and multi-path graphs.



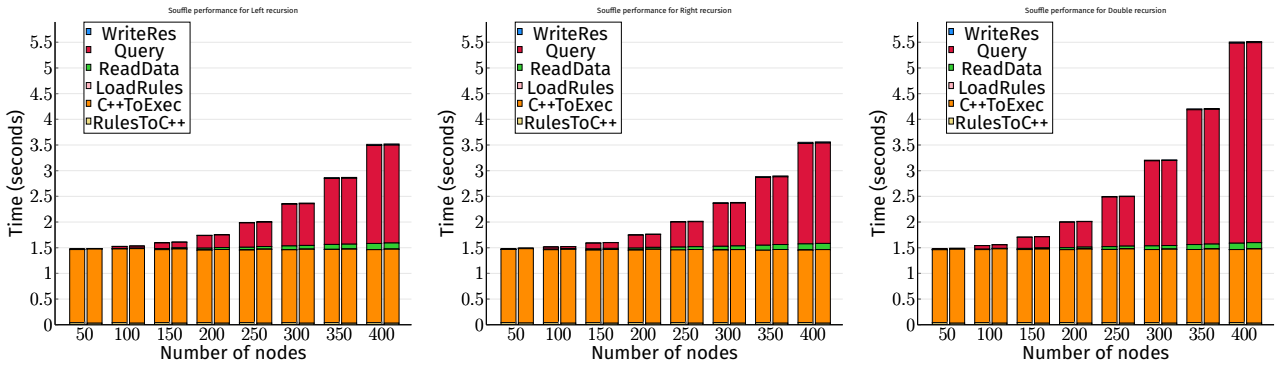
(a) (b) (c)

Figure 3.10: Soufflé performance for (a) LR, (b) RR, and (c) DR on Cycle graphs.

Double Recursion (DR) For double recursion, the results show that Soufflé's performance is notably slower for double recursion, with time complexity exploding. This strategy is the slowest of the three variants, as shown in Figure 3.9 to Figure 3.12.

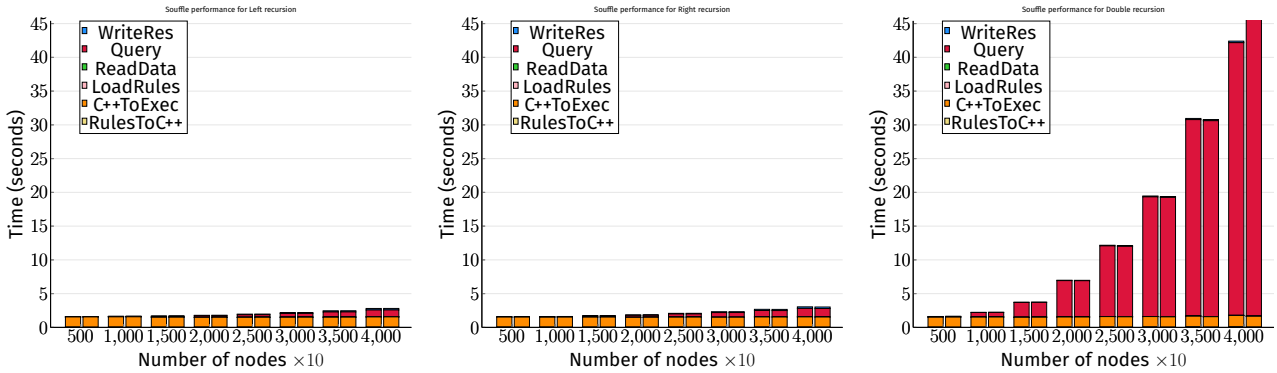
Alda results

The performance of Alda for each transitive closure variant is as follows. The complete set of results is available in the output/alda directory. For Alda, we only have the total time taken for each query execution.



(a) (b) (c)

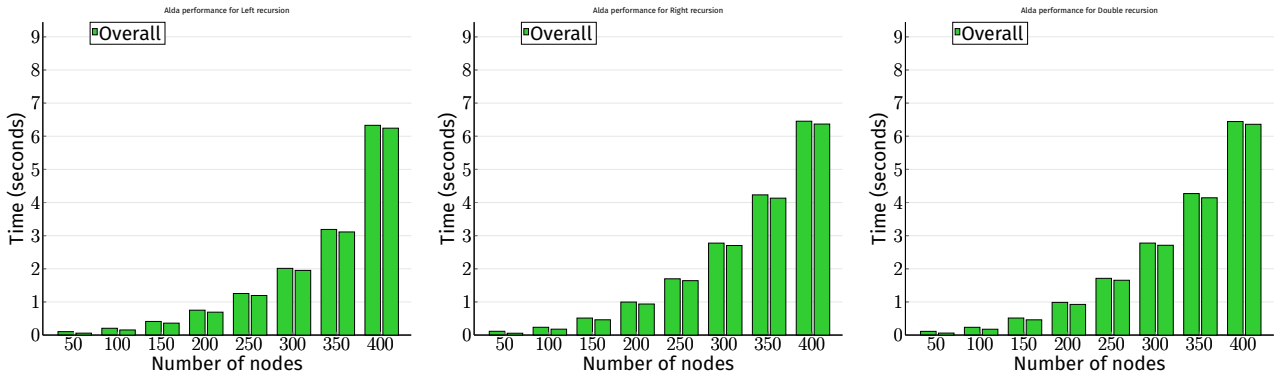
Figure 3.11: Soufflé performance for (a) LR, (b) RR, and (c) DR on Max Acyclic graphs.



(a) (b) (c)

Figure 3.12: Soufflé performance for (a) LR, (b) RR, and (c) DR on Multi-Path graphs.

