Foundations of Artificial Intelligence

Lorenzo Bozzoni

December 12, 2023

Contents

0.1 Theorem proving

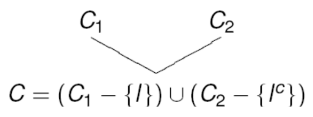
We are going to consider three algorithms:

- Resolution
- Forward chaining
- Backward chaining

0.2 Resolution

Propositional resolution is an extremely powerful inference procedure for propositional logic. Propositional resolution works on any set of sentences in clausal form (CNF): set of clauses. It works by refutation : to prove that $KB \models \alpha$ it proves that $KB \wedge \neg\alpha$ is unsatiasfiable, namely it builds a refutation for $KB \wedge \neg\alpha$ by looking for the empty clause (similar to DPLL).

Resolution applies (in all the possible ways) the resolution rule to the clauses (original ones + those derived by the previous applications of the resolution rule). Suppose C_1, C_2 are two clauses such that literal l in C_1 and literal l^C in C_2 are complementary (same propositional symbol with different sign), then clauses C_1 and C_2 can be resolved into a new clause C called **resolvent**: $C = (C_1 - \{l\}) \cup (C_2 - \{l^C\})$.

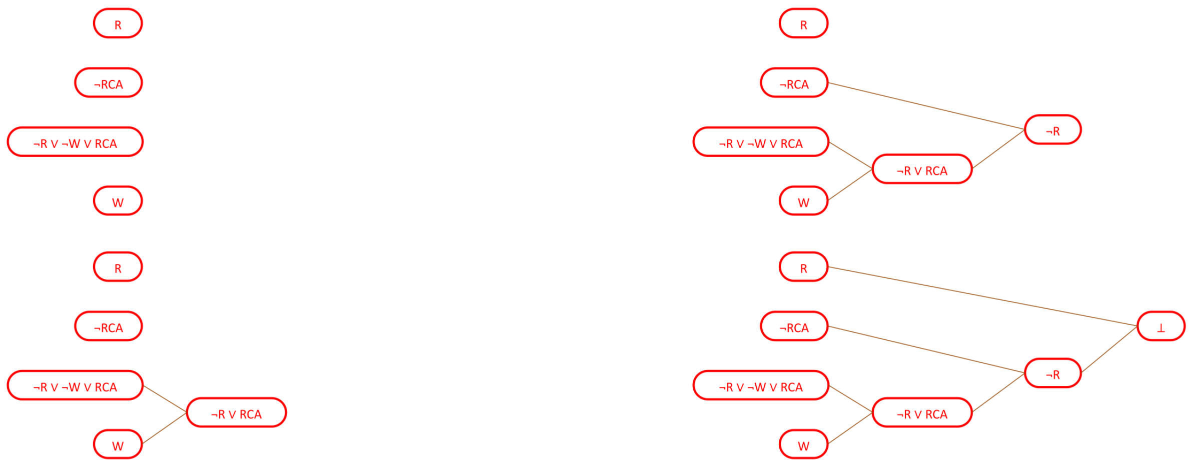


$$\frac{A \vee B, \neg B}{A} \qquad \frac{A \vee B \vee \neg C \vee D, \neg A \vee \neg E \vee F}{B \vee \neg C \vee D \vee \neg E \vee F}$$

The set of clauses: $\ldots\{A\},\ldots,\{\neg A\},\ldots$ is unsatiasfiable because it is stating that $\cdots \wedge A \wedge \cdots \wedge \neg A \wedge \ldots$. Resolution resolves A and $\neg A$ to the empty clause $\perp$. When the resolution finds the empty clause amounts to say that the original set of clauses is unsatiasfiable.

Example of resolution:
Consider again the example of Alice and the raincoat, in which we want to prove $\{a, b\} \models c$.

- $R \wedge W \to RCA$ becomes $\neg R \vee \neg W \vee RCA$
- $R \wedge \neg RCA$ becomes R and $\neg RCA$ (two clauses)
- $\neg W$ is negated to W



Conclusion: the set $\{a, b, \neg c\}$ is unsatiasfiable, therefore $\{a, b\} \models c$.

