CS 188 Introduction to Artificial Intelligence
Spring 2024 Note 8

Author (all other notes): Nikhil Sharma

Author (Bayes' Nets notes): Josh Hug and Jacky Liang, edited by Regina Wang

Author (Logic notes): Henry Zhu, edited by Peyrin Kao

Credit (Machine Learning and Logic notes): Some sections adapted from the textbook *Artificial Intelligence: A Modern Approach*.

Last updated: March 5, 2024

Propositional Logic

Like other languages, logic has multiple dialects. We will introduce two: propositional logic and first-order logic. **Propositional logic** is written in sentences composed of **proposition symbols**, possibly joined by logical connectives. A proposition symbol is generally represented as a single uppercase letter. Each proposition symbol stands for an atomic proposition about the world. A **model** is an assignment of true or false to all the proposition symbols, which we might think of as a "possible world". For example, if we had the propositions A = "today it rained" and B = "I forgot my umbrella" then the possible models (or "worlds") are:

1. {A=true, B=true} ("Today it rained and I forgot my umbrella.")
2. {A=true, B=false} ("Today it rained and I didn't forget my umbrella.")
3. {A=false, B=true} ("Today it didn't rain and I forgot my umbrella.")
4. {A=false, B=false} ("Today it didn't rain and I did not forget my umbrella.")

In general, for N symbols, there are 2^N possible models. We say a sentence is **valid** if it is true in all of these models (e.g. the sentence *True*), **satisfiable** if there is at least one model in which it is true, and **unsatisfiable** if it is not true in any models. For example, the sentence $A \wedge B$ is satisfiable because it is true in model 1, but not valid since it is false in models 2, 3, 4. On the other hand $\neg A \wedge A$ is unsatisfiable as no choice for A returns True.

Below are some useful logical equivalences, which can be used for simplifying sentences to forms that are easier to work and reason with.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) \quad \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) \quad \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) \quad \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) \quad \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha \quad \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) \quad \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) \quad \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) \quad \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) \quad \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) \quad \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) \quad \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) \quad \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 7.11 Standard logical equivalences. The symbols α, β, and γ stand for arbitrary sentences of propositional logic.

One particularly useful syntax in propositional logic is the **conjunctive normal form** or **CNF** which is a conjunction of clauses, each of which a disjunction of literals. It has the general form $(P_1 \vee \cdots \vee P_i) \wedge \cdots \wedge (P_j \vee \cdots \vee P_n)$, i.e. it is an '**AND**' of '**OR**'s. As we'll see, a sentence in this form is good for some analyses. Importantly, every logical sentence has a logically equivalent conjunctive normal form. This means that we can formulate all the information contained in our knowledge base (which is just a conjunction of different sentences) as one large CNF statement, by 'AND'-ing these CNF statements together.

CNF representation is particularly important in propositional logic. Here we will see an example of converting a sentence to CNF representation. Assume we have the sentence $A \Leftrightarrow (B \vee C)$ and we want to convert it to CNF. The derivation is based on the rules in Figure 7.11.

1. Eliminate $\Leftrightarrow$: expression becomes $(A \Rightarrow (B \vee C)) \wedge ((B \vee C) \Rightarrow A)$ using **biconditional elimination**.
2. Eliminate $\Rightarrow$: expression becomes $(\neg A \vee B \vee C) \wedge (\neg(B \vee C) \vee A)$ using **implication elimination**.
3. For CNF representation, the "nots" ($\neg$) must appear only on literals. Using **De Morgan's** rule we obtain $(\neg A \vee B \vee C) \wedge ((\neg B \wedge \neg C) \vee A)$.
4. As a last step we apply the **distributivity law** and obtain $(\neg A \vee B \vee C) \wedge (\neg B \vee A) \wedge (\neg C \vee A)$.

The final expression is a conjunction of three OR clauses and so it is in CNF form.

Propositional Logical Inference

Logic is useful and powerful because it grants the ability to draw new conclusions from what we already know. To define the problem of inference we first need to define some terminology.