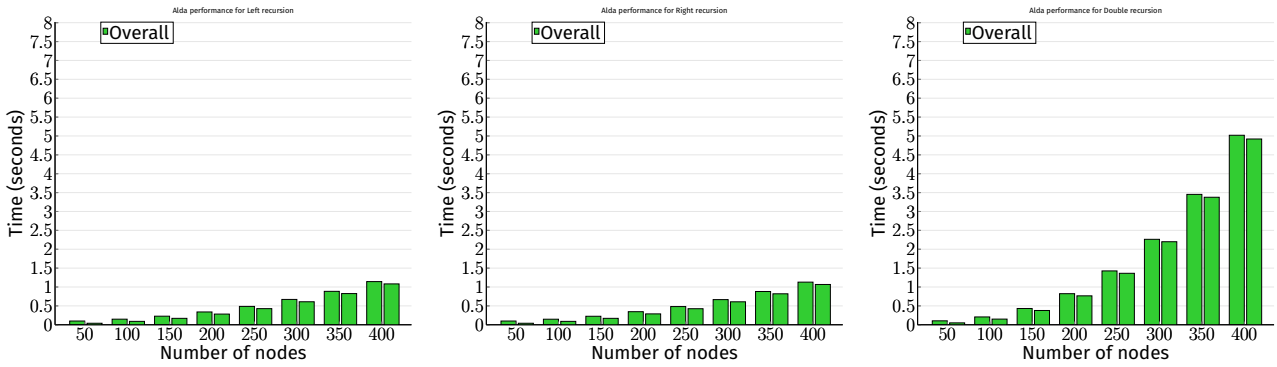
Left Recursion (LR) Just like XSB, Clingo, and Soufflé, Alda's performances for left recursion are as shown in Figure 3.13 to Figure 3.16 as the leftmost figures. The results indicate that Alda's performance is generally linear or slightly superlinear across various graph types, with query times increasing as the Graph Size grows.



(a) (b) (c)

Figure 3.13: Alda performance for (a) LR, (b) RR, and (c) DR on Complete graphs.

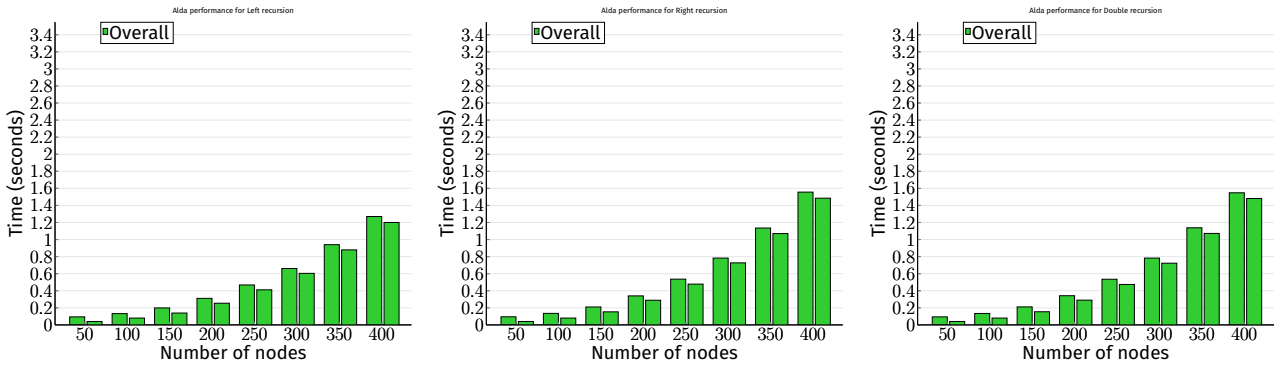
Right Recursion (RR) Alda's performance for right recursion mirrors that of left recursion, with right recursion slightly slower than left recursion. The plots are at the middle of Figure 3.13 to Figure 3.16.



(a) (b) (c)

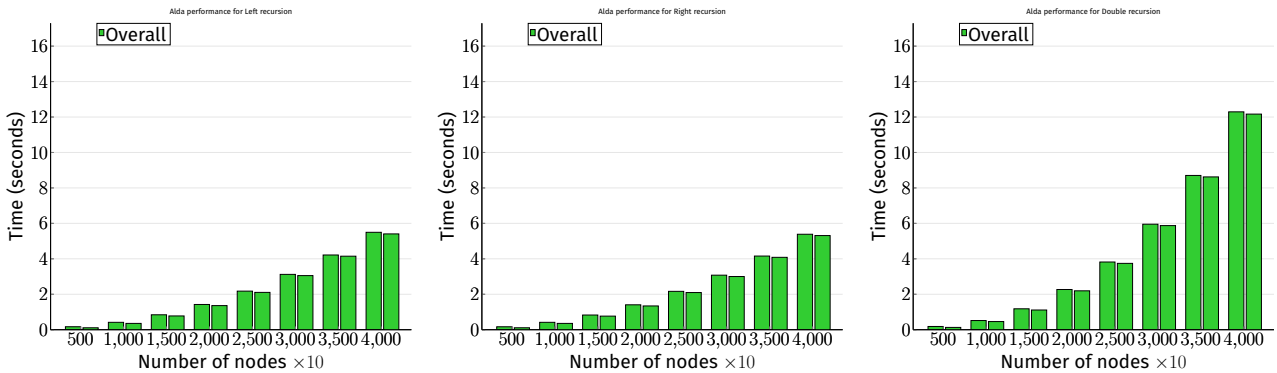
Figure 3.14: Alda performance for (a) LR, (b) RR, and (c) DR on Cycle graphs.

Double Recursion (DR) For double recursion, the results show that Alda's performance is notably slower for double recursion, with time complexity escalating quickly. This is likely due to the redundancy in computations inherent in double recursion strategies. This is evident in Figure 3.13 to Figure 3.16.



(a) (b) (c)

Figure 3.15: Alda performance for (a) LR, (b) RR, and (c) DR on Max Acyclic graphs.