Using propositional resolution, it is possible to build a theorem prover that is sound and complete for PL

- Resolution rule is sound
- Theorem: resolvent C is satisfiable if and only if clauses C_1, C_2 are simultaneously satisfiable
- Since resolvent is smaller than parent clauses, resolution stops at some point

Resolution strategies:

- **Unit resolution**: at least one of the parent clauses is a unit clause (it contains a single literal). Incomplete in general, but complete for Horn clauses (see later)
- **Input resolution**: at least one of the two parent clauses is a member of the initial (i.e., input) set of clauses $KB \land \neg\alpha$. Incomplete in general, but complete for Horn clauses (see later).
- **Linear resolution**: generalization of input resolution in which at least one of the parents is either in the initial set of clauses or is an ancestor of the other parent. Complete
- **Set of support resolution**: given a set of support S (which is a subset of the initial clauses such that the clauses not in S are satisfiable), every resolution involves a clause in S (the resolvent is added to S)

Implementation of resolution:

```python
def pl_resolution(kb, alpha):
    """
    [Figure 7.12]
    Propositional-logic resolution: say if alpha follows from KB.
    >>> pl_resolution(horn_clauses_KB, A)
    True
    """
    clauses = kb.clauses + conjuncts(to_cnf(~alpha))
    new = set()
    while True:
        n = len(clauses)
        pairs = [(clauses[i], clauses[j])
                 for i in range(n) for j in range(i + 1, n)]
        for (ci, cj) in pairs:
            resolvents = pl_resolve(ci, cj)
            if False in resolvents:
                return True
            new = new.union(set(resolvents))
        if new.issubset(set(clauses)):
            return False
        for c in new:
            if c not in clauses:
                clauses.append(c)
```

A **horn clause** is a special PL sentence:

- (conjunction of symbols) $\rightarrow$ symbol or
- propositional symbol (equivalent to true $\rightarrow$ symbol)

A KB composed of only Horn clauses is in Horn clauses is in Horn form, for example $C \land (B \rightarrow A) \land (C \land D \rightarrow B)$. **When converted in CNF, a Horn clause is a clause with at most one positive literal, while a definite clause has exactly one positive literal**: $C \land (B \rightarrow A) \land (C \land D \rightarrow B)$ becomes $C \land (\neg B \lor A) \land (\neg C \lor \neg D \lor B)$. Horn clauses represent:

- rules, $C \land D \rightarrow B$
- facts, C (equivalent to $T \rightarrow C$)
- goals $A \land B \rightarrow \perp$
- empty clause, $T \rightarrow \perp$

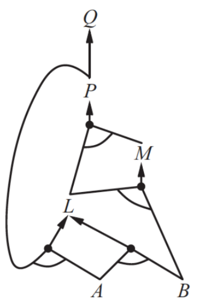
0.3 Forward chaining

Forward chaining considers definite clauses and applies **modus ponens** to generate new facts: given $X_1 \land X_2 \land \dots \land X_n \rightarrow X_1, X_2, \dots, X_n$, infer Y. Forward chaining keeps applying this rule, adding new facts, until nothing more can be added.

Example of forward chaining:

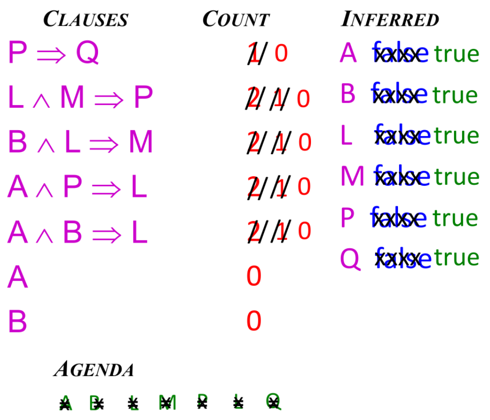
$$
\begin{aligned}
P &\implies Q \\
L \land M &\implies P \\
B \land L &\implies M \\
A \land P &\implies L \\
A \land B &\implies L \\
&\implies A \\
&\implies B
\end{aligned}
$$

Goal: Q



Forward chaining algorithm:

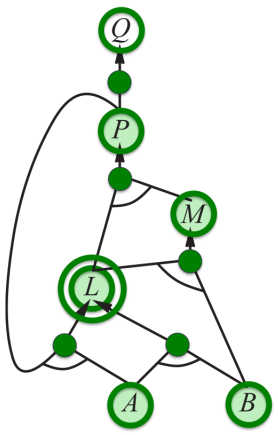
```
function PL-FC-ENTAILS?(KB, q) returns true or false
    count <- a table, where count[c] is the number of symbols in c’s premise
    inferred <- a table, where inferred[s] is initially false for all s
    agenda <- a queue of symbols, initially symbols known to be true in KB
    while agenda is not empty do
        p <- Pop(agenda)
        if p = q then return true
        if inferred[p] = false then
            inferred[p]<-true
            for each clause c in KB where p is in c.premise do
                decrement count[c]
                if count[c] = 0 then add c.conclusion to agenda
    return false
```



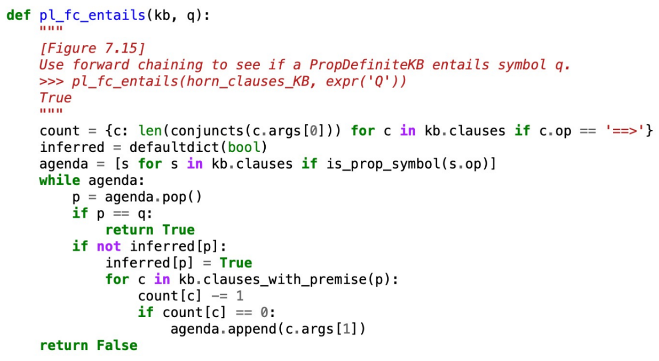
CLAUSES	COUNT	INFERRED
$P \Rightarrow Q$	~~1~~ 0	A ~~false~~ true
$L \land M \Rightarrow P$	~~2~~ ~~1~~ 0	B ~~false~~ true
$B \land L \Rightarrow M$	~~2~~ ~~1~~ 0	L ~~false~~ true
$A \land P \Rightarrow L$	~~2~~ ~~1~~ 0	M ~~false~~ true
$A \land B \Rightarrow L$	~~2~~ ~~1~~ 0	P ~~false~~ true
A	0	Q ~~false~~ true
B	0	

AGENDA

~~A~~ ~~B~~ ~~L~~ ~~M~~ ~~P~~ ~~L~~ ~~Q~~



Actual python implementation of forward chaining:



```
def pl_fc_entails(kb, q):
    """
    [Figure 7.15]
    Use forward chaining to see if a PropDefiniteKB entails symbol q.
    >>> pl_fc_entails(horn_clauses_KB, expr('Q'))
    True
    """
    count = {c: len(conjuncts(c.args[0])) for c in kb.clauses if c.op == '==>'}
    inferred = defaultdict(bool)
    agenda = [s for s in kb.clauses if is_prop_symbol(s.op)]
    while agenda:
        p = agenda.pop()
        if p == q:
            return True
        if not inferred[p]:
            inferred[p] = True
            for c in kb.clauses_with_premise(p):
                count[c] -= 1
                if count[c] == 0:
                    agenda.append(c.args[1])
    return False
```

Forward chaining is sound and complete for KBs composed of definite clauses.

- Soundness: follows from the soundness of modus ponens
- Completeness: the algorithm reaches a fixed point from where no new atomic sentences can be derived

Forward chaining can only derive new positive literals (facts), it cannot derive new rules.

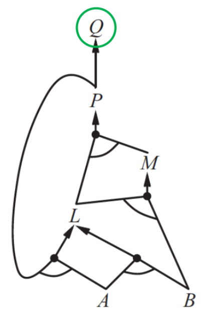
0.4 Backward chaining

Backward chaining works like forward chaining, but backward. Start from the goal (query) q, which is a positive literal. To prove q, check if q is already in the KB, otherwise prove all the premises of an implication concluding q. Avoid loops: check if new subgoal is already on the goal stack. Avoid repeated work: check if new subgoal has already been proved true, or has already failed.

