

Strength of Weakness of Backward Chaining

Lecture 11

Resolution and practical deduction techniques

We have seen that Modus Ponens can be used when KB contains only horn sentences. We also see how backward-chaining uses the inference rule.

We will see some practical issues with chaining, and an alternative rule which allows the use of all CNF sentences.

Reference:

- Textbook: Section 9.5–9.6 and 10.2

AI(0270)

AI(0270)-11.1

- Backward chaining depends on that the **required conclusion** can be either **established directly**, or **found using one of only a few rules**.
- If this is the case, the **branching factor will be small**.
- It **does not matter** that the **same facts** can lead to **a lot of consequences**: we will only generate what we need.
- But sometimes, **the reverse is true**.
That is, a new known fact can only generate a few new sentences, but to deduce one of the new sentences, we need to examine a huge number of rules.
- E.g., suppose our KB does not contain *American(West)*. Instead, it contains *LiveIn(West, Florida)* and *LiveIn(x, Florida) ⇒ American(x)*.
And "*LiveIn(x, WashingtonDC) ⇒ American(x)*", "*LiveIn(x, California) ⇒ American(x)*", "*LiveIn(x, Texas) ⇒ American(x)*", ...
- To establish *American(West)*, the reasoning system has to use **all these sentences as subgoals**, although most fails!

An alternative: Forward Chaining

- In such cases, it would be more efficient if we **insert *American(West)* into KB once we know *LiveIn(West, Florida)***.
- This is called **forward chaining**.
- The procedure is done **without a focused goal**. E.g., if the agent learns *LiveIn(Chan, Florida)*, it will also insert *American(Chan)* into KB—even though perhaps no one is interested in *Chan*.
- It can be considered to be a **data-preprocessing phase**.
- **When to use forward chaining?**
 1. The rule should be applicable when new **knowledge** comes (i.e., from percepts), rather than when new **queries** come.
 2. It cannot create **too many new sentences**. Otherwise the reverse problem comes: KB is filled with lots of irrelevant knowledge.

AI(0270)-11.2

Code

```

procedure FORWARD-CHAIN(KB, p)
  if there is a sentence in KB that is a renaming of p then return
  Add p to KB
  for each ( $p_1 \wedge \dots \wedge p_n \Rightarrow q$ ) in KB such that for some i, UNIFY( $p_i, p$ ) =  $\theta$  do
    FIND-AND-INFER(KB, [p1, ..., pi-1, pi+1, ..., pn], q,  $\theta$ )
  end

procedure FIND-AND-INFER(KB, premises, conclusion,  $\theta$ )
  if premises = [] then
    FORWARD-CHAIN(KB, SUBST( $\theta$ , conclusion))
  else for each p' in KB such that UNIFY(p', SUBST( $\theta$ , FIRST(premises))) =  $\theta_2$  do
    FIND-AND-INFER(KB, REST(premises), conclusion, COMPOSE( $\theta, \theta_2$ ))
  end
  
```

The job is a bit easier than the backward chaining algorithm, but the code is even more difficult to read—unless you know what is the meaning of each argument of FIND-AND-INFER.

AI(0270)-11.3

Understanding the code

- FORWARD-CHAIN checks that a predicate *p* is **new**, and **add** it to KB. Then it starts off a **new round of forward chaining**.
- It finds all sentences with a predicate in the premise that **unifies with the new predicate *p***.
- The **remaining predicates must be proven** using facts (single predicate sentences) in KB before we can add the conclusion of the sentence to KB. This is done in FIND-AND-INFER.
- FIND-AND-INFER uses recursion to **find facts and unifications that prove each remaining predicate**.
- Once all premises are proven, the **combined substitution** is made to the **conclusion**.
- The result is **added to KB using FORWARD-CHAIN**, starting another round of Forward Chaining.

AI(0270)-11.4

Supporting chaining: implementing KB

During chaining, we frequently want to find all rules that has a **certain predicate** in a **conclusion** (backward chaining) or **premise** (forward chaining), or as the **whole atomic sentence** (both types).

To avoid having to search everything in the KB, we'd like to have **indices** listing them out **for each predicate**. E.g.,

Key	Positive	Negative	Conclusion	Premise
Brother	Brother(Richard, John) Brother(Ted, Jack) Brother(Jack, Bobbie)	¬Brother(Ann, Sam)	Brother(x, y) ∧ Male(y) ⇒ Brother(y, x)	Brother(x, y) ∧ Male(y) ⇒ Brother(y, x) Brother(x, y) ⇒ Male(x)
Male	Male(Jack) Male(Ted) ...	¬Male(Ann) ...	Brother(x, y) ⇒ Male(x)	Brother(x, y) ∧ Male(y) ⇒ Brother(y, x)

Some rules appears **multiple times** in the table because *Brother* appears in **both** the **premise** and the **conclusion** of the same sentence.