(a) (b) (c)

Figure 3.16: Alda performance for (a) LR, (b) RR, and (c) DR on Multi-Path graphs.

3.4.5 Rule System Performance Comparison: XSB, Clingo, and Soufflé

The performance of XSB, Clingo, and Soufflé for each transitive closure variant was compared across various graph types and sizes. These comparisons stack up the CPU times of all the phases discussed section 3.2.3. In charts in the following subsections, some phases took longer than orders while some phases were very negligible. Hence, some of the metrics listed in the legend may not be visible in the charts. For closer and clearer comparisons, refer to the raw data files in the `timing` directory. In all, Souffle was the slowest, followed by Clingo, and XSB was the fastest. Souffle's longest phase was the `Running` phase which becomes more pronounced as the graph size increases. Clingo's longest phase was the `Grounding` phase, while XSB's longest phase was shared between the `writing` and `querying` phases. Included here are some of the graphs for comparison. The complete set of results is available in the output/comparison/charts/combined directory. In all the graphs, the left bars represent XSB, the middle bars represent Clingo, and the right bars represent Soufflé.

Comparison using `complete` graphs

Figure 3.17 shows the performance of XSB, Clingo, and Soufflé for each transitive closure variant on `complete` graphs. XSB consistently outperforms Clingo and Soufflé across all variants, with Clingo following closely behind. Soufflé lags behind in all comparisons.

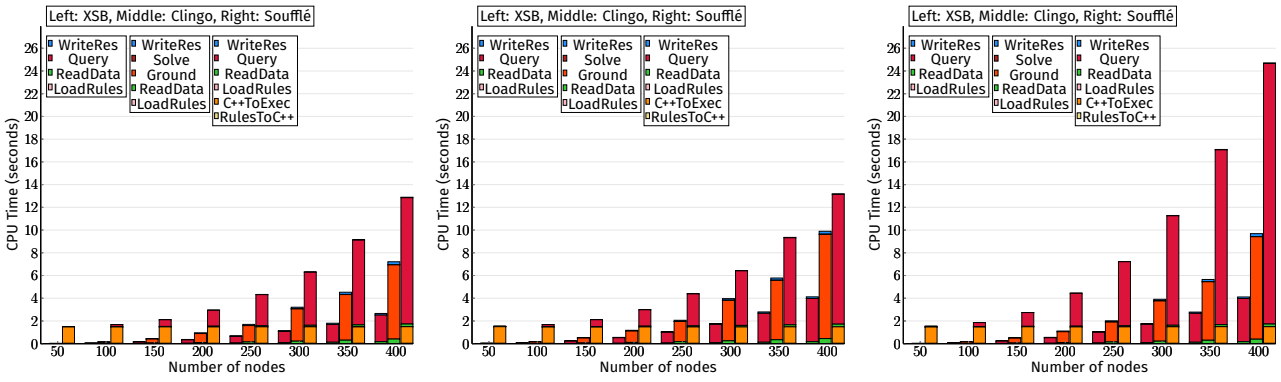


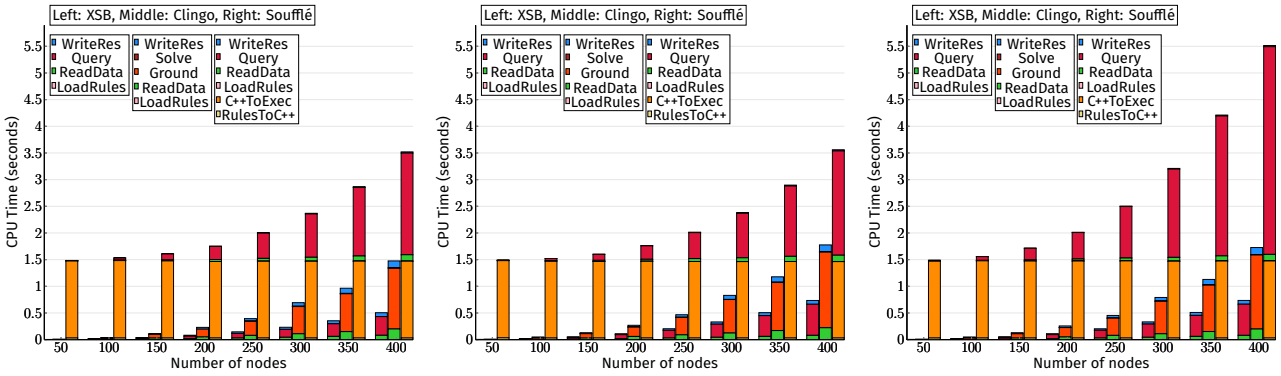
Figure 3.17: Running times for (a) LR, (b) RR, and (c) DR on complete graphs.

Comparison using `max_acyclic` graphs

Shown below is the performance of XSB, Clingo, and Soufflé for each transitive closure variant on `max_acyclic` graphs. The plots are shown in Figure 3.18.