Example of backward chaining:

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

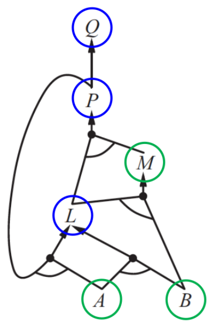
Goal: Q



AND-OR graph of the example

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

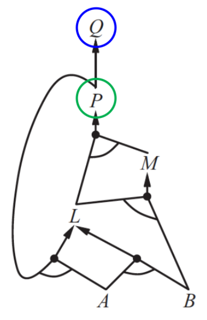
Goal: Q



AND-OR graph of the example

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

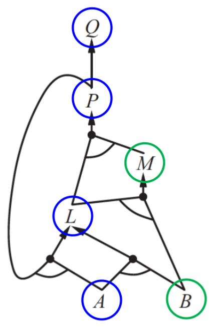
Goal: Q



AND-OR graph of the example

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

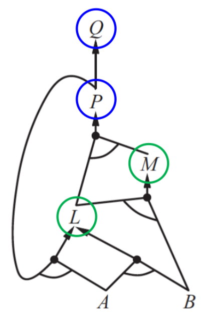
Goal: Q



AND-OR graph of the example

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

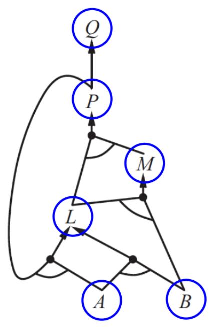
Goal: Q



AND-OR graph of the example

KB:
$P \Rightarrow Q$
$L \wedge M \Rightarrow P$
$B \wedge L \Rightarrow M$
$A \wedge P \Rightarrow L$
$A \wedge B \Rightarrow L$
$\Rightarrow A$
$\Rightarrow B$

Goal: Q



AND-OR graph of the example

Implementation of backward chaining: search on an AND-OR graph

```python
def and_or_search(problem):
    "Find a plan for a problem that has nondeterministic actions."
    return or_search(problem, problem.initial, [])

def or_search(problem, state, path):
    "Find a sequence of actions to reach goal from state, without repeating states on path."
    if problem.is_goal(state): return []
    if state in path: return failure # check for loops
    for action in problem.actions(state):
        plan = and_search(problem, problem.results(state, action), [state] + path)
        if plan != failure:
            return [action] + plan
    return failure

def and_search(problem, states, path):
    "Plan for each of the possible states we might end up in."
    if len(states) == 1:
        return or_search(problem, next(iter(states)), path)
    plan = {}
    for s in states:
        plan[s] = or_search(problem, s, path)
        if plan[s] == failure: return failure
    return [plan]
```

Backward chaining is sound and complete for KBs composed of definite clauses.

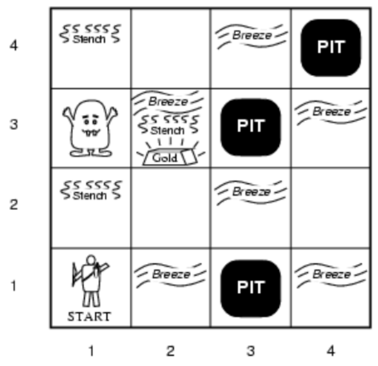
Forward chaining is **data-driven**, automatic, “unconscious” processing, can derive everything that is entailed by the KB, but lots of work is irrelevant to a specific goal. Backward chaining is **goal-driven**, appropriate for problem-solving, its complexity can be much less than linear in the size of the KB.Formulate the problems solved by inference procedures for propositional logic as search problems Define initial state, actions and result functions, goal test, and step cost. Formulate the satisfiability problem as a constraint satisfaction problem (CSP) Define variables, domains, and constraints.

Example of logical agent: Wumpus world (section 7.2 and 7.7.1 of the textbook).

The goal of the agent is to find the gold without falling into a pit or being eaten by the wumpus The agent can move between cells.

Cells adjacent to a pit are breezy. Cells adjacent to the wumpus are smelly.

The agent can just perceive its current cell.



- Perceptions: Stench, Breeze, ...

- Actions: LeftTurn, RightTurn, Forward, ...

Propositional symbols:

- $P_{i,j}$ means that there is a pit in cell (i, j)
- $B_{i,j}$ menas that there is a breeze in cell (i, j)

If the agent starts from cell $(1, 1)$ its KB is composed of:

- initial state: $\neg P_{1,1}$
- what the agent has perceived $\neg B_{1,1}$
- the rules of the world $B_{1,1} \iff (P_{1,2} \vee P_{2,1}), B_{2,1} \iff (P_{1,1} \vee P_{3,1} \vee P_{2,2}), \dots$

Given the KB at the initial step: