

Motivation

Lecture 5

Logical Agents and Propositional Logic

So far all the knowledge we know about the world is in the form of a state, which is more or less a black-box to our reasoning system.

We will learn a new way to represent the world, so that the reasoning system can do much more for us.

Reference:

- Textbook Chapter 7

- We represent our knowledge about the world as “states”, which are usually treated as **black-boxes**.
- Most of the time we haven't argued about the inside of these black-boxes. Instead we argue about their **relations**.
- We have partially opened these blackboxes in the past, when we talk about **heuristics** and **optimization** and **evaluation functions**.
- They work well **only** in very limited environments. E.g., a heuristic for 15-puzzle is useless in the vacuum world.
But in the simple environments that we have studied, they are really the best representation, since such states and functions can be created very quickly.
- Another major exception: **CSPs**. There the states are represented in a way that the search algorithm can see and reason **directly**.
- Our hope: **all the world** is that open to search strategies.

AI(0270)

AI(0270)-5.1

How agents would work...

- Knowledge is represented as a **set of “sentences”**.
“Sentences” here has a specific meaning. We will see them in a moment. Unlike English sentences, such sentences are very concrete.
- The agent has a **knowledge base**, or **KB**, which captures the sentences “believed” by the agent. Initially some **background knowledge** may be known by the agent.
- Percepts are **turned into sentences** and added to KB. When an action is needed, a **query** in form of a **sentence** is made about the KB.
- The two operations are called **Tell** and **Ask** respectively.
- **Inference** will be done to **derive new sentences** from known sentences in KB. This can happen both when **new sentences are added** and when **query is made** to the KB.

AI(0270)-5.2

In a program...

As usual, we present the logic as a program, with parts to be filled in...

```

KB = initial_knowledge
t = 0 # time

# Call this everytime percept comes
def KB_Agent(percept):

    # Tell the KB about the percepts at time t
    Tell(KB, MakePerceptSentence(percept, t))

    # Ask the KB what action to do
    action = Ask(KB, MakeActionQuery(t))

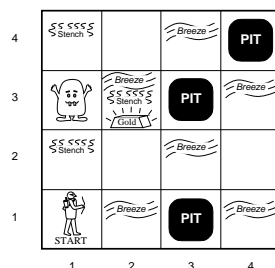
    # Tell the KB that the action it suggested is done at time t
    Tell(KB, MakeActionSentence(action, t))

    t += 1
    return action
    
```

AI(0270)-5.3

Example problem: Wumpus World

- You are at a corner of a 4 × 4 dark “cave”, like the one on the right.
- There is a nasty animal called the **wumpus** in one of the rooms, and will eat anyone entering the room.
But you can shoot an arrow to kill it.
- Each room may contain bottomless **pit**. Entering them results in death.
- We want the **gold** in the pit.
- The world is less hazardous than it sounds: when you are real close to a room with a pit or the wumpus, you can **sense** it.
- We have **no prior knowledge** to where are the pits, wumpus and gold.
Partial information makes it hard to apply problem solving approaches we learnt in previous lectures. The alternative to employ belief states is exponential.



AI(0270)-5.4

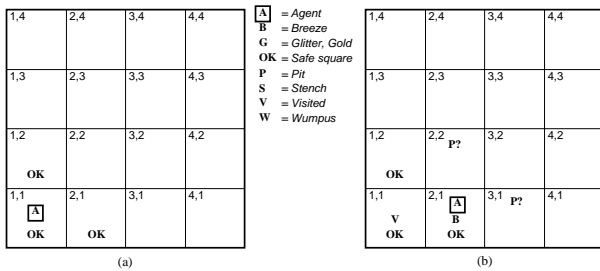
More formally...