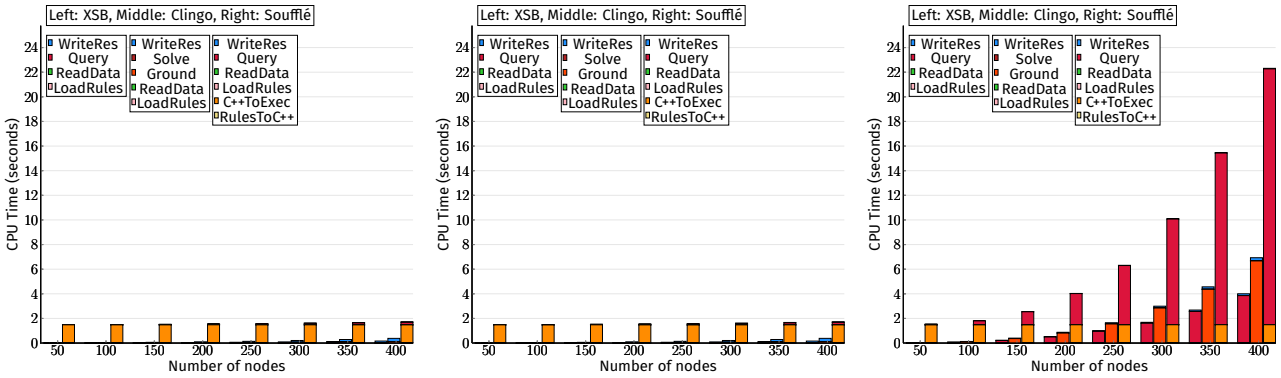
Comparison using `cycle` graphs

The performance of XSB, Clingo, and Soufflé for each transitive closure variant on `cycle` graphs is shown in Figure 3.19. XSB outperforms Clingo and Soufflé across all variants, with Clingo following closely behind. Soufflé lags behind in all comparisons.



(a) (b) (c)

Figure 3.18: Running times for (a) LR, (b) RR, and (c) DR on max_acyclic graphs.

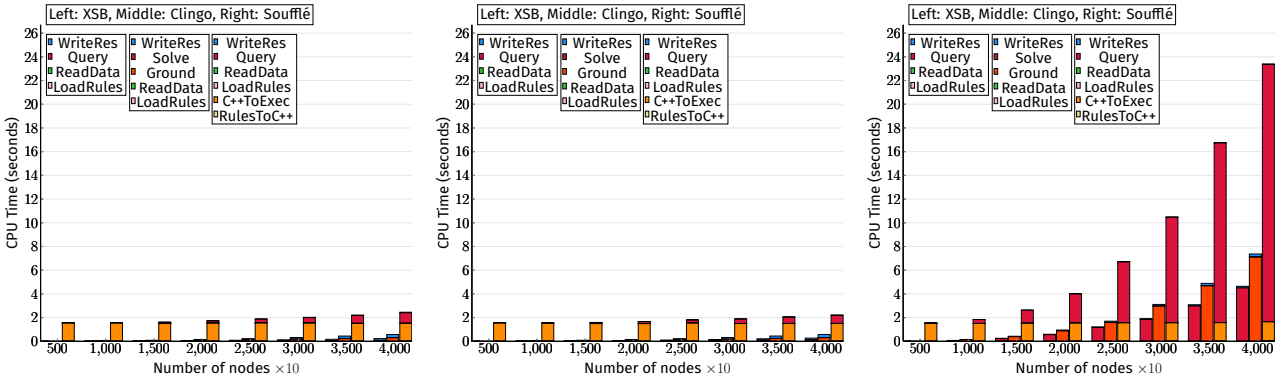


(a) (b) (c)

Figure 3.19: Running times for (a) LR, (b) RR, and (c) DR on cycle graphs.

Comparison using `cycle_with_shortcuts` graphs

Cycle with shortcuts graphs are used to compare the performance of XSB, Clingo, and Soufflé for each transitive closure variant. The results are shown in Figure 3.20.

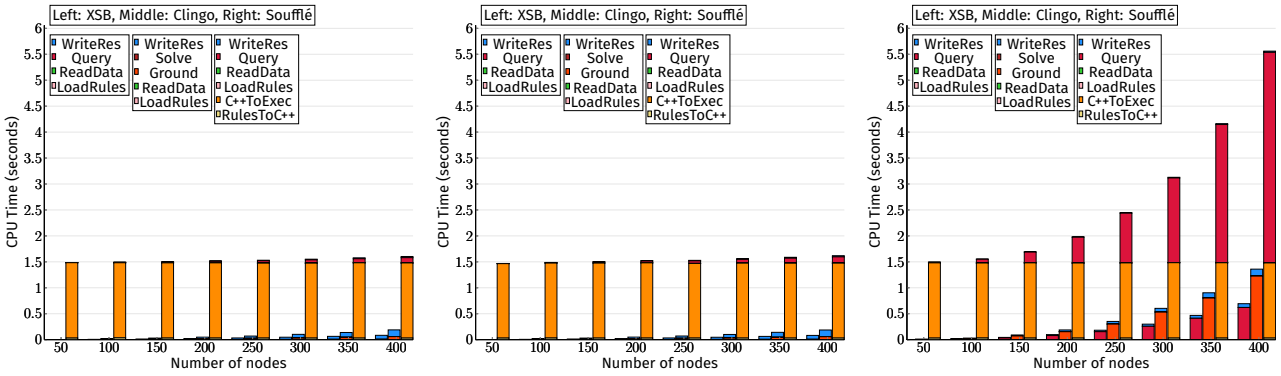


(a) (b) (c)

Figure 3.20: Running times for (a) LR, (b) RR, and (c) DR on cycle_with_shortcuts graphs.

Comparison using `path` graphs

Depicted in Figure 3.21 is the performance of XSB, Clingo, and Soufflé.

