Knowledge Engineering: Homework 3

Yifei Zuo
PB20061254

Yunqin Zhu
PB20061372

1 Rule-based Logics

1.1 Follow the "curiosity kill the cat" example given in the slides, show how to use forward chaining, backward chaining and resolution to solve this problem.

Solution

- **Problem setting:**
  - **Knowledge base:**
    (1) $\exists x, \text{Dog}(x) \land \text{Owns}(\text{Jack}, x)$
    (2) $\forall x, (\exists y, \text{Dog}(y) \land \text{Owns}(x, y)) \rightarrow \text{AnimalLover}(x)$
    (3) $\forall x, \text{AnimalLover}(x) \rightarrow (\forall y, \text{Animal}(y) \rightarrow \neg\text{Kills}(x, y))$
    (4) $\text{Kills}(\text{Jack}, \text{Tuna}) \lor \text{Kills}(\text{Curiosity}, \text{Tuna})$
    (5) $\text{Cat}(\text{Tuna})$
    (6) $\forall x, \text{Cat}(x) \rightarrow \text{Animal}(x)$
  - **Horn clauses:**
    (1) $\text{Dog}(D)$
    (2) $\text{Owns}(\text{Jack}, D)$
    (3) $\text{Dog}(y_1) \land \text{Owns}(x_1, y_1) \rightarrow \text{AnimalLover}(x_1)$
    (4) $\text{AnimalLover}(x_2) \land \text{Animal}(y_2) \rightarrow \neg\text{Kills}(x_2, y_2)$
    (5) $\neg\text{Kills}(\text{Jack}, \text{Tuna}) \rightarrow \text{Kills}(\text{Curiosity}, \text{Tuna})$
    (6) $\text{Cat}(\text{Tuna})$
    (7) $\text{Cat}(x_3) \rightarrow \text{Animal}(x_3)$
  - **Goal:** $\text{Kills}(\text{Curiosity}, \text{Tuna})$
- **Forward chaining:**