We say that a sentence A **entails** another sentence B if in all models that A is true, B is as well, and we represent this relationship as $A \models B$. Note that if $A \models B$ then the models of A are a subset of the models of B, $(M(A) \subseteq M(B))$. The inference problem can be formulated as figuring out whether $KB \models q$, where KB is our knowledge base of logical sentences, and q is some query. For example, if Elicia has avowed to never set foot in Crossroads again, we could infer that we will not find her when looking for friends to sit with for dinner.

We draw on two useful theorems to show entailment:

i.) ($A \models B$ *iff* $A \Rightarrow B$ is valid).

Proving entailment by showing that $A \Rightarrow B$ is valid is known as a **direct proof**.

ii.) ($A \models B$ *iff* $A \wedge \neg B$ is unsatisfiable).

Proving entailment by showing that $A \wedge \neg B$ is unsatisfiable is known as a **proof by contradiction**.

Model Checking

One simple algorithm for checking whether $KB \models q$ is to enumerate all possible models, and to check if in all the ones in which KB is true, q is true as well. This approach is known as **model checking**. In a sentence with a feasible number of symbols, enumeration can be done by drawing out a **truth table**.

For a propositional logical system, if there are N symbols, there are 2^N models to check, and hence the time complexity of this algorithm is $O(2^N)$, while in first-order logic, the number of models is infinite. In fact the problem of propositional entailment is known to be co-NP-complete. While the worst case runtime will inevitably be an exponential function of the size of the problem, there are algorithms that can in practice terminate much more quickly. We will discuss two model checking algorithms for propositional logic.

The first, proposed by Davis, Putnam, Logemann, and Loveland (which we will call the **DPLL algorithm**) is essentially a depth-first, backtracking search over possible models with three tricks to reduce excessive backtracking. This algorithm aims to solve the satisfiability problem, i.e. given a sentence, find a working assignment to all the symbols. As we mentioned, the problem of entailment can be reduced to one of satisfiability (show that $A \wedge \neg B$ is not satisfiable), and specifically DPLL takes in a problem in CNF. Satisfiability can be formulated as a constraint satisfaction problem as follows: let the variables (nodes) be the symbols and the constraints be the logical constraints imposed by the CNF. Then DPLL will continue assigning symbols truth values until either a satisfying model is found or a symbol cannot be assigned without violating a logical constraint, at which point the algorithm will backtrack to the last working assignment. However, DPLL makes three improvements over simple backtracking search:

1. **Early Termination**: A clause is true if any of the symbols are true. Therefore the sentence could be known to be true even before all symbols are assigned. Also, a sentence is false if any single clause is false. Early checking of whether the whole sentence can be judged true or false before all variables are assigned can prevent unnecessary meandering down subtrees.

2. **Pure Symbol Heuristic**: A pure symbol is a symbol that only shows up in its positive form (or only in its negative form) throughout the entire sentence. Pure symbols can immediately be assigned true or false. For example, in the sentence $(A \vee B) \wedge (\neg B \vee C) \wedge (\neg C \vee A)$, we can identify A as the only pure symbol and can immediately A assign to true, reducing the satisfying problem to one of just finding a satisfying assignment of $(\neg B \vee C)$.

3. **Unit Clause Heuristic**: A unit clause is a clause with just one literal or a disjunction with one literal and many falses. In a unit clause, we can immediately assign a value to the literal, since there is only one valid assignment. For example, B must be true for the unit clause $(B \vee false \vee \cdots \vee false)$ to be true.

function DPLL-SATISFIABLE?(s) **returns** *true* or *false*
inputs: s, a sentence in propositional logic

clauses ← the set of clauses in the CNF representation of s
symbols ← a list of the proposition symbols in s
return DPLL(*clauses*, *symbols*, { })

---

function DPLL(*clauses*, *symbols*, *model*) **returns** *true* or *false*

if every clause in *clauses* is true in *model* **then return** *true*
if some clause in *clauses* is false in *model* **then return** *false*
P, *value* ← FIND-PURE-SYMBOL(*symbols*, *clauses*, *model*)
if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
P, *value* ← FIND-UNIT-CLAUSE(*clauses*, *model*)
if P is non-null **then return** DPLL(*clauses*, *symbols* – P, *model* ∪ {P=*value*})
P ← FIRST(*symbols*); *rest* ← REST(*symbols*)
return DPLL(*clauses*, *rest*, *model* ∪ {P=*true*}) **or**
DPLL(*clauses*, *rest*, *model* ∪ {P=*false*}))