- **Performance measure:** +1000 for getting gold, -1000 for falling into pit or eaten by wumpus, -1 for each action, -10 for using up the arrow.
- **Environment:** 4x4 grid of rooms, agent start at [1,1] facing right. Location of the gold and the wumpus is randomly chosen, and each location other than [1,1] has a probability of 0.2 to be a pit. Arrows fly until it hit the wall, walking into the wall has no effect.
- **Actuators:** turn **left** by 90 degrees, turn **right** by 90 degrees, move **forward** by 1 room, **grab** the gold, **shoot** his (only) arrow.
- **Sensors:** 5 boolean parts: a **stench** indicating wumpus, a **breeze** indicating pit, a **glitter** indicating gold, a **bump** indicating a wall, and a **scream** indicating death of the wumpus.
E.g., at [1,1], we sense the percept [False, False, False, False, False]. At [2,1] we would sense [False, True, False, False, False].

AI(0270)-5.5

Solving the problem ourselves

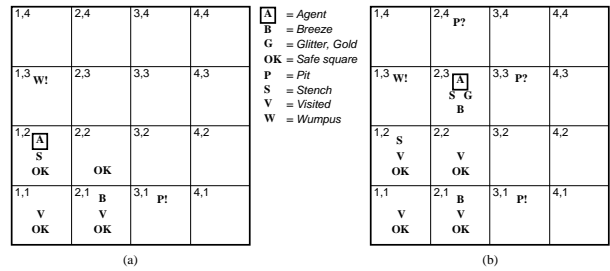
To see **what queries we would make** to our state, let's try solving the problem ourselves—using the cave we just see.



- Initially, we don't sense anything, meaning 1,2 and 2,1 are safe.
- We enter 2,1, feeling a breeze, and conclude a pit is at either 2,2 or 3,1.

AI(0270)-5.6

Solving the problem ourselves (cont'd)



- Then we try 1,2, feeling a stench, meaning the wumpus is at 1,3 or 2,2.
- Combining the two pieces of knowledge, now 2,2 becomes safe. We enter that, and found that 2,3 and 3,2 are both safe.
- Eventually we go to 2,3 and found the gold.

AI(0270)-5.7

Automatic reasoning

- What our state has to represent? Things like "Room 1,2 contains no pit", "Either room 2,2 or room 1,3 contains a wumpus", etc.
- Can we represent the state in a struct just as before?** Yes...
We can have 4 matrices stating whether there is a stench, a breeze, known to contain no pit, and known to contain no wumpus.
- But it will require **a lot of programming efforts to reason** and update the state to infer that there is a wumpus in a certain room, etc.
E.g., "if [1,2] has breeze and both [1,1] and [2,2] are known to have no pit, assign HasPit([3,1]) to true."
- Can we write a program which **automate the reasoning**? In particular, we **only want to tell knowledge about the world**, not how to reason about it. Inference should be the job of the agent.
And should be solved once for all [easy] problem.

AI(0270)-5.8

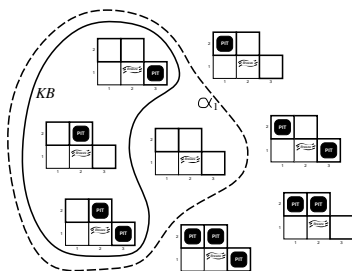
Logic as Knowledge Representation

- We have many **intermediate states** when reasoning about the Wumpus world. To be successful to automating reasoning, those steps must be represented.
- "Sentences" in our agent: **logic statements**. This gives us enough expressive power to represent anything we have so far.
- Syntax** of logic governs what is a logic statement and what is not.
We have that in Algebra: $x+y=4$ is an equation, $x=4y+$ is not.
- Semantic** of logic governs what sentence is **true** and what is **false**, given each **possible world**, or "**model**".
We have that in Algebra as well: $x+y=4$ is true in a model where $x=2$ and $y=2$.
- Reasoning involves checking whether a statement "logically follows from", or is **entailed by**, another statement.
We have that in Algebra as well: $x+y=4$ is entailed by $x=4-y$.

AI(0270)-5.9

Example 1

We use the Wumpus world to give an example. Suppose we walked into [2,1] and feel Breezy. We want to know which of [1,2], [2,2], [3,1] have pit.

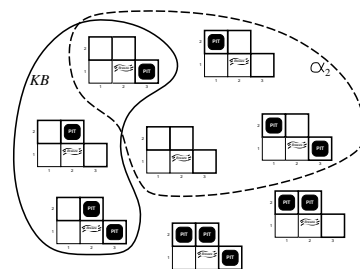


KB entails α_1 : "There is no pit in [1,2]", because all models "consistent" with KB have α_1 being true. We write $KB \models \alpha_1$.

AI(0270)-5.10

Example 2

On the other hand, KB does not entail α_2 : "There is no pit in [2,2]".



This is because in some models consistent with KB, α_2 is false. We write $KB \not\models \alpha_2$.

AI(0270)-5.11

Inference procedure

- Any algorithm that answers questions like “Given KB, is α entailed?” is called an **Inference algorithm**.
- Example: The **model checking algorithm** we have just seen, i.e., **check all models** to see whether any state consistent with KB is have α being false. If so, α is not entailed. Otherwise α is entailed.
- An algorithm is **sound** (or truth-preserving) if it only answer true for sentences that are logically entailed.
- An algorithm is **complete** if it can derive any sentence that is true. I.e., it never answer false or loop forever if given an entailed query.
- Sound and complete reasoning is important to the success of a logical reasoning agent. Although we will see that some strategies in use are not complete, or even not sound.

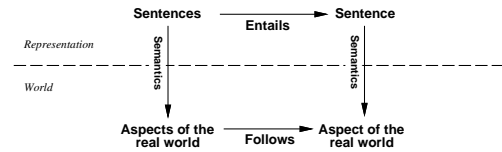
AI(0270)-5.12

Relations with real world

Grounding: for what reason (ground) we think the sentences correspond to the real world?

Answer: the percepts are **Tell**ed into KB, which are known to correspond to the world.

Entailment then enlarge the set of sentences that are known to the agent:



To go further, we need to fix a logic to use. We will see a simple one.

AI(0270)-5.13

Propositional Logic: a first, simple logic

- “Propositional logic” is a logic composed only of **propositions**.
- A proposition is an “atomic” sentence which **cannot be decomposed** into simpler ones. It can take values either **true** or **false**.
- We will use **propositional symbols** to represent them. E.g., $P_{1,1}$ might means [1,1] has pit, $B_{2,1}$ might mean there is breeze at [2,1].
- There are two special symbols **True** and **False**, which has values true and false respectively for all worlds. Other propositional symbols have different values in different worlds.
- A sentence can be a **simple sentence**, composed of just a **propositional symbol** or a **special symbol**, and nothing else.
- A sentence can also be a **complex sentence**, composed of other propositional symbol(s) and **connectives**.

AI(0270)-5.14

All the connectives

If P and Q are sentences (either simple or complex), then

- $\neg P$ is a **negation** sentence. It has the intuitive meaning of logical “not”.
- $(P \wedge Q)$ is a **conjunction** sentence. It has the intuitive meaning of logical “and”. Logicians usually call $P \wedge Q$ a *conjunction* of the *conjuncts* P and Q —in the same way that we call $1 + 2$ a sum.
- $(P \vee Q)$ is a **disjunction** sentence. It has the intuitive meaning of logical “or”. P and Q are called the *disjuncts*.
- $(P \Rightarrow Q)$ is an **implication** sentence. If P is false, then the implication is true. Otherwise, it has the same value as Q . P is called the *premise*, Q is called the *consequence*. (**Compositional!**)
- $(P \Leftrightarrow Q)$ is a **biconditional** sentence. It is true if P and Q are both true or both false.

The above list is from high to low precedence. Parentheses can be dropped if they follows from precedence.

AI(0270)-5.15

Simple knowledge base

For example, in the wumpus world, after the first step going East, the KB might end up with the following:

- There is no pit in [1,1]:

$$R_1: \quad \neg P_{1,1}$$

- A square is breezy if there is a pit around it. E.g.,

$$R_2: \quad B_{1,1} \Leftrightarrow P_{1,2} \vee P_{2,1}$$

$$R_3: \quad B_{2,1} \Leftrightarrow P_{1,2} \vee P_{2,2} \vee P_{3,1}$$

- Percepts so far...

$$R_4: \quad \neg B_{1,1}$$

$$R_5: \quad B_{2,1}$$

AI(0270)-5.16

Inference based on truth table

How to do inference in propositional logic? One can use model checking, or “**truth table**”. E.g., for our case:

- With 7 symbols ($B_{1,1}, B_{2,1}, P_{1,1}, P_{1,2}, P_{2,2}, P_{2,1}, P_{3,1}$) in this case, there are $2^7 = 128$ possible worlds.
- To know whether a particular sentence (say $\neg P_{1,2}$) is true, we check all combinations to find those states that are consistent with KB, ...
- And see whether $\neg P_{1,2}$ is true in all states.
- This directly implements the definition of entailment, thus is **sound** and **complete**.

AI(0270)-5.17

Truth table checking

Checking truth table is quite straight-forward...

```
def TTEntails(KB, sentence): # Does KB ⊨ sentence?
    symbols = KB.propositions # A list of propositional symbols
    return TTCheckAll(KB, sentence, symbols, [])

# check whether all truth assignments of symbols, given the known
# assignments in model, have KB => sentence
def TTCheckAll(KB, sentence, symbols, model):
    if symbols.empty():
        if (PLTrue(KB, model)): # Is KB true under model?
            return PLTrue(sentence, model)
        return true
    P = symbols.first()
    rest = symbols.rest()
    return TTCheckAll(KB, sentence, rest, model.extend((P, true))) and \
           TTCheckAll(KB, sentence, rest, model.extend((P, false)))
```

It always terminate, since there are finitely many models to check. But it is rather slow (Cost: $O(2^n)$). We will see faster ways.

AI(0270)-5.18

Some basic concepts

- Two sentences α and β are **logically equivalent** if they have the same value in **all models**. We write $\alpha \equiv \beta$. E.g., $P \vee Q \equiv Q \vee P$.
- A sentence is **valid**, if it is true under **all** possible models. E.g., $P \vee \neg P$ is valid. Valid sentences are called tautologies.
- **Deduction theorem**: $P \models Q$ if and only if $P \Rightarrow Q$ is valid.
- A sentence is **satisfiable** if it is true under **some** models. E.g., for our example, $R_1 \wedge R_2 \wedge R_3 \wedge R_4 \wedge R_5$ is satisfiable, although not valid.
- A sentence is **unsatisfiable** if it is not satisfiable.
- **Connections**: α is valid if and only if $\neg\alpha$ is unsatisfiable. $\neg\alpha$ is not valid if and only if α is satisfiable.
- Proof by **contradiction** or **refutation**: $P \models Q$ if and only if $(P \wedge \neg Q)$ is unsatisfiable.

AI(0270)-5.19

Common logic equivalences

- $\alpha \wedge \beta \equiv \beta \wedge \alpha$: Commutativity of \wedge .
- $\alpha \vee \beta \equiv \beta \vee \alpha$: Commutativity of \vee .
- $(\alpha \wedge \beta) \wedge \gamma \equiv \alpha \wedge (\beta \wedge \gamma)$: Associativity of \wedge .
- $(\alpha \vee \beta) \vee \gamma \equiv \alpha \vee (\beta \vee \gamma)$: Associativity of \vee .
- $\neg(\neg\alpha) \equiv \alpha$: Double negation elimination.
- $\alpha \Rightarrow \beta \equiv \neg\beta \Rightarrow \neg\alpha$: Contraposition.
- $\alpha \Rightarrow \beta \equiv \neg\alpha \vee \beta$: Implication elimination.
- $\alpha \Leftrightarrow \beta \equiv (\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$: Biconditional elimination.
- $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$: de Morgan.
- $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$: de Morgan.
- $\alpha \wedge (\beta \vee \gamma) \equiv (\alpha \wedge \beta) \vee (\alpha \wedge \gamma)$: Distribution of \wedge .
- $\alpha \vee (\beta \wedge \gamma) \equiv (\alpha \vee \beta) \wedge (\alpha \vee \gamma)$: Distribution of \vee .

AI(0270)-5.20

Inference based on standard patterns

- One way to do inference is to **apply common patterns of rules** to the known knowledge. E.g., **Modus Ponens** is written like this:

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}$$

- This means that if we already know both $\alpha \Rightarrow \beta$ and α , we can infer β .
- We can show that this is correct, by showing that $((\alpha \Rightarrow \beta) \wedge \alpha) \Rightarrow \beta$ is a valid sentence (using truth table).
- Another common pattern: **And-Elimination**:

$$\frac{\alpha \wedge \beta}{\alpha}$$

- Also, all the logical equivalences can be used in **both** directions as inference rule.

AI(0270)-5.21

Inference based on standard patterns: continued

- Once we show that a pattern is correct, we can apply it to generate sound inferences **without checking models**.
- This is okay because **other parts of KB won't change** our conclusion: **monotonicity**.
- To establish a sentence, we can select some known sentences, choose an inference rule to apply, and repeat. This results in new sentences. If the desired knowledge is generated, we call the sequence a **proof**.
- Finding proof is just a search problem, and can be solved using any strategy we have learnt in the first few lectures.
- It can be more efficient than model checking because it allows us to **ignore irrelevant propositions** of KB.

AI(0270)-5.22

Example inference

Suppose KB contains R_1 through R_5 we have mentioned...

- $R_6: (B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$ (Bicond-Elim to R_2)
- $R_7: ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$ (And-Elim to R_6)
- $R_8: (\neg B_{1,1} \Rightarrow \neg(P_{1,2} \vee P_{2,1}))$ (Contra-position to R_7)
- $R_9: \neg(P_{1,2} \vee P_{2,1})$ (Modus-Ponens to R_8 and R_4)
- $R_{10}: \neg P_{1,2} \wedge \neg P_{2,1}$ (de-Morgan to R_9)

So neither [1,2] nor [2,1] have a pit.

It is clearly sound, but whether it is complete depends on what inference rules we use. The more rules, the more likely to be complete, but the slower.

AI(0270)-5.23

How many is enough?

It turned out that **one** is enough if we choose the right one! The choice: **resolution**: if l_i and m_j are negation of each other:

$$\frac{l_1 \vee \dots \vee l_k, \quad m_1 \vee \dots \vee m_n}{l_1 \vee \dots \vee l_{i-1} \vee l_{i+1} \vee \dots \vee l_k \vee m_1 \vee \dots \vee m_{j-1} \vee m_{j+1} \vee \dots \vee m_n}$$

Here each l_x and m_x is a literal, i.e., either a simple symbol or its negation. For example, if we have $P \vee Q$ and $\neg Q \vee R$, we conclude $P \vee R$.

Intuitively: if Q is false, P must be true. If Q is true, R must be true. Since Q is either true or false, $P \vee R$ must be true. (We say Q is resolved.)

Technicality 1: if a symbol appears both as l_x and m_y ($x \neq i, y \neq j$), then they should be **merged** into one.

Technicality 2: if a l_x is the negation of a m_y ($x \neq i, y \neq j$), the result is useless.

AI(0270)-5.24

But... it works only on disjunctions!

If resolution works only on disjunctions, how can we say it's **complete**?

- Answer part 1: It is **complete** for disjunctions. If KB contains only disjunctions, then resolution can prove any disjunctions that logically follows from KB.
- Answer part 2: Anything can be converted into the conjunction of **many disjunctions** (i.e., a set of disjunctions), so the limitation is not serious. We say the KB is in "Conjunctive normal form", or CNF.

So a resolution proof involves **two processes**:

- Put the knowledge base into CNF.
- Use a search algorithm to apply resolution to clauses repeatedly until either the goal is found, or no more clause can be generated.

This always terminates, since there is a **finite** number of disjunctions.

AI(0270)-5.25

Putting sentences into CNF

The process is mechanical. E.g., to convert $B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$:

- Step 1: eliminate biconditions (using bi-conditional elimination):
 $(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$
- Step 2: eliminate conditional (using conditional elimination):
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$
- Step 3: move negations "inwards" using de-Morgans and double-negation elimination:
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$
- Step 4: distribute \vee over \wedge whenever possible:
 $(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$

So we end up into 3 (completely unreadable) disjunctions.

AI(0270)-5.26

But... something can't be derived, how it can be complete?

- There are some sentences that resolution cannot derive. E.g., if KB contains just A , it cannot derive $A \vee B$, although it is clearly entailed.
- Completeness of resolution is in a **special sense**: we can **prove any sentences** using **refutation**, i.e., "refutation completeness".
- Instead of trying to derive $A \vee B$, we should instead add $\neg(A \vee B)$ into KB and show that the result is **unsatisfiable**. By refutation, $A \vee B$ is proved.
- What will happens after adding $\neg(A \vee B)$? We won't really do that, since we must turn it into CNF first. So we actually add $\neg A$ and $\neg B$.
- By resolving A and $\neg A$, we get a disjunct of nothing, i.e., "false". So our modified KB proves "false", a contradiction. A can be considered as $false \vee A$, so we end up with $false$ after resolution. Or we can simply say that an empty clause means contradiction.

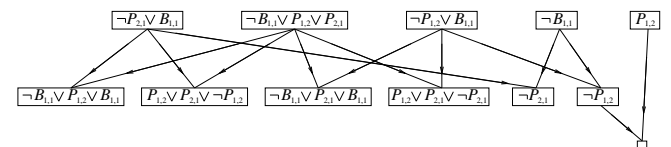
AI(0270)-5.27

Actual algorithm

```
def PL_Resolution(KB, sentence):
    clauses = set of clauses in (KB) and (not sentence)
    while true:
        new = []
        for C1 in clauses:
            for C2 in clauses:
                resolvent = PL_Resolve(C1, C2)
                if resolvent == Failed:
                    continue
                if resolvent == []:
                    return true
                if not resolvent in clauses:
                    new.append(resolvent)
            if new.empty():
                return false
        for new_clause in new:
            clauses.add(new_clause)
```

Example

To show there is no pit in [1,2] using R_1 through R_5 , we add $P_{1,2}$ to KB , and do resolution:



Since at the end we get the empty clause, we say $\neg P_{1,2}$ is entailed.

Note that we have generated a lot of **irrelevant sentences**. (Only the last 3 clauses are really used to generate the empty clause.)

In the next chapter, in the context of a richer logic, we will study techniques to reduce the amount of irrelevant sentences generated.

Although it is still rather slow.

AI(0270)-5.28

AI(0270)-5.29

Why resolution works?

- Since resolution is a “correct” inference rule, the algorithm which uses resolution exclusively is clearly **sound**.
I.e., if it declares a sentence is entailed, it is really entailed.
- But is it **complete**? In particular, if resolution tried all possibilities and at the end it cannot find new clauses, does it mean that the sentence we give is **not entailed**?
- Since we are doing refutation, we are actually checking whether KB and \neg sentence is unsatisfiable. By not entailed, we mean that there is a model which satisfy $KB \wedge \neg$ sentence.
- So all we need: **find a model in which $KB \wedge \neg$ sentence is satisfied**.
- The model depends on the **resolution closure** of $KB \wedge \neg$ sentence, i.e., the clauses variable at the end when we run *PL_Resolution*.

AI(0270)-5.30

Why it works (cont'd)

Given the resolution closure C of a set of clauses, we want to **find a model** in which **all clauses are satisfied**.

We can do the following to assign a value for each variable P_i :

- If there is a clause in the C containing $\neg P_i$, with all its other literals became false already, assign false to P_i .
- Otherwise, assign true to P_i .

It turns out that the resulting assignment always make all clauses in C satisfied, provided that C is closed under resolution. We leave the proof as an assignment question.

“ C is closed under resolution” means that if you take any 2 elements in C and perform resolution, the result is always in C . E.g., the set of real numbers is closed under addition, but not closed under division (e.g., take 2 and 0 to divide, you get a non-number).

AI(0270)-5.31

A restriction: Horn clauses

In the worst case, entailment takes exponential time. But if we **restrict** the possible sentences, we can have much more efficient algorithms.

- A CNF clause is said to be a **Horn clause** if at most 1 literal is positive.
- E.g., $\neg P_{2,1} \vee B_{1,1}$ is a Horn clause, while $\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}$ is not.
- Horn clauses with **exactly** 1 positive literal, like the Horn clause above, are called **definite clauses**, which forms basis for Logic Programming. For simplicity we assume clauses are all definite.
- Definite clauses can be written in a very readable “implication form”, like $P_{2,1} \Rightarrow B_{1,1}$ or $P_{1,1} \wedge P_{2,1} \Rightarrow B_{1,1}$. The positive part is called the head, the remainder is called the body.
Note that due to de-Morgan, the body is a conjunction, not disjunction!
- How much “more efficient”? Inference can be done in **linear** time!

AI(0270)-5.32

Chaining

- Again, we apply only 1 inference rule: Modus Ponens:

$$\frac{P_1 \wedge P_2 \wedge \dots \wedge P_k \Rightarrow Q, P_1, \dots, P_k}{Q}$$

- We won't use refutation. Instead we will prove the query directly. We call such algorithms **chaining** algorithms.
- Since Modus Ponens **can only prove 1-symbol sentences**, that is all chaining can do.
- Like resolution, the forward chaining strategy is a **data driven** strategy. It tries to proof new sentences from known knowledge base.
- It keeps counting, for each clause, the number of symbols which is not yet proved. Once this number becomes 0, the clauses “fires”, proving the symbol which is its head.

AI(0270)-5.33

Forward chaining algorithm

```
def PL_FC_Entails(KB, q):
    count = a table with one entry per clause, initially the size of body
    inferred = a table with one “false” entry per symbol
    agenda = a queue of symbols, initially empty
    for clause in KB.clauses:
        if (count[clause] == 0)
            agenda.push(clause.head)
            inferred[clause.head] = true
    while not agenda.empty():
        p = agenda.pop()
        # “KB.clauses with p in body” should be pre-computed for each symbol
        for clause in (KB.clauses with p in body):
            count[clause] -= 1
            if (count[clause] == 0 and not inferred[clause.head]):
                if (clause.head == q):
                    return true
                agenda.push(clause.head)
                inferred[clause.head] = true
    return false
```

AI(0270)-5.34

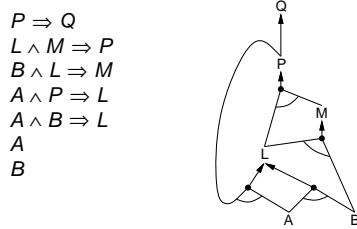
Completeness

- **For knowledge and query possible** with chaining, it is clearly **sound**, since Modus Ponens is correct.
- But it is also **complete**. Why? Suppose we run *PL_FC_Entails* for a KB and a q , and it returns false...
- Some symbols (in set *inferred*) are proved during *PL_FC_Entails*, and some not proved. Consider the model in which all the former symbols are true, all the latter symbols are false.
- In this model, q must be false (otherwise it is proved). But KB must be true, otherwise KB must have a clause with true body and false head, and the algorithm wouldn't have terminated!
- Since a model has KB true and q false, we must conclude that q is not inferred from KB .
- Since *PL_FC_Entails* always terminates, the algorithm is complete.