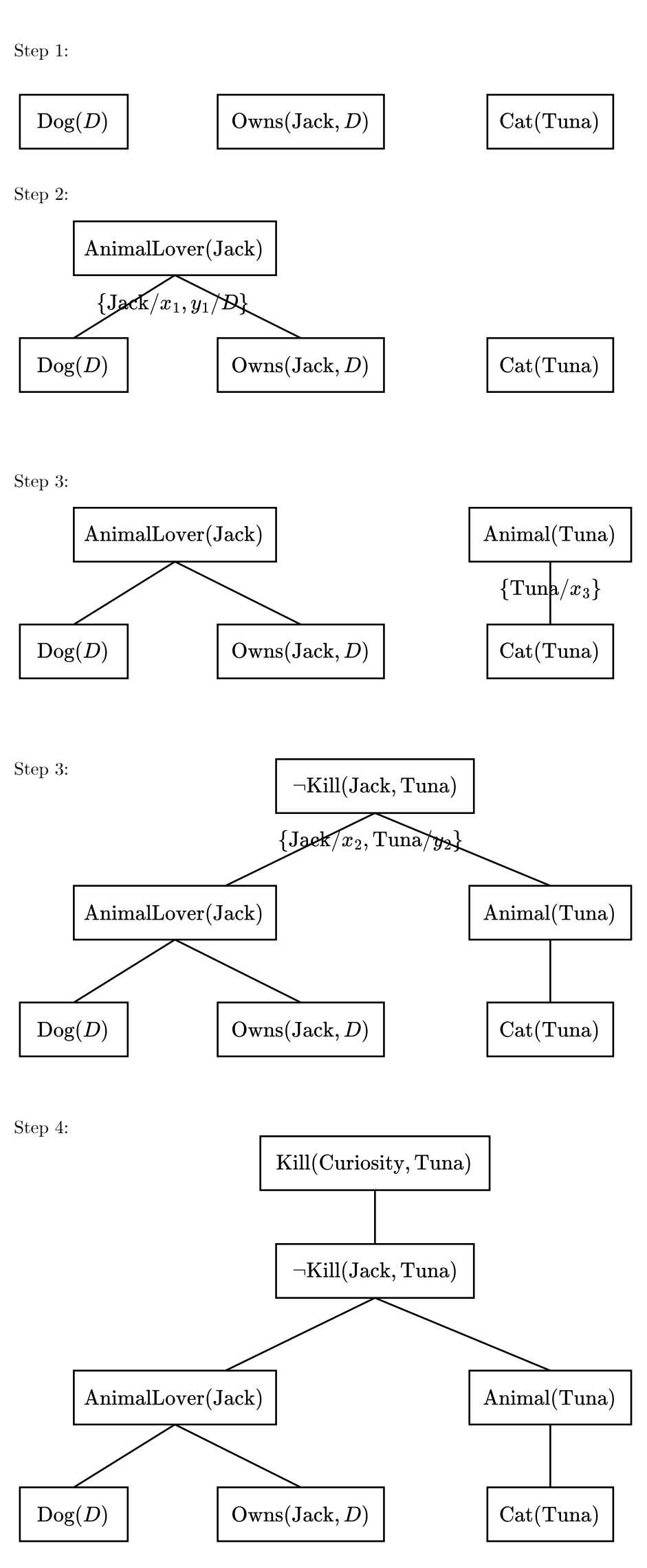

Figure 1. Solving the problem using forward chaining.

- **Backward chaining:**

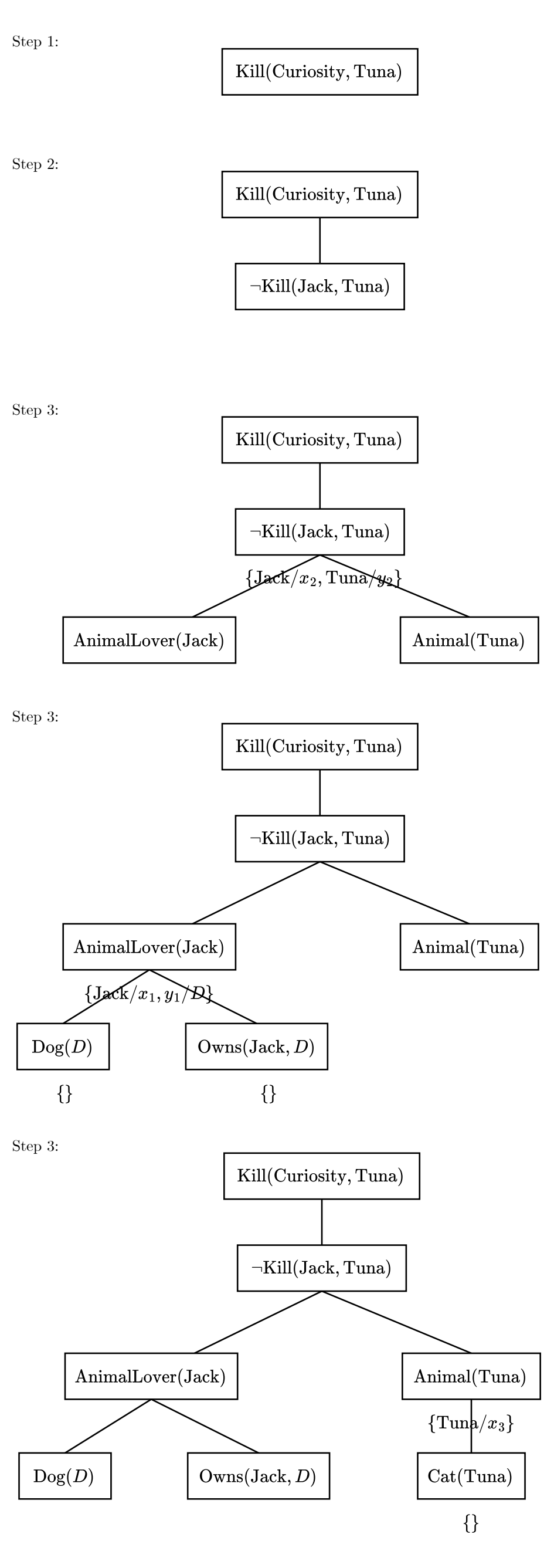

Figure 2. Solving the problem using backward chaining.

