

CS 6364 Artificial Intelligence
Spring 2024

Assignment 3: Knowledge Representation and Reasoning
Part I: Due electronically by Friday, March 29, 11:59 p.m.

Part I: Programming (100 points)

In this problem you will be implementing a theorem prover for a clause logic using the resolution principle. Well-formed sentences in this logic are clauses. As mentioned in class, instead of using the implicative form, we will be using the disjunctive form, since this form is more suitable for automatic manipulation. The syntax of sentences in the clause logic is thus:

$$\begin{aligned} \text{Clause} &\rightarrow \text{Literal} \vee \dots \vee \text{Literal} \\ \text{Literal} &\rightarrow \neg \text{Atom} \mid \text{Atom} \\ \text{Atom} &\rightarrow \mathbf{True} \mid \mathbf{False} \mid P \mid Q \mid \dots \end{aligned}$$

We will regard two clauses as identical if they have the same literals. For example, $q \vee \neg p \vee q$, $q \vee \neg p$, and $\neg p \vee q$ are equivalent for our purposes. For this reason, we adopt a standardized representation of clauses, with duplicated literals always eliminated.

When modeling real domains, clauses are often written in the form:

$$\text{Literal} \wedge \dots \wedge \text{Literal} \Rightarrow \text{Literal}$$

In this case, we need to transform the clauses such that they conform to the syntax of the clause logic. This can always be done using the following simple rules:

1. $(p \Rightarrow q)$ is equivalent to $(\neg p \vee q)$
2. $(\neg(p \vee q))$ is equivalent to $(\neg p \wedge \neg q)$
3. $(\neg(p \wedge q))$ is equivalent to $(\neg p \vee \neg q)$
4. $((p \wedge q) \wedge \dots)$ is equivalent to $(p \wedge q \wedge \dots)$
5. $((p \vee q) \vee \dots)$ is equivalent to $(p \vee q \vee \dots)$
6. $(\neg(\neg p))$ is equivalent to p

The proof theory of the clause logic contains only the resolution rule:

$$\frac{\neg a \vee l_1 \vee \dots \vee l_n, \quad a \vee L_1 \vee \dots \vee L_m}{l_1 \vee \dots \vee l_n \vee L_1 \vee \dots \vee L_m}$$

If there are no literals l_1, \dots, l_n and L_1, \dots, L_m , the resolution rule has the form:

$$\frac{\neg a, a}{\mathbf{False}}$$

Remember that inference rules are used to generate new valid sentences, given that a set of old sentences are valid. For the clause logic this means that we can use the resolution rule to generate new valid clauses given a set of valid clauses. Consider a simple example where $p \Rightarrow q$, $z \Rightarrow y$ and p are valid clauses. To prove that q is a valid clause we first need to rewrite the rules to disjunctive form: $\neg p \vee q$, $\neg z \vee y$ and p . Resolution is then applied to the first and last clause, and we get:

$$\frac{\neg p \vee q, p}{q}$$

If **False** can be deduced by resolution, the original set of clauses is inconsistent. When making proofs by contradiction this is exactly what we want to do. The approach is illustrated by the resolution principle explained below.

The Resolution Principle

To prove that a clause is valid using the resolution method, we attempt to show that the negation of the clause is *unsatisfiable*, meaning it cannot be true under any truth assignment. This is done using the following algorithm:

1. Negate the clause and add each literal in the resulting conjunction of literals to the set of clauses already known to be valid.
2. Find two clauses for which the resolution rule can be applied. Change the form of the produced clause to the standard form and add it to the set of valid clauses.
3. Repeat 2 until **False** is produced, or until no new clauses can be produced. If no new clauses can be produced, report failure; the original clause is not valid. If **False** is produced, report success; the original clause is valid.