AI(0270)-11.5

Supporting disjunctions and conjunctions

We have mentioned that **chaining can only produce positive single-predicate terms**.

- What if we want to **query a disjunction**? E.g., Check if my birthday is sunday or holiday: $Sunday(MyBirthday) \vee Holiday(MyBirthday)$?
- This could be easy: **check each disjunct independently**, and combine the resulting answers.
- What if we want to **query a conjunction**? E.g., find a neighbouring room that is okay: $Neighbour(x, Room-1-1) \wedge OK(x)$?
- This is in fact simple. We can **temporarily add into KB a sentence** $Neighbour(x, Room-1-1) \wedge OK(x) \Rightarrow Query(x)$, and ask $Query(x)$.
- More complicated combination of conjunctions and disjunctions: use De Morgan's rule to make it a disjunction of conjunctions.
- How about **negations**??

AI(0270)-11.6

Negations and Closed-World assumption

- Note that chaining **never deal with negated sentences**.
- One possible way to deal with this: assume that **if we can't deduce a sentence, then its negation is true**.
- This is called the **closed world assumption**.
- If we take this view, the **default, negated knowledge** must be a **safe one to assume**: we shouldn't lose a lot if it's wrong. E.g.,
 - Instead of keeping predicates like $Wumpus(Room-1-1)$, we should keep predicates like $NoWumpus(Room-1-1)$.
Otherwise we'd walk into it because we "don't know there's a wumpus there"!
 - Instead of keeping predicates like $Smelly(Room-1-1)$, we should keep predicates like $NotSmelly(Room-1-1)$.
Otherwise we can't derive $NoWumpus(Room-1-1)$: smell implies the **presence** of wumpus, i.e., $\neg NoWumpus(Room-1-1)$, which we can't work with.

AI(0270)-11.7

Consequences of Closed-World assumption

- The closed-world assumption can be wrong, leading to **non-monotonic logic**: the **truth** of sentences **change over time**.
- E.g., at the beginning we might think that there is smell in a room, so there might be wumpus nearby. Later, when we actually walk in the room, we can conclude that the room has no smell, and thus no wumpus in neighbouring rooms.
- But this usually won't have large impact to our system: **our wrong negated knowledge won't be used to derive new knowledge**. Modus Ponens work with positive terms, and negative knowledge can't be used.
E.g., it is impossible for a rule to conclude that "because a room $x \neq Room-1-1$ have $\neg NoWumpus(x)$, $Wumpus(Room-1-1)$ must be true".
- Since chaining won't work with negated terms, **many systems choose not to store them at all**.

AI(0270)-11.8

Practical concerns of Forward Chaining

- Forward chaining, if performed, is **limited to a set of sentences** that is suitable for forward chaining.
- The knowledge base writer will determine which rules to allow forward chaining, based on **whether the result is small enough**, and **whether it is close enough to the information source**.
E.g., if our agent sense, at situation S_0 , no breeze percepts, it might forward-chain to that room-1-1 is not-breezy, and that room-1-2 and room-2-1 have no pit. He might decide that $OK(Room-1-2)$ is too far.
- If you know that some predicate will **always be added by forward chaining** if it is provable, you can save backward chaining time by **not looking for them during backward chaining**.
E.g., one might decide that one won't need to use backward chaining to look for not-breezy, because if it can be established, it should be in KB already.
- This saves time because if the predicate is not in KB, we don't need to spend time trying to prove it from things that FC already handled.

AI(0270)-11.9

Be careful!

Backward Chaining can be **removed** only for sentences **close to knowledge source**, or our reasoning agent will be considerably **weakened**.

E.g., suppose our KB contains:

1. Forward chaining: $Mammal(x) \Rightarrow Animal(x)$.
2. Backward chaining: $Pig(x) \Rightarrow Mammal(x)$

- Now if the knowledge that $Pig(McDull)$ comes, we can't do FC at all.
- But if we query $Animal(McDull)$, can't prove! No backward chaining rule can be found about the predicate $Animal$.
- If the forward and backward chaining rules are reversed, then our query can be answered.

This is some sort of reasoning about the reasoner—**meta-reasoning**.