- **Resolution:**

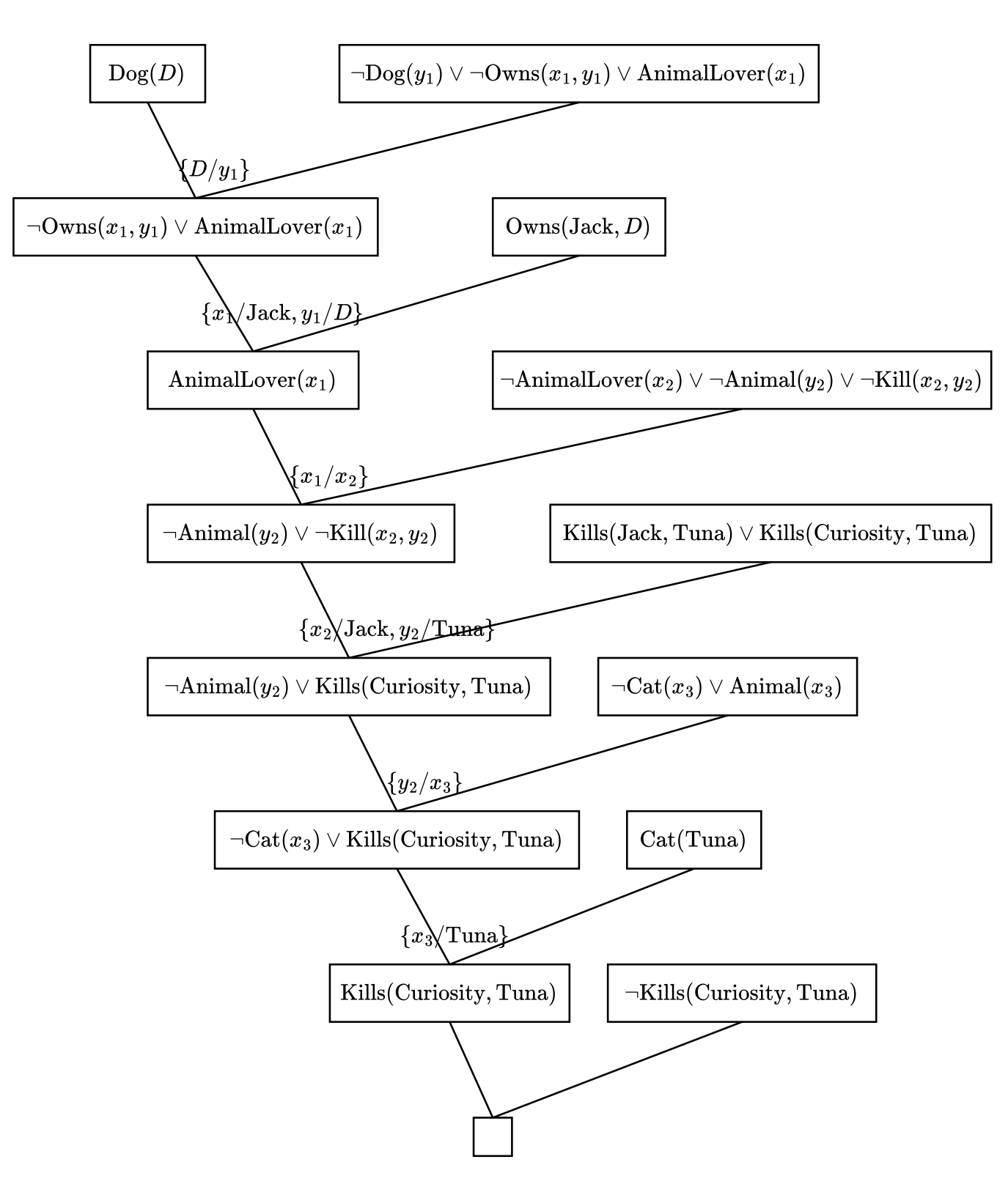

Figure 3. Solving the problem using resolution.

1.2 Explain why transitive closure cannot be formalized in classical first-order logic.

Solution

Given a binary predicate $\text{Edge}(\cdot,\cdot)$, we define its transitive closure $\text{Path}(\cdot,\cdot)$ as the minimal transitive predicate containing $\text{Edge}(\cdot,\cdot)$. In second-order logic, it can be formalized as

$$\text{Path}(x,y) \leftarrow (\forall T, (\forall a, b, \text{Edge}(a,b) \rightarrow T(a,b)) \land (\forall a,b,c, T(a,b) \land T(b,c) \rightarrow T(a,c)) \rightarrow T(x,y))$$

In first-order logic, $\text{Path}(x,y)$ can be defined via the following recursive formula:

$$\text{Path}(x,y) \leftarrow \text{Edge}(x,y) \lor \text{Edge}(x,z) \land \text{Path}(z,y) \tag{1}$$

which expands to:

$$\text{Path}(x,y) \leftarrow \text{Edge}(x,y) \lor \bigvee_{i=1}^{\infty} (\text{Edge}(x,z_1) \land \cdots \land \text{Edge}(z_{i-1},z_i) \land \text{Edge}(z_i,y)) \tag{2}$$

We can prove its minimality by induction, which is omitted here. Both Formula 1 and Formula 2 involve the use of infinitary logic owing to infinite long formula or proof, which is beyond the scope of classical first-order logic.

In a model-theoretic sense, we cannot write a formula in classical first-order logic using predicate symbols $\text{Edge}(\cdot,\cdot)$ and $\text{Path}(\cdot,\cdot)$ that will be satisfied in any model if and only if $\text{Path}(\cdot,\cdot)$ is the transitive closure of $\text{Edge}(\cdot,\cdot)$ (Wikipedia 2023). It is necessary to extend the language with transitive closure operator $\text{TC}(\cdot)$, or more general least fixed point operator $\text{LFP}(\cdot)$, to formalize transitive closure.

1.3 Compute the well-founded model, the classical model and the stable model of the following program:

$$p \leftarrow \textbf{not } q$$
$$q \leftarrow \textbf{not } p$$

Solution

- **Well-founded model:**
  (1) Make inferences: Nothing to do.
  (2) Find the largest unbounded set: $\varnothing$.
  (3) Stop. The partial model is $\varnothing$, i.e. both p and q are unknown.
- **Classical model:**
  (1) Compute the completion of the program:

$$p \leftrightarrow \neg q$$
$$q \leftrightarrow \neg p$$

  (2) Find the model of the completion: $\{p, \neg q\}$ or $\{\neg p, q\}$.
- **Stable model:**

Table 1. Find the stable models of the program.

Instantiation	Reduct of the program	Minimum model of the reduct
$\{\neg p, \neg q\}$	$p \leftarrow \quad q \leftarrow$	$\{p, q\}$
$\{p, \neg q\}$	$p \leftarrow$	$\{p, \neg q\}$
$\{\neg p, q\}$	$q \leftarrow$	$\{\neg p, q\}$
$\{p, q\}$		$\{\neg p, \neg q\}$

The program has two stable models: $\{p, \neg q\}$ and $\{\neg p, q\}$.

1.4 Again, pick up one of the application areas mentioned in the slides. Go deeper and deeper to explain how rule-based logics are used in these areas.

Solution

- **Area:** Deductive Database
- **Definition:**
  A deductive database is a database system that can infer additional facts based on facts stored in the extensional database (EDB) and rules stored in the intensional database (IDB). By combining logic programming with relational databases, deductive databases achieve a trade-off between expressiveness and scalability and are used in many applications, such as data integration, information extraction, static program analysis, etc. (Wikipedia 2022)
- **How rule-based logics are used:**
  Deductive database query languages are mostly related to Prolog. Among them, Datalog has become the de-facto standard language for specifying facts, rules and queries in deductive databases.
  - **Datalog:**
    Datalog is a declarative logic programming language, syntactically a subset of Prolog. A Datalog program consists of a list of rules, i.e. Horn clauses, with no function symbols. The following BNF grammar expresses the structure of a Datalog program (Wikipedia 2023):

```
<program> ::= <rule> <program> | ""
<rule> ::= <atom> ":-" <atom-list> "."
<atom> ::= <relation> "(" <term-list> ")"
<atom-list> ::= <atom> | <atom> "," <atom-list> | ""
<term> ::= <constant> | <variable>
<term-list> ::= <term> | <term> "," <term-list> | ""
```

    There are three equivalent approaches to the semantics of Datalog programs:
    (1) **Model-theoretic:** Define the minimal Herbrand model to be the meaning of the program.
    (2) **Proof-theoretic:** Define the set of facts with corresponding proof trees to be the meaning of the program. This semantics suggest *SLD resolution*, which is the top-down evaluation algorithm used in Prolog.
    (3) **Fixed-point:** Define the least fixed point of T to be the meaning of the program, where T is the immediate consequence operator of the program. This semantics suggest *naïve evaluation*, which is one of the bottom-up evaluation algorithms used in Datalog.

    Unlike Prolog, where the goal is matched against the rules to find a proof tree, Datalog program computes the entire minimal model by applying rules to a set of known facts iteratively until a fixed point is reached. Actually, both methods can be used to handle deductive database queries. The top-down method uses goal-driven *backward chaining* inference, while the bottom-up method uses data-driven *forward chaining* inference. Another branch of algorithms, namely *Magic Sets*, optimize a program into a more efficient one while still using bottom-up evaluation.

    The decision problem for Datalog is P-COMPLETE w.r.t. data complexity, and EXPTIME-COMPLETE w.r.t. program complexity. In particular, evaluating Datalog programs always terminates.

    However, Datalog is quite limited in its expressivity. It is not TURING-COMPLETE. It does not support negation, aggregation, arithmetic operators or other modern features. Therefore, real-world deductive database languages are usually based on extensions of Datalog. Some of these extensions are listed in the next section.
  - **Datalog-like languages:**
    (1) **Datalog$_\neg$:** A variant of Datalog that allows negation in rules but requires stratification to avoid infinite recursion.
    (2) **Datalog$_\mathbb{Z}$:** An variant of Datalog that allows integer arithmetic and comparisons.
    (3) **DLV:** A variant of Datalog that allows disjunctions in the heads of rules, applied in the context of reasoning about ontologies in the semantic web.
    (4) **LogiQL:** (Aref et al. 2015) A commercial variant of Datalog with negation, aggregation, static typing, etc., implemented in a deductive database LogicBlox, used in various application areas such as supply chain management, financial services, and healthcare.
    (5) **Soufflé:** (Scholz et al. 2016) An open-source variant of Datalog with negation, aggregation, static typing, etc., including an interpreter and a compiler that targets parallel C++, typically used to build static analyzers.

    Most modern Datalog engines, including BigDatalog, Soufflé, SociaLite and RecStep, use *semi-naïve evaluation* as the core evaluation method, which improves upon naïve evaluation by taking care to use only newly discovered facts in subsequent iterations. LogicBlox also used semi-naïve evaluation in its earlier versions, but then switched to other methods based on incremental view maintenance. (Ketsman & Koutris 2022)

    Here are some important features of modern Datalog extensions:
    (1) **Stratified negation:**
    The core fragment of Datalog cannot express non-monotone properties. To tackle this problem, we introduce *Datalog with negation*, which allows rules to have, besides positive atoms, also negated atoms in the body of rules. The use of negation has a safety restriction, which means that each variable in the body of a rule must occur in at least one positive atom.
    A Datalog program with negation is *stratified* if the rules of the program can be divided in disjoint strata $S_1, S_2, \ldots, S_n$ of rules such that the following properties are true for each IDB predicate P:
    (a) All rules with a head P belong to the same stratum S_i;
    (b) All rules with P in the body belong to some stratum S_j, where $j > i$;
    (c) All rules with **not** P in the body belong to some stratum S_j, where $j \geq i$;
    We note that stratification is not always possible. The output of a stratified Datalog programs over a given database is defined as the result of evaluating the different strata of the program one-by-one, while taking as input the result of the previous stratum.
    (2) **Stratified aggregation:**
    In order to support any type of data analytics, it is necessary to extend Datalog to support some form of aggregation. A rule with stratified aggregation has the following form.

$$P(x_1, \ldots, x_n, F(y)) \leftarrow Q(x_1, \ldots, x_n, y).$$

    where F is an aggregate function, for instance min, max, sum, count. Unfortunately, Datalog with only stratified aggregation is not sufficiently expressive to solve a large class of recursive problems, such as shortest paths. Some early work proposed to use monotonic aggregate functions to support aggregation inside recursion, but there is no widely accepted standard on how recursive aggregation can be supported in a practical system.
  - **Between Datalog and SQL:** (Wikipedia 2023)
    The non-recursive subset of Datalog is closely related to query languages for relational databases, such as SQL.
    On the one hand, each non-recursive Datalog rule corresponds precisely to a conjunctive query. Therefore, many of the techniques from database theory used to speed up conjunctive queries are applicable to bottom-up evaluation of Datalog, such as index selection and data structures, e.g. hash table and B-tree. Many such techniques are implemented in modern bottom-up Datalog engines such as Soufflé. Some Datalog engines integrate SQL databases directly.
    On the other hand, some widely used relational database systems include ideas and algorithms developed for Datalog. For example, the SQL:1999 standard includes recursive queries. Many modern relational databases, e.g. Db2, also implement the aforementioned Magic Sets algorithm to optimize complex queries.
  - **Example:**
    The following example is adapted from Sáenz-Pérez 2011.
    (1) **Datalog version:**

```
% Program:
path(X, Y, 1) :- edge(X, Y).
path(X, Y, L) :- path(X, Z, L0), edge(Z, Y),
                 count(edge(A, B), Max), L0 < Max, L is L0 + 1.
% Query:
shortest_paths(X, Y, L) :- min(path(X, Y, Z), Z, L).
```

    (2) **SQL version:**

```
--View:
CREATE VIEW shortest_paths(src, dest, len) AS (
  WITH RECURSIVE path(src, dest, len) AS (
    SELECT *, 1 FROM edge
    UNION
    SELECT path.src, edge.dest, path.len + 1 FROM path, edge
    WHERE path.dest = edge.src AND path.len < (SELECT MAX(len) FROM path)
  )
  SELECT src, dest, MIN(len) FROM path
  GROUP BY src, dest
);
--Query:
SELECT * FROM shortest_paths;
```

    The program calculates the shortest paths between all pairs of nodes in a graph. While the Datalog version is more concise, the SQL version achieves the same functionality by declaring a recursive view to compute the transitive closure of the graph.

1.5 Go deeper to discuss the similarities and differences among rule based logic.

Solution

- **Similarities:**
  (1) Both production rules and the other logics are based on rules and entailment.
  (2) All systems use variables and unification for matching patterns.
  (3) They all follow a well-defined syntax.
- **Differences:**
  Here we present brief summarization on each of the four typical rule based logic and list their differences with each other.
  - **Production Rules:**
    Production rules, also known as production systems or rule-based systems, are a form of rule-based logic that consist of a set of rules and an execution mechanism. Each rule is composed of a condition (the "if" part) and an action (the "then" part). When the condition is satisfied, the action is executed.
    (1) Production rules are focused on the *execution of actions*, while the other logics are more focused on reasoning and deduction.
    (2) Production rules use an explicit control strategy, typically *forward or backward chaining*, whereas the other logics rely on a logical inference mechanism.
    (3) Production rules are usually used for problem-solving, while the other logics are more suited to knowledge representation and querying.
  - **Horn Logic:**
    A subset of first-order logic that restricts the form of its rules to make them more computationally efficient. The rules in Horn logic are implications, with a single positive predicate in the head and a conjunction of literals (both positive and negative) in the body. Horn logic is the basis for Prolog.
    (1) Horn logic *allows for both positive and negative literals in the body*, while production rules and Datalog typically only allow positive literals.
    (2) Horn logic uses *resolution or forward chaining as an inference mechanism*, while production rules use a control strategy and Datalog uses a bottom-up evaluation strategy.
    (3) Horn logic is more expressive than production rules and Datalog but less expressive than Answer Set Programming.
  - **Datalog:**
    A declarative logic programming language that is a subset of Horn logic. It is specifically designed for querying and manipulating relational databases. Datalog uses recursion and stratified negation for expressing complex queries, and it is typically more efficient than general-purpose logic programming languages like Prolog.
    (1) Datalog is specifically designed for *querying and manipulating relational databases*, while the other logics have broader applications.
    (2) Datalog uses a *bottom-up evaluation* strategy, while production rules use a control strategy and Horn logic uses resolution or forward chaining.
    (3) Datalog has limited expressivity compared to Horn logic and Answer Set Programming.
  - **Answer Set Programming (ASP):**
    A declarative programming paradigm that is an extension of logic programming. ASP allows for the representation of incomplete and inconsistent information, which is useful for solving complex problems. ASP is based on the stable model semantics, which defines the intended models of a program as its "answer sets."
    (1) ASP is more expressive than the other logics, allowing for the *representation of incomplete and inconsistent information.*
    (2) ASP uses the *stable model semantics for reasoning*, while production rules use a control strategy, Horn logic uses resolution or forward chaining, and Datalog uses a bottom-up evaluation strategy.
    (3) ASP can handle disjunctions in the head of the rule and non-stratified negation, which makes it more powerful than the other logics in terms of expressivity.
    (4) ASP is particularly suited for combinatorial search problems, knowledge representation, and reasoning with incomplete information, whereas the other logics have different focuses and use cases.

1.6 Go deeper to discuss the similarities and differences between rule based logics and classical logics.

Solution

- **Classical logics** such as propositional logic and first-order logic, are formal systems that employ a set of symbols, syntactic rules, and axioms to represent and reason about statements. They typically use a model-theoretic semantics to determine the truth values of statements and employ inference rules like Modus Ponens to derive new statements from existing ones.
- **Rule-based logics**, such as production rules, Horn logic, Datalog, and Answer Set Programming, are formal systems that rely on rules and entailment for knowledge representation and reasoning. They use various reasoning mechanisms and semantics, making them well-suited for different tasks and applications
- **Similarities:**
  (1) Both classical logics and rule-based logics are formal systems for knowledge representation and reasoning.
  (2) Both types of logic use a well-defined syntax and semantics.
  (3) Both types of logic rely on inference rules or mechanisms to derive new conclusions.
  (4) Both are lacking in learning process.
- **Differences:**
  (1) Classical logics use a fixed set of logical connectives (e.g., and, or, not, implies), while rule-based logics use rules with a specific structure (e.g., condition-action, head-body).
  (2) Classical logics generally focus on the truth values of statements, while rule-based logics focus on the application of rules and the generation of new facts or actions.
  (3) Classical logics rely on a model-theoretic semantics / truth table, whereas rule-based logics use various semantics, such as operational semantics (production rules), resolution or forward chaining (Horn logic), bottom-up evaluation (Datalog), or stable model semantics (ASP).
  (4) Classical logics often have a broader scope of expressivity, while rule-based logics can be more specialized and computationally efficient for specific tasks.