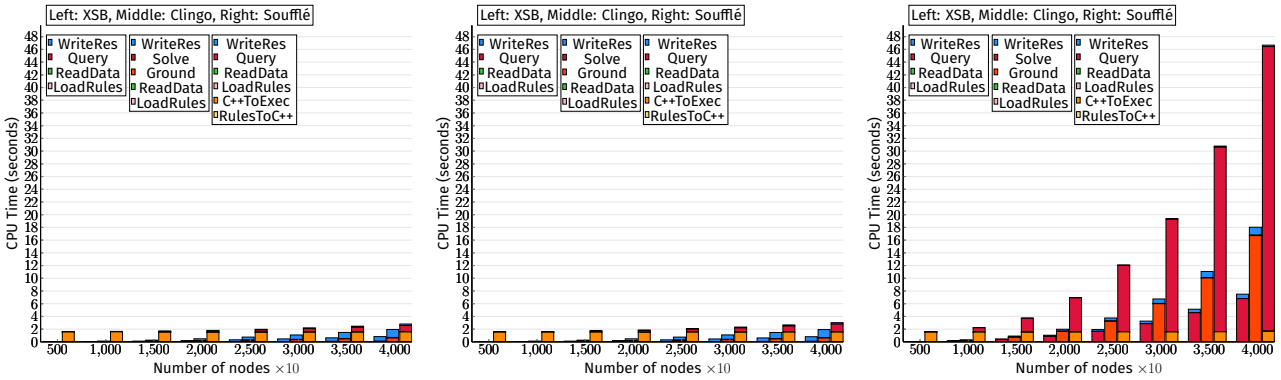


(a) (b) (c)

Figure 3.21: Running times for (a) LR, (b) RR, and (c) DR on path graphs.

Comparison using `multi_path` graphs

For `multi_path` graphs, the performance of XSB, Clingo, and Soufflé for each transitive closure variant is as shown in Figure 3.22. XSB outperforms Clingo and Soufflé across all variants, with Clingo following closely behind. Soufflé lags behind in all comparisons.



(a) (b) (c)

Figure 3.22: Running times for (a) LR, (b) RR, and (c) DR on multi_path graphs.

Comparison using `binary_tree` graphs

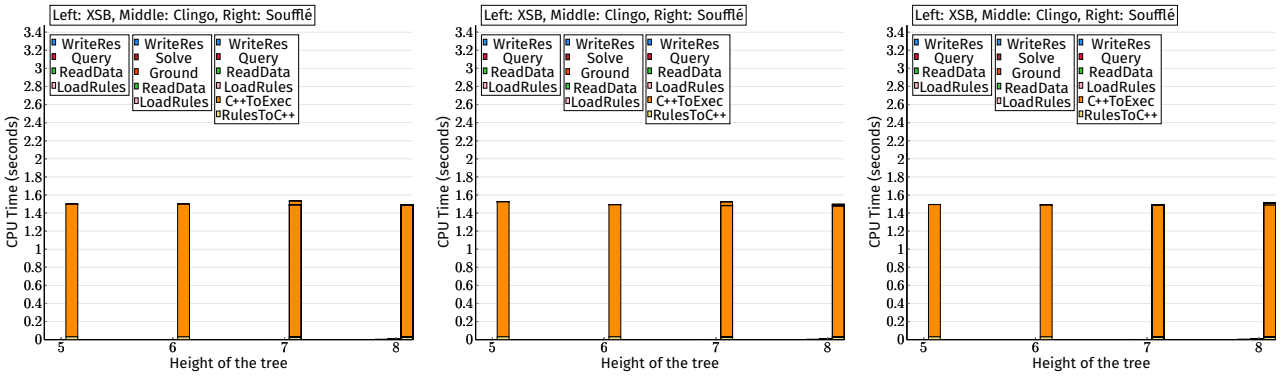
Binary tree graphs are used to compare the performance of XSB, Clingo, and Soufflé for each transitive closure variant. The results are shown in Figure 3.23.

Comparison using `reverse_binary_tree` graphs

For reverse binary tree graphs, the performance of XSB, Clingo, and Soufflé for each transitive closure variant is as shown in Figure 3.24.

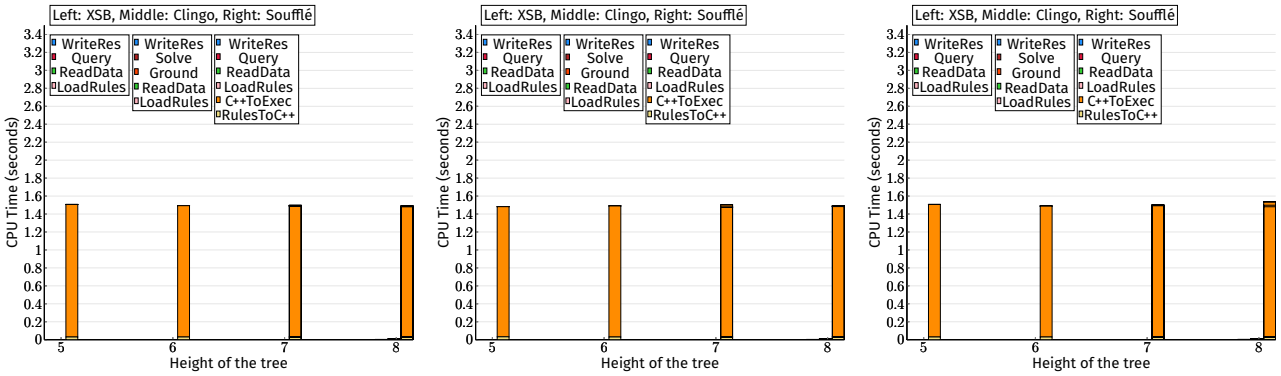
Comparison using y graphs

Figure 3.25 shows the performance of XSB, Clingo, and Soufflé for each transitive closure variant on y graphs.