Figure 7.17 The DPLL algorithm for checking satisfiability of a sentence in propositional logic. The ideas behind FIND-PURE-SYMBOL and FIND-UNIT-CLAUSE are described in the text; each returns a symbol (or null) and the truth value to assign to that symbol. Like TT-ENTAILS?, DPLL operates over partial models.

DPLL: Example

Suppose we have the following sentence in conjunctive normal form (CNF):

$$(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$$

We want to use the DPLL algorithm to determine whether it is satisfiable. Suppose we use a fixed variable ordering (alphabetical order) and a fixed value ordering (true before false).

On each recursive call to the DPLL function, we keep track of three things:

- *model* is a list of the symbols we've assigned so far, and their values. For example, $\{A : T, B : F\}$ tells us the values of two symbols assigned so far.
- *symbols* is a list of unassigned symbols that still need assignments.
- *clauses* is a list of clauses (disjunctions) in CNF that still need to be considered on this call or future recursive calls to DPLL.

In other words, each call to DPLL is solving a smaller satisfiability problem, usually with fewer clauses, fewer symbols, and a model with some symbols already assigned.

We start by calling DPLL with an empty *model* (no symbols assigned yet), *symbols* containing all the symbols in the original sentence, and *clauses* containing all the clauses in the original sentence.

Our initial DPLL call looks like this:

- *model*: $\{\}$
- *symbols*: $[L, M, N, P, Q, R, S]$

- *clauses*: $(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$

First, we apply early termination: we check if given the current model, every clause is true, or at least one clause is false. Since the model hasn't assigned any symbol yet, we don't know which clauses are true or false yet.

Next, we check for pure literals. There are no symbols that only appear in a non-negated form, or symbols that only appear in a negated form, so there are no pure literals that we can simplify. For example, N is not a pure literal because the first clause uses the negated $\neg N$, and the second clause uses the non-negated N.

Next, we check for unit clauses (clauses with just one symbol). There's one unit clause S. For this overall sentence to be true, we know that S has to be true (there's no other way to satisfy that clause). Therefore, we can make another call to DPLL with S assigned to true in our model, and S removed from the list of symbols that still need assignments.

Our second DPLL call looks like this:

- *model*: $\{S : T\}$
- *symbols*: $[L, M, N, P, Q, R]$
- *clauses*: $(\neg N \vee \neg S) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (S)$

First, we can simplify the clauses by substituting in the new assignment (S is true, and $\neg S$ is false) from our model:

$$(\neg N \vee F) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L) \wedge (T)$$

$$(\neg N) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L)$$

With our new simplified clauses, we can check for early termination. We still don't have enough information to conclude that all sentences are true, or at least one sentence is false.

Next, we check for pure literals. As before, there are no symbols that only appear in a non-negated form, or symbols that appear in a negated form.

Next, we check for unit clauses. There's one unit clause $(\neg N)$. For this overall sentence to be true, $(\neg N)$ must be true, so N must be false.

Therefore, we can make another call to DPLL with N assigned to false in our model, and N removed from the list of symbols that still need assignments. We can also use the simplified clause that we computed from this call in DPLL (where we simplified S out of the clauses).

Our third DPLL call looks like this:

- *model*: $\{S : T, N : F\}$
- *symbols*: $[L, M, P, Q, R]$
- *clauses*: $(\neg N) \wedge (M \vee Q \vee N) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee N) \wedge (\neg R \vee \neg L)$

The first thing we do on this call is simplifying clauses by substituting in the new assignment (N is false, and $\neg N$ is true) from our model:

$$(T) \wedge (M \vee Q \vee F) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P \vee F) \wedge (\neg R \vee \neg L)$$

$$(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

With our new simplified clause, we check for early termination, and then we check for pure literals. As before, we don't find either one.

Next, we check for unit clauses. We don't find any clauses with just one symbol left.

At this point, we need to try to assign a value to a variable. From our fixed variable ordering, we'll assign M first, and from our fixed value ordering, we'll try making M true first. If assigning M true leads to an unsatisfiable sentence, then we need to backtrack and try again with M assigned to false. If assigning M false also leads to an unsatisfiable sentence, then we'll know that the entire sentence is unsatisfiable. In other words, we'll now make two recursive calls to DPLL, one with M true and one with M false, and check if either one produces a satisfiable assignment.

On the first DPLL call on the branch with M true, we'll add M true to our model, and use the simplified clause from the previous call:

- *model*: $\{S:T, N:F, M:T\}$
- *symbols*: $[L, P, Q, R]$
- *clauses*: $(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

First, we simplify clauses by substituting in the new assignment (M true) from our model:

$$(T \vee Q) \wedge (L \vee F) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(L) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

With our new simplified clause, we check for early termination; as before, we don't find it. However, we do find a pure literal, $\neg Q$ (recall that since there are no instances of Q and only instances of $\neg Q$, this counts as a pure literal). We set Q to be false so that $\neg Q$ can be true and proceed.

On our second DPLL call on the branch with M true:

- *model*: $\{S:T, N:F, M:T, Q:F\}$
- *symbols*: $[L, P, R]$
- *clauses*: $(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

We simplify our clauses accordingly:

$$(L) \wedge (L \vee T) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

Checking for early termination and pure literals, we find neither. We do find the unit clause (L) which we can then set to true.

On the next call in this same branch with M being true, we now have:

- *model*: $\{S:T, N:F, M:T, Q:F, L:T\}$
- *symbols*: $[P, R]$
- *clauses*: $(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

Let's simplify our clauses:

$$(T) \wedge (F \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee F)$$

$$(\neg P) \wedge (R \vee P) \wedge (\neg R)$$

Checking for early termination and pure literals, we find nothing. When checking for unit clauses, we find $(\neg P)$. Let's set that entire expression to true, i.e. setting P to false, for the next DPLL call.

Our next call proceeds as follows:

- *model*: $\{S:T, N:F, M:T, Q:F, L:T, P:F\}$
- *symbols*: $[R]$
- *clauses*: $(\neg P) \wedge (R \vee P) \wedge (\neg R)$

We simplify with P being set to false and get the clauses:

$$(T) \wedge (R \vee F) \wedge (\neg R)$$
$$(R) \wedge (\neg R)$$

We check for early termination. We note that this sentence has both R and $\neg R$, which cannot both be satisfied at the same time. At this point, we can say that this sentence is unsatisfiable.

Because the M true branch has ended in an unsatisfiable sentence, we backtrack to the point before assigning M true, and we make a DPLL call with M false instead. Our first DPLL call on the branch with M false:

- *model*: $\{S:T, N:F, M:F\}$
- *symbols*: $[L, P, Q, R]$
- *clauses*: $(M \vee Q) \wedge (L \vee \neg M) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

We simplify clauses by substituting in the new assignment (M false) from our model:

$$(F \vee Q) \wedge (L \vee T) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(Q) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

We aren't able to terminate early, and we don't find any pure literals. We find a unit clause Q, so we make another call to DPLL with Q true (and removed from our symbols list).

Our second DPLL call on the branch with M false:

- *model*: $\{S:T, N:F, M:F, Q:T\}$
- *symbols*: $[L, P, R]$
- *clauses*: $(Q) \wedge (L \vee \neg Q) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

Substituting the new assignment (Q true) into our clauses:

$$(T) \wedge (L \vee F) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

$$(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$$

We aren't able to terminate early, and we don't find any pure literals. We find a unit clause L, so we make another DPLL call with L true (and removed from our symbols list).

Our third DPLL call on the branch with M false:

- *model*: $\{S:T, N:F, M:F, Q:T, L:T\}$
- *symbols*: $[P, R]$
- *clauses*: $(L) \wedge (\neg L \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee \neg L)$

Substituting the new assignment (L true) into our clauses:

$$(T) \wedge (F \vee \neg P) \wedge (R \vee P) \wedge (\neg R \vee F)$$

$$(\neg P) \wedge (R \vee P) \wedge (\neg R)$$

We aren't able to terminate early, and we don't find any pure literals. We find two unit clauses $(\neg P)$ and $(\neg R)$. By our variable ordering, we choose P first, and so we make another DPLL call with P false (and removed from our symbols list).

Our third DPLL call on the branch with M false:

- *model*: $\{S:T, N:F, M:F, Q:T, L:T, P:F\}$
- *symbols*: $[R]$
- *clauses*: $(\neg P) \wedge (R \vee P) \wedge (\neg R)$

Substituting the new assignment (P false) into our clauses:

$$(T) \wedge (R \vee F) \wedge (\neg R)$$

$$(R) \wedge (\neg R)$$

We check for early termiination. We note that this sentence has both R and $\neg R$, which cannot both be satisfied at the same time. At this point, we can say that this sentence is unsatisfiable.

Because the M true assignment resulted in an unsatisfiable sentence, and the M false assignment resulted in an unsatisfiable sentence, we can conclude that this entire sentence is unsatisfiable, and we're done.

Theorem Proving

An alternate approach is to apply rules of inference to KB to prove that $KB \models q$. For example, if our knowledge base contains A and $A \Rightarrow B$ then we can infer B (this rule is known as *ModusPonens*). The two previously mentioned algorithms use the fact ii.) by writing $A \wedge \neg B$ in CNF and show that it is either satisfiable or not.

We could also prove entailment using three rules of inference:

1. If our knowledge base contains A and $A \Rightarrow B$ we can infer B (**Modus Ponens**).
2. If our knowledge base contains $A \wedge B$ we can infer A. We can also infer B. (**And-Elimination**).
3. If our knowledge base contains A and B we can infer $A \wedge B$ (**Resolution**).

The last rule forms the basis of the **resolution algorithm** which iteratively applies it to the knowledge base and to the newly inferred sentences until either q is inferred, in which case we have shown that $KB \models q$, or there is nothing left to infer, in which case $KB \not\models q$. Although this algorithm is both **sound** (the answer will be correct) and **complete** (the answer will be found) it runs in worst case time that is exponential in the size of the knowledge base.

However, in the special case that our knowledge base only has literals (symbols by themselves) and implications: $(P_1 \wedge \cdots \wedge P_n \Rightarrow Q) \equiv (\neg P_1 \vee \cdots \vee \neg P_2 \vee Q)$, we can prove entailment in time linear to the size of the knowledge base. One algorithm, **forward chaining** iterates through every implication statement in which the **premise** (left hand side) is known to be true, adding the **conclusion** (right hand side) to the list of known facts. This is repeated until q is added to the list of known facts, or nothing more can be inferred.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  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 symbols
  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
```

Figure 7.15 The forward-chaining algorithm for propositional logic. The *agenda* keeps track of symbols known to be true but not yet "processed." The *count* table keeps track of how many premises of each implication are as yet unknown. Whenever a new symbol p from the agenda is processed, the count is reduced by one for each implication in whose premise p appears (easily identified in constant time with appropriate indexing.) If a count reaches zero, all the premises of the implication are known, so its conclusion can be added to the agenda. Finally, we need to keep track of which symbols have been processed; a symbol that is already in the set of inferred symbols need not be added to the agenda again. This avoids redundant work and prevents loops caused by implications such as $P \Rightarrow Q$ and $Q \Rightarrow P$.

Forward Chaining: Example

Suppose we had the following knowledge base:

1. $A \to B$
2. $A \to C$
3. $B \wedge C \to D$
4. $D \wedge E \to Q$
5. $A \wedge D \to Q$
6. A

We'd like to use forward chaining to determine if Q is true or false.

To initialize the algorithm, we'll initialize a list of numbers *count*. The *i*th number in the list tells us how many symbols are in the premise of the *i*th clause. For example, the third clause $B \wedge C \to D$ has 2 symbols (B and C) in its premise, so the third number in our list should be 2. Note that the sixth clause A has 0 symbols in its premise, because it is equivalent to True $\to A$.

Then, we'll initialize *inferred*, a mapping of each symbol to true/false. This tells us which symbols we've found to be true. Initially, all symbols will be false, because we haven't proven any symbols to be true yet.

Finally, we'll initialize a list of symbols *agenda*, which is a list of symbols that we can prove to be true, but have not propagated the effects of yet. For example, if D were in the agenda, this would indicate that we're ready to prove that D is true, but we still need to check how that affects any of the other clauses. Initially, *agenda* will only contain the symbols we directly know to be true, which is just A here. (In other words, *agenda* starts with any clauses with 0 symbols in its premise.)

Our starting state looks like this:

- *count*: $[1,1,2,2,2,0]$
- *inferred*: $\{A:F, B:F, C:F, D:F, E:F, Q:F\}$
- *agenda*: $[A]$

On each iteration, we'll pop an element off *agenda*. Here, there's only one element that we can pop off: A. The symbol we popped off is not the symbol we want to analyze (Q), so we're not done with the algorithm yet.

According to the *inferred* table, A is false. However, since we've just popped A off the agenda, we're able to set it to true.

Next, we need to propagate the consequences of A being true. For each clause where A is in the premise, we'll decrement its corresponding count to indicate that there is one fewer symbol in the premise that needs to be checked. In this example, clauses 1, 2, and 5 contain A in the premise, so we'll decrement elements 1, 2, and 5 in *count*.

Finally, we check if any clauses have reached a count of 0. We note that this happened on clauses 1 and 2. This indicates that every premise in clauses 1 and 2 have been satisfied, so the conclusions in clauses 1 and 2 are ready to be inferred. For example, in clause 1, all premises (just A here) have been satisfied, so the conclusion B is ready to be inferred. We'll add the conclusions in clauses 1 and 2 to the agenda.

After iteration 0, our algorithm look like this:

- *count*: $[0,0,2,2,1,0]$
- *inferred*: $\{A:T, B:F, C:F, D:F, E:F, Q:F\}$
- *agenda*: $[B,C]$

On the next iteration, we'll pop an element off *agenda*. Here we've chosen to pop off B. The symbol we popped off is not the symbol we want to analyze (Q), so we're not done with the algorithm yet.

According to the *inferred* table, B is false. However, since we've just popped B off the agenda, we're able to set it to true.

Next, we need to propagate the consequences of B being true. The only clause where B is in the premise is clause 3. We have to decrement its corresponding count.

Finally, we check if any clauses have reached a count of 0. None of the clauses have newly reached a count of 0, so we can't draw any new conclusions, and we can't add anything new to the agenda.

After iteration 1, our algorithm look like this:

- *count*: $[0,0,1,2,1,0]$
- *inferred*: $\{A:T, B:T, C:F, D:F, E:F, Q:F\}$
- *agenda*: $[C]$

Next, we'll pop off C from the *agenda* (which is not Q so the algorithm isn't done yet). We can set C to true on the *inferred* list.

To propagate the consequences of C being true, we decrement the count for clause 3 (the only clause with C in the premise).

Clause 3 has newly reached a count of 0, so we can add its conclusion, D, to the agenda.

After iteration 2, our algorithm look like this:

- *count*: $[0,0,0,2,1,0]$
- *inferred*: $\{A:T, B:T, C:T, D:F, E:F, Q:F\}$
- *agenda*: $[D]$

Next, we'll pop off D from the *agenda* (not Q, so algorithm isn't done). We can set D to true on the *inferred* list.

To propagate the consequences of D being true, we decrement the counts for clauses 4 and 5 (which contain D in the premise).

Clause 5 has newly reached a count of 0, so we add its conclusion, Q, to the agenda.

After iteration 3, our algorithm look like this:

- *count*: $[0,0,0,1,0,0]$
- *inferred*: $\{A:T, B:T, C:T, D:T, E:F, Q:F\}$
- *agenda*: $[Q]$

Next, we'll pop off Q from the *agenda*. This is the symbol we wanted to evaluate, and popping it off the agenda indicates that it has been proven to be true. We conclude that Q is true and finish the algorithm.