1.7 Evaluate production system from 6E.

Solution

6E consist of following 6 aspects: Elegant, Extensible, Expressive, Efficient, Educable and Evolvable:

(1) **Elegant:**
Production systems can be considered elegant due to their simplicity and clarity in representing knowledge. Rules are written in an "if-then" format, making them easy to understand and interpret. It is close to how humans articulate knowledge and easy to get rules out of a client / user. This simplicity also makes it easier to trace the reasoning process and debug the system, as the flow of control is clear through the application of rules.

(2) **Extensible:**
Production systems are extensible, as new rules can be added without altering the existing rules. This allows for easy integration of new knowledge into the system, and the modularity of rules makes it simple to update, remove, or modify specific rules as needed.

(3) **Expressive:**
Production systems are expressive enough in areas where the problem-solving process can be naturally represented as a series of condition-action pairs. And few rules are needed for some problems. However, compared to more powerful logics like first-order logic, their expressivity may be limited, as they typically only allow positive literals in the conditions and may struggle to represent more complex relationships. It also lack the ability to handle uncertainty, making it hard to represent ambiguous concept and probabilistic actions.

(4) **Efficient:**
Production systems can be efficient for certain types of problems, especially when appropriate rule-firing strategies, such as forward or backward chaining, are employed. However, they can also be inefficient if there is a large number of rules, and a significant amount of time is spent searching for applicable rules. Optimizations, such as indexing rules or using conflict resolution strategies (e.g., the Rete algorithm), can help improve efficiency. It is also highly non-paralleled, only one rule at a time fires and results can depend on which rule fires first.

(5) **Educable:**
Production systems are considered educable because they are relatively easy for domain experts to understand and work with, even without extensive training in formal logic or programming. The "if-then" structure of the rules closely resembles natural language, making it easier for experts to write, review, and update the rules. Additionally, the simplicity of the rule format allows for easier knowledge acquisition and the potential for machine learning techniques to help refine and improve the rule set.

(6) **Evolvable:**
Production systems can be considered evolvable, as they can adapt to changing requirements or new knowledge through the addition, removal, or modification of rules. However, their evolution may be limited by their expressivity and the potential for rule interactions to become complex or difficult to manage as the system grows. It is also hard to get the rules in the first place, making it not suitable for elucidating the rules.

References

Wikipedia contributors. (2023). Transitive closure. In *Wikipedia, The Free Encyclopedia.* Retrieved 04:07, April 17, 2023, from https://en.wikipedia.org/w/index.php?title=Transitive_closure&oldid=1135887831

Wikipedia contributors. (2022). Deductive database. In *Wikipedia, The Free Encyclopedia.* Retrieved 15:24, April 17, 2023, from https://en.wikipedia.org/w/index.php?title=Deductive_database&oldid=1111344828

Wikipedia contributors. (2023). Datalog. In *Wikipedia, The Free Encyclopedia.* Retrieved 05:12, April 18, 2023, from https://en.wikipedia.org/w/index.php?title=Datalog&oldid=1147027344

Ketsman, B., & Koutris, P. (2022). Modern Datalog Engines. *Foundations and Trends in Databases*, 12(1), 1-68.

Aref, M., ten Cate, B., Green, T. J., Kimelfeld, B., Olteanu, D., Pasalic, E., ... & Washburn, G. (2015). Design and implementation of the LogicBlox system. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (pp. 1371-1382).

Scholz, B., Jordan, H., Subotić, P., & Westmann, T. (2016). On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (pp. 196-206).

Sáenz-Pérez, F. (2011). DES: A deductive database system. *Electronic Notes in Theoretical Computer Science*, 271, 63-78.