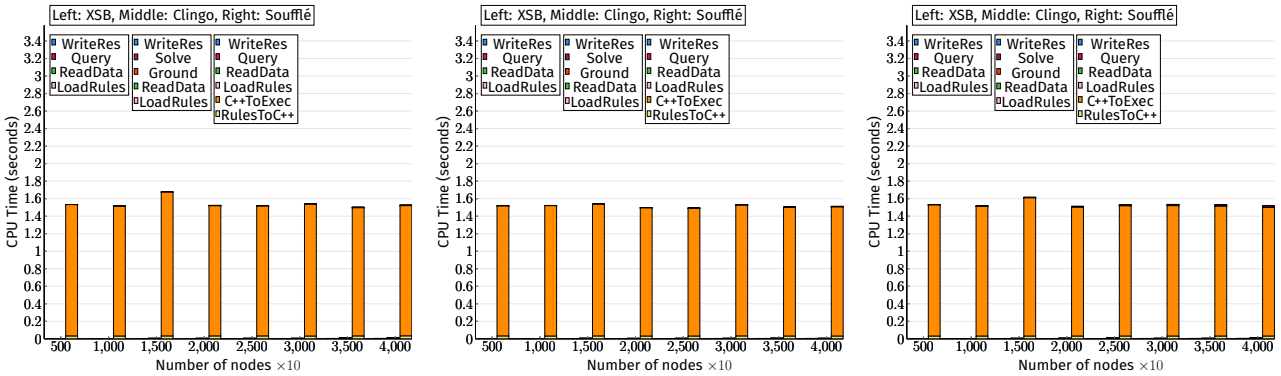
(a) (b) (c)

Figure 3.23: Running times for (a) LR, (b) RR, and (c) DR on binary_tree graphs.



(a) (b) (c)

Figure 3.24: Running times for (a) LR, (b) RR, and (c) DR on reverse_binary_tree graphs.



(a) (b) (c)

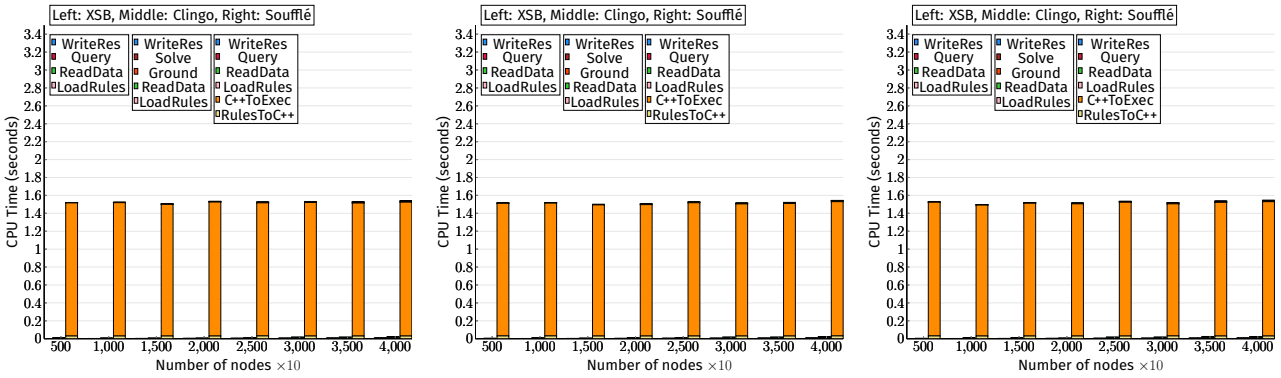
Figure 3.25: Running times for (a) LR, (b) RR, and (c) DR on y graphs.

Comparison using w graphs

Shown in Figure 3.26 is the performance of XSB, Clingo, and Soufflé for each transitive closure variant on w graphs.

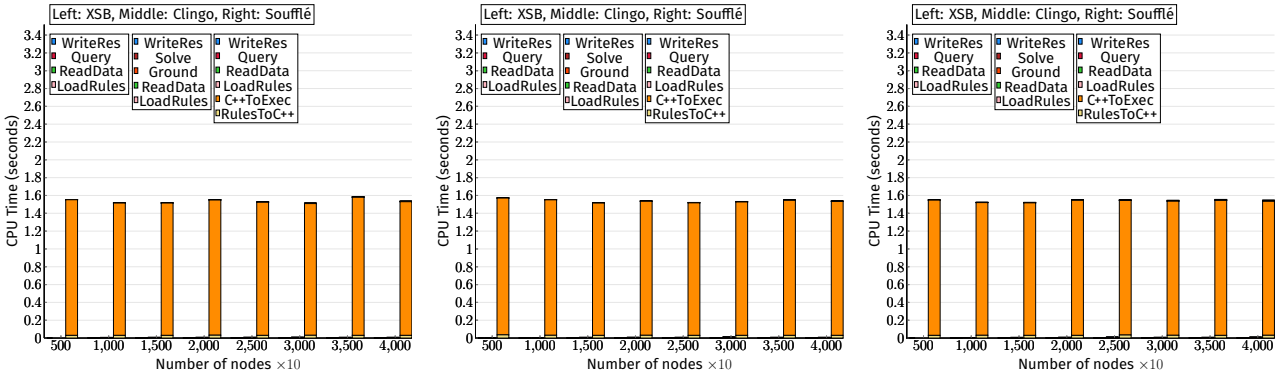
Comparison using x graphs

Figure 3.27 shows the performance of XSB, Clingo, and Soufflé for each transitive closure variant on x graphs.



(a) (b) (c)

Figure 3.26: Running times for (a) LR, (b) RR, and (c) DR on w graphs.

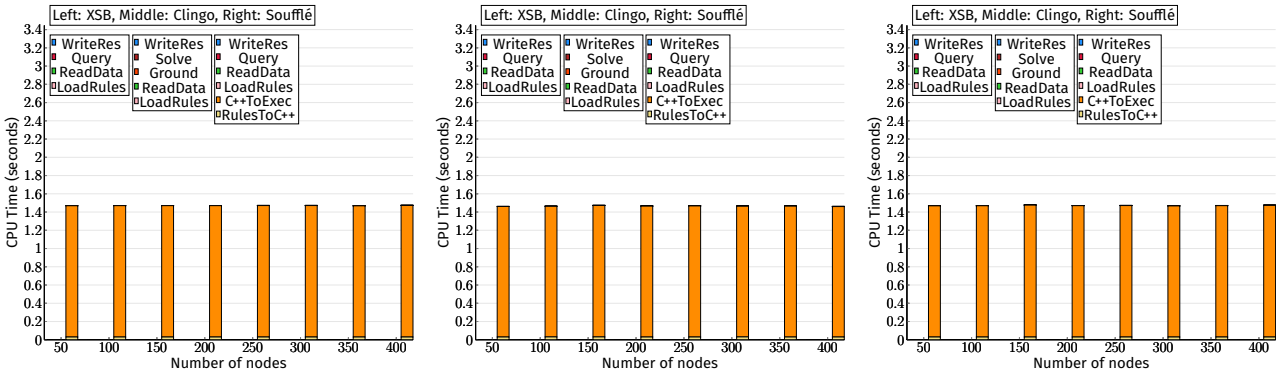


(a) (b) (c)

Figure 3.27: Running times for (a) LR, (b) RR, and (c) DR on x graphs.