AI(0270)-5.35

And-Or graph

And-Or graph is one way to show how the query is proved. E.g.,



Note the two different symbols we use for AND nodes and OR nodes.

Apart from working bottom-up like Forward Chaining, one can work top-down from the query (“**Backward chaining**”). However, the algorithm can only be linear if intermediate results are remembered.

AI(0270)-5.36

AI(0270)-5.37

State of the art

- So we have a **fast** algorithm (Forward Chaining), but it works **only** on **Horn** clauses, and produce only **positive literals**. We have an algorithm which can work on **everything**, but is hopelessly **slow** (Resolution).
- Is there any other solution? There are a number of heuristics for **satisfiability**, which is nearly equivalent to our problem of entailment (but simpler). Basically, the idea of refutation is employed:
- Suppose we are given a KB , and is asked a query q .
- Instead of asking whether $KB \models q$, we can ask whether $KB \wedge \neg q$ is satisfiable.
- We conclude that $KB \models q$ if and only if the satisfiability checker says that $KB \wedge \neg q$ is unsatisfiable.
- So satisfiability and entailment are complement of each other. Since SAT is NP-hard, entailment is co-NP-hard.

Satisfiability with back-tracking: DPLL

First strategy: Use **Truth-Table** approach (i.e., check every combinations to look for a way to make every clause true), but speed it up with various **heuristics**. The name “DPLL” is the initials of the authors.

- **Early termination:** if a clause contains a literal that is assigned true, the clause can be removed from consideration. If a clause contains only false literals, the branch is doomed, so we can backtrack immediately.
- **Pure symbol heuristic:** If a literal is positive in all clauses, we can assign true to it without changing the satisfiability. Similarly, if a literal is negated in all clauses, we can assign false to it.
- **Unit clause:** If a clause only has 1 unassigned literal, but still not removed from consideration, that literal is assigned true immediately.

The only hard part is how to maintain enough data structures so that all these can be done and undone quickly in each recursive step.

AI(0270)-5.38

Satisfiability with Local search

Second strategy to solve satisfiability problems: use local search.

- We can start with an **arbitrary assignment**, and patch it up using **hill-climbing** or **simulated annealing**.
- The popular version, called **WalkSAT**, looks like this:

```

def WalkSAT(clauses, prob, max_flips):
    model = a random assignment of true and false to symbols in clauses
    for i in range(0, max_flips):
        if model satisfies clauses:
            return model
        with probability p:
            var = a random variable
        else:
            var = the variable which flipping would satisfy most clauses
    flip var in model
    return failure
    
```

AI(0270)-5.39

DPLL vs. local WalkSAT

- If the problem is **satisfiable**, **local search is usually faster** in finding it. The textbook shows a nice graph about the performance of the algorithms showing the running time of the two algorithms at different “difficulty level”.
- But it **never tells you that a problem is unsatisfiable**, which is the case where entailment is established.
- The only hope to use local search is to run it for a certain amount of time, and if it cannot find a solution, we **assume** that the query is entailed.
- Depending on how much risk we want to take, this can make it slower to use than DPLL in reality.

AI(0270)-5.40

The downside

But there is a primary problem of propositional logic: it is very tedious. We need **many symbols**, and **many long sentences**.

- E.g., if we might have 16 symbols $W_{1,1}$, $W_{1,2}$, etc., to represent whether a cell has a wumpus.

- How to express that **there is one wumpus?**

$$W_{1,1} \vee W_{1,2} \vee W_{1,3} \vee \dots$$

- How to express that there is **only one** wumpus? 256 sentences!

$$\begin{aligned}
 W_{1,1} &\Rightarrow \neg W_{1,2} \\
 W_{1,1} &\Rightarrow \neg W_{1,3} \\
 W_{1,1} &\Rightarrow \neg W_{1,4} \\
 W_{1,1} &\Rightarrow \neg W_{2,1} \\
 &\dots
 \end{aligned}$$

AI(0270)-5.41