Consider again our example. Assume we now want to prove that $\neg z \vee y$ is valid. First, we negate the clause and get $z \wedge \neg y$. Then each literal is added to the set of valid clauses (see 4. and 5.). The resulting set of clauses is:

1. $\neg p \vee q$
2. $\neg z \vee y$
3. p
4. z
5. $\neg y$

Resolution on 2. and 5. gives:

1. $\neg p \vee q$
2. $\neg z \vee y$

3. p
4. z
5. $\neg y$
6. $\neg z$

Finally, we apply the resolution rule on 4. and 6. which produces **False**. Thus, the original clause $\neg z \vee y$ is valid.

(A) The Program

Files and Task Description

Your program should take exactly one argument from the command line:

1. The path to a `.kb` file that contains the initial KB and the clause whose validity we want to test. The input file contains n lines organized as follows: the first $n - 1$ lines describe the initial KB, while line n contains the (original) clause to test. Note that the KB is written in CNF, so each clause represents a disjunction of literals. The literals of each clause are separated by a blank space, while negated variables are prefixed by \sim .

Your program should adhere to the following policy:

- If the negated version of the clause to validate has ANDs, your program should split it into separate clauses. These clauses should be added to the KB from left to right order.
- Resolution should proceed as follows: For each clause $i[1,n]$ (where n is the last clause in the KB), attempt to resolve clause i with every previous clause $j[1,i]$ (in order). If a new clause is generated, it is added to the end of the KB (therefore the value of n changes). Your system should continue trying to resolve the next clause $(i + 1)$ with all previous clauses until 1) a contradiction is found (in which case 'Contradiction' should be added to the KB) or 2) all possible resolutions have been performed.
- Redundant generated clauses should not be added to the KB. A clause is redundant if the KB contains another clause which is logically equivalent to it.
- Clauses that evaluate to True should not be added to the KB.
- Generated clauses should not have redundant (repeated) literals.

Requirements: Output

Your program should implement the resolution algorithm as explained in the previous section. Your program should output a line for every clause in the final KB (in the order they were added), each line should be single-space-separated and contain: 1) the clause number followed by a period (starting from 1), 2) the clause in DNF, and 3) the parent clauses (if this clause was generated through resolution) written as $\{i, j\}$. Finally, your program should print a final line containing the word *Valid* or *Fail* depending on whether the proof by contradiction succeeded or not.

Let us consider a correct solution for testing the validity of $\neg z \vee y$, given the input:

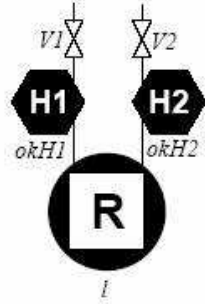
$\sim p \quad q$
 $\sim z \quad y$
 p
 $\sim z \quad y$

Your program's output should be:

1. $\sim p \quad q \quad \{\}$
2. $\sim z \quad y \quad \{\}$
3. $p \quad \{\}$
4. $z \quad \{\}$
5. $\sim y \quad \{\}$
6. $q \quad \{3, 1\}$
7. $y \quad \{4, 2\}$
8. $\sim z \quad \{5, 2\}$
6. Contradiction $\{7, 5\}$
Valid

(B) Power Plant Diagnosis

In the last part of this assignment you will be using your resolution prover to verify the safety requirements of a reactor unit in a nuclear power plant. The reactor unit is shown in the figure on the next page and consists of a reactor R , two heat exchangers $H1$ and $H2$, two steam valves $V1$ and $V2$, and a control stick l for changing the level of energy production. The state of the reactor unit is given by 5 propositional variables l , $okH1$, $okH2$, $V1$ and $V2$. If l has the value **True** the production level is 2 energy units. Otherwise, the production level is 1 energy unit. At least one working heat exchanger is necessary for each energy unit produced to keep the reactor from overheating. Unfortunately a heat exchanger i can start leaking reactive water from the internal cooling system to the surroundings. $okHi$ is **False** if heat exchanger Hi is leaking. Otherwise, $okHi$ is **True**. When a heat exchanger i is leaking, it must be shut off by closing its valve Vi . The state variable Vi indicates whether the valve Vi is closed (**False**) or open (**True**). Formally, the safety requirements are described by the following clauses:



$$\begin{aligned}
NoLeak \wedge LowTemp &\Rightarrow ReactorUnitSafe \\
NoLeakH1 \wedge NoLeakH2 &\Rightarrow NoLeak \\
okH1 &\Rightarrow NoLeakH1 \\
\neg okH1 \wedge \neg V1 &\Rightarrow NoLeakH1 \\
okH2 &\Rightarrow NoLeakH2 \\
\neg okH2 \wedge \neg V2 &\Rightarrow NoLeakH2 \\
l \wedge V1 \wedge V2 &\Rightarrow LowTemp \\
\neg l \wedge V1 &\Rightarrow LowTemp \\
\neg l \wedge V2 &\Rightarrow LowTemp
\end{aligned}$$

Assume that the current state of the reactor unit is given by the clauses:

$$\begin{aligned}
&\neg l \\
&\neg okH2 \\
&okH1 \\
&V1 \\
&\neg V2
\end{aligned}$$

1. Rewrite the safety rules from their implicative form to the disjunctive form used by your resolution prover. The initial set of valid clauses is the union of the rule clauses and the clauses defining the current state. Write the clauses in a file called `facts.txt`.
2. Use your resolution prover to test whether *LowTemp* is a valid clause:
 - (a) Save the input in a file called `task1.in`.
 - (b) Test the result of your prover.
3. Now test the validity of *ReactorUnitSafe* in a similar way:
 - (a) Save the input in a file called `task2.in`.
 - (b) Test the result of your prover.
4. Consider a simpler set of safety rules:

$$\begin{aligned}
NoLeakH1 \wedge NoLeakH2 &\Rightarrow NoLeak \\
okH1 &\Rightarrow NoLeakH1 \\
\neg okH1 \wedge \neg V1 &\Rightarrow NoLeakH1 \\
okH2 &\Rightarrow NoLeakH2 \\
\neg okH2 \wedge \neg V2 &\Rightarrow NoLeakH2
\end{aligned}$$

and a reduced current state description:

$$\begin{aligned}
&\neg okH2 \\
&okH1 \\
&\neg V2
\end{aligned}$$

Test the validity of $\neg NoLeak$:

- (a) Save the input in a file `task3.in`.
- (b) Test the result of your prover.

Implementation and Submission Requirements

The program must be written in C++ (g++ 7.5.0), Java (openjdk 11.0.13 2021-10-19), or Python (3.6.9), and needs to be run/compile on the UTD Linux boxes without installing extra software.

You may submit as many source files as needed, but you must make sure you provide a main code entry that follows the following naming convention. Specifically, if you are using:

- **Python**
 - Make sure that your primary source file is `main.py` and that your code runs successfully after executing `python main.py <path_to_kb_file>`.
- **C++**
 - Make sure that your primary source file is `main.cpp` and that your code runs successfully after executing `g++ main.cpp -o a.out -std=c++17` and `./a.out <path_to_kb_file>`.
- **Java**
 - Make sure that your primary source file is `Main.java` and that your code runs successfully after executing `javac Main.java` and `java Main <path_to_kb_file>`.

Your code should not use any external libraries (except for `numpy-1.19.5` and `pandas-1.1.5` if you are using Python). If it does, the autograder will not be able to run your code and you will receive no credit.

Submission

Once you are done, sign in to gradescope. You will be able to see *Assignment 3 - Part I* under the Assignments section. Directly submit all your source files, task1.in, task2.in, task3.in and the facts.txt file to this submission folder. Do not create any folder and do not rename the files or upload the files in a zip file or folder (your homework will not be graded otherwise). If you worked in a group, make sure to add your partner when you submit!

Grading

We will be using an output-based auto-grader for this submission, so make sure you follow the formatting from the example test files: be careful not to insert extra lines, tabs instead of spaces, etc ... When you submit, your code will be **graded using hidden test cases**, so we encourage you to test your code thoroughly. More information about the autograder will be available on Piazza shortly.

Important: Be mindful of the efficiency of your implementation; as the test cases we will use are quite long, poorly written code might time out (and receive no credit!). We will provide you with example input files and approximate timings for each, so you can get an idea of how fast your code is.