Comparison using `star` graphs

For star graphs, the performance of XSB, Clingo, and Soufflé for each transitive closure variant is as shown in Figure 3.28.

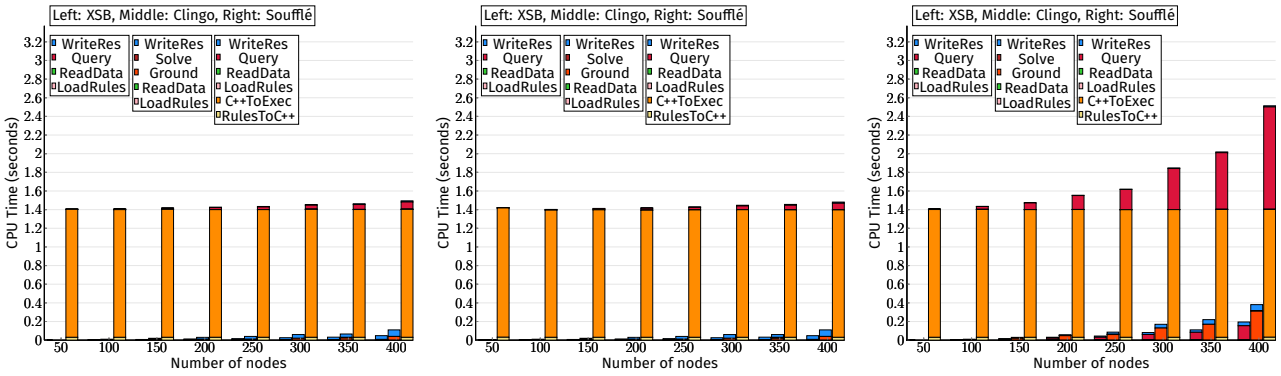


(a) (b) (c)

Figure 3.28: Running times for (a) LR, (b) RR, and (c) DR on star graphs.

Comparison using `grid` graphs

Grid graphs are used to compare the performance of XSB, Clingo, and Soufflé for each transitive closure variant. The results are shown in Figure 3.29.



(a) (b) (c)

Figure 3.29: Running times for (a) LR, (b) RR, and (c) DR on grid graphs.

3.4.6 Potential Reasons for Jagged plots

The jagged patterns in the performance plots can be succinctly explained by several key factors:

Graph Structure and Complexity: Some graphs such as cycles with shortcuts, star graphs and others have varying complexity as they scale, leading to unpredictable performance due to changing connectivity and path complexity.

Memory Management and Caching Effects: Fluctuations in performance can also stem from how memory and caching are managed since rule system's efficiency in handling recursion depth and intermediate results can cause performance peaks and troughs.

Implementation Specifics: The compilation and execution mechanisms of the systems influence performance, especially how compiler optimizations are applied to different Graph Sizes and types.

Measurement Noise: External factors like system load and CPU availability during the testing period can introduce variability in the performance data, leading to jagged plot lines.

3.4.7 Insights and Comparison with state-of-the-art

The results are promising in unravelling the true performance of transitive closure.

Left and Right Recursion (LR & RR): display constant, linear or near-linear time complexities ($O(n)$ or $O(n \log n)$) depending on the shape of the graph, particularly efficient in simpler or sparser graph structures such as binary trees or paths. These strategies optimize stack depth and recursive calls, maintaining relatively consistent and low execution times across increasing Graph Sizes.

Double Recursion(DR): exhibits polynomial or exponential time complexities ($O(n^2)$ or higher) in more complex and densely connected graphs like complete and max acyclic graphs. This increase is due to the redundant calculations inherent in double recursion, which re-evaluate paths multiple times, significantly impacting performance as graph complexity escalates.

Rule System Comparison: Across XSB, Clingo, and Soufflé, XSB consistently outperforms the others in most scenarios, with Clingo following closely behind. Soufflé tends to lag behind in most comparisons, particularly in more complex graph types and sizes. XSB's performance is generally superior due to its optimized handling of recursion and intermediate results using tabling, while Clingo's performance is commendable, maintaining competitive margins with XSB.

In all, the Left Recursion (LR) tends to be a better choice for computations with linear to slightly super-linear performances depending on the graph shapes and sizes and the rule system used.

3.5 Future Work

The current automated implementation offers some exploration into evaluating rule system performances. Future enhancements will include:

- Generalizing the benchmarking suite to include more rule systems, transitive closure variants, graph types, queries, and other problem domains.
- Incorporating more complex graph structures and queries to test the rule systems' performance in handling more intricate and real-world scenarios.
- Better modularization and abstraction to allow for easy integration of new rule systems and graph types.

Bibliography

[1] Rakesh Agrawal, Shaul Dar, and H. V. Jagadish. Direct transitive closure algorithms: design and performance evaluation. *ACM Trans. Database Syst.*, 15(3):427–458, 1990.

[2] Stefan Brass and Mario Wenzel. Performance analysis and comparison of deductive systems and SQL databases. In Mario Alviano and Andreas Pieris, editors, *Datalog 2.0 2019 - 3rd International Workshop on the Resurgence of Datalog in Academia and Industry co-located with the 15th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2019) at the Philadelphia Logic Week 2019, Philadelphia, PA (USA), June 4-5, 2019*, volume 2368 of *CEUR Workshop Proceedings*, pages 27–38. CEUR-WS.org, 2019.

[3] Filippo Cacace, Stefano Ceri, and Maurice Houtsma. A survey of parallel execution strategies for transitive closure and logic programs. *Distributed and Parallel Databases*, 1:337–382, 1993.

[4] Shaul Dar and Raghu Ramakrishnan. A performance study of transitive closure algorithms. *SIGMOD Rec.*, 23(2):454–465, 1994.

[5] M. Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = asp + control: Preliminary report. *ArXiv*, abs/1405.3694, 2014.

[6] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Multi-shot asp solving with clingo. *Theory and Practice of Logic Programming*, 19(1):27–82, 2019.

[7] John Owolabi Idogun. Performance analysis of tc. https://github.com/Sirneij/trans-bench/tree/update/, April 2024.

[8] Federico Sevilla III. Set of unique pseudo-random tuples generator. https://drive.google.com/drive/u/1/folders/16U4FxWII2yNrcKvkFjSBM9pEIYk7-lwN, march 2014.

[9] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[10] Kjell Lemström and Lauri Hella. Approximate pattern matching and transitive closure logics. *Theoretical computer science*, 299(1-3):387–412, 2003.

[11] Senlin Liang, Paul Fodor, Hui Wan, and Michael Kifer. Openrulebench: an analysis of the performance of rule engines. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, page 601–610, New York, NY, USA, 2009. Association for Computing Machinery.

[12] Yanhong A. Liu, Scott D. Stoller, Yi Tong, and Bo Lin. Integrating logic rules with everything else, seamlessly. *Theory and Practice of Logic Programming*, 23(4):678–695, 2023.

[13] Yanhong A. Liu, Scott D. Stoller, Yi Tong, and K. Tuncay Tekle. Benchmarking for integrating logic rules with everything else. *Electronic Proceedings in Theoretical Computer Science*, 385:12–26, Sep 2023.

[14] Yanhong Annie Liu. Personal email communication, 2024. Personal email communication on May 14, 2024.

[15] Potassco, the Potsdam Answer Set Solving Collection. Clingo api documentation. https://potassco.org/clingo/python-api/5.4/, 2024.

[16] Python Software Foundation. argparse documentation. https://docs.python.org/3/library/argparse.html. Last accessed 08 April 2024. Python 3.9.6.

[17] Python Software Foundation. logging documentation. https://docs.python.org/3/library/logging.html. Last accessed 08 April 2024. Python 3.9.6.

[18] Python Software Foundation. math documentation. https://docs.python.org/3/library/math.html. Last accessed 08 April 2024. Python 3.9.6.

[19] Python Software Foundation. os documentation. https://docs.python.org/3/library/os.html. Last accessed 09 April 2024. Python 3.9.6.

[20] Python Software Foundation. pathlib documentation. https://docs.python.org/3/library/pathlib.html. Last accessed 08 April 2024. Python 3.9.6.

[21] Python Software Foundation. pickle documentation. https://docs.python.org/3/library/pickle.html. Last accessed 08 April 2024. Python 3.9.6.

[22] Python Software Foundation. subprocess documentation. https://docs.python.org/3/library/subprocess.html. Last accessed 08 April 2024. Python 3.9.6.

[23] Konstantinos Sagonas, Terrance Swift, and David S. Warren. Xsb as an efficient deductive database engine. *SIGMOD Rec.*, 23(2):442–453, may 1994.

[24] Souffle Contributors. Souffle c++ interface. https://souffle-lang.github.io/interface, 2024.

[25] Yi Tong. rule to rules; constraint to constraints; python3 to python; and other minor changes. https://github.com/DistAlgo/alda/blob/f924f3897a0d870f05d2edf19033d511c0529b97/benchmarks/rules/trans/launcher.da, 2023.

[26] Yi Tong. rule to rules; constraint to constraints; python3 to python; and other minor changes. https://github.com/DistAlgo/alda/blob/master/da/rules/xsb/extfilequery.P, 2023.

Appendices

Modifications from Part II Submission

Following the feedback received after the Part II submission, several adjustments were made to enhance the clarity, depth, and comprehensiveness of the project documentation. Here are the key areas where modifications were implemented:

A.1 Input and Output Specifications

A.1.1 Input

The description of the input has been refined to explicitly list the rule systems employed, the specific rules applied, and the types of data shapes considered.

A.1.2 Output

The output description has been revised for greater precision and conciseness.

A.2 Enhanced State of the Art and Use Cases

A.2.1 State of the Art

The section on state-of-the-art has been expanded to include a more detailed review of relevant literature. This enhancement not only cites additional sources that directly or indirectly contribute to the project's foundation but also situates the project within the broader context of current research and developments.

A.2.2 Example Use Cases

New examples illustrating practical applications of Transitive Closure in various domains have been added. These examples demonstrate the utility and versatility of the implemented system, showcasing its applicability to real-world problems and scenarios.

A.3 Inclusion of Additional References

References to seminal and recent works related to the rule systems and tools utilized in the project have been incorporated.

A.4 Project Design and Documentation Integration

A.4.1 Unified Design Document

The design and proposal documents have been merged into a single cohesive document.

A.4.2 Detailed Design Overview

An in-depth discussion of the project's design has been introduced, including the architectural diagram and decisions, components, and the flow of operations. This section elucidates the design rationale, detailing how each component contributes to the system's overall functionality and how they interact within the system.

A.4.3 README updated

The README on the project now explains what each file and folder contain and do.