AI(0270)-11.10

Incompleteness

- It is quite clear that **chaining is not complete**: some sentences simply can't be represented. E.g., suppose KB contains only
 1. $Rain(x) \Rightarrow GoParty(x)$
 2. $\neg Rain(x) \Rightarrow GoParty(x)$
- What will happen when we query $GoParty(Today)$?
- Note that the **sentence 2 can't be used by chaining**.
- Backward chaining will find that conclusion of sentence 1 matches the query, and try to establish $Rain(x)$ with $\{x/Today\}$ (i.e., $Rain(Today)$).
- But it can't be established. So backward chaining stops, and can't prove $GoParty(Today)$.
- However, $GoParty(Today)$ is of course entailed by the KB!

AI(0270)-11.11

Resolution

- If we want our agent to be complete, we **must use something more powerful than Modus Ponens**.
- In particular, our rule must be able to deal with **multiple positive terms in a single disjunction**.
This can be interpreted as some negative terms on the left of the implication.
- One such rule exists: **resolution**. Again, we will look for **using only one rule** to simplify program design.
- To recall, basic resolution is as follows:

$$\frac{\alpha \vee \beta, \quad \neg\beta \vee \gamma}{\alpha \vee \gamma}$$
- But this is too limiting: each clause must **contain exactly two terms**, and we have **no variables** in the rule.

AI(0270)-11.12

Inference rule: Generalized Resolution

- We will allow both sentences to have any number of terms, as long as one term is **exactly the reverse** of the other.
- And we will allow the terms to match only after unifications.
- Inference rule **Generalized Resolution**: for (positive or negative) terms $p_1, p_2, \dots, p_m, q_1, q_2, \dots, q_n$, where $\text{UNIFY}(p_i, \neg q_j) = \theta$:

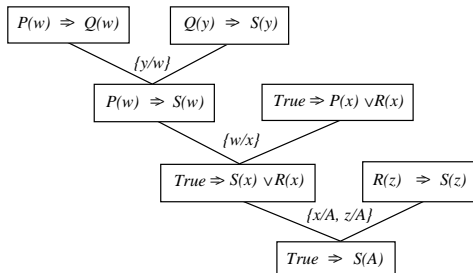
$$\frac{p_1 \vee p_2 \vee \dots \vee p_m, \quad q_1 \vee q_2 \vee \dots \vee q_n}{\text{SUBST}(\theta, \quad p_1 \vee p_2 \vee \dots \vee p_{i-1} \vee p_{i+1} \vee \dots \vee p_m \vee \quad q_1 \vee q_2 \vee \dots \vee q_{j-1} \vee q_{j+1} \vee \dots \vee q_n)}$$

- In other words, once we have **substitutions** that get p_i and q_j to be **exact reverse of each other**, we can **combine** two sentences, by **concatenating** them and **removing** both p_i and q_j .

AI(0270)-11.13

Doing chaining with resolution

We can use forward and backward chaining with resolution. Example:



Although INF is used, you can view them just as CNF sentences. Just turn the LHS of implications to negated terms.

AI(0270)-11.14

Incompleteness

- But even this is not complete. E.g., suppose KB contains $Q(A)$, and we query that $P(x) \vee \neg P(y)$.
- In other words, with no prior knowledge, ask **whether there are substitutions** of x and y that makes either $P(x)$ true, or $P(y)$ not true.
- There is of course such x : substitute x with A , then our query becomes $P(A) \vee \neg P(A)$, which is trivially true.
- But with chaining, there is **nothing to chain with**: P does not appear in any other sentence, so **backward chaining will stop at once**.
- Problem here: **the essential knowledge** to prove the query is **locked in the query itself**, which cannot be used because it is not proven yet.
- Is there any way for us to **pose the same question**, but **allow the knowledge of the query to be used**?

AI(0270)-11.15

Refutation

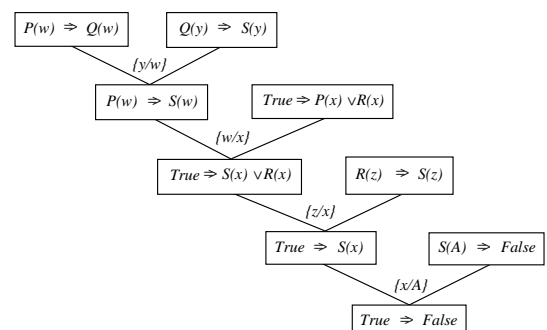
Solution: we know from grade school mathematics—**proof by contradiction**, or more formally **refutation**.

- What we want in FOL form: $\exists x, y \quad P(x) \vee \neg P(y)$.
- We want to prove it, so we **negate** it and put it into KB.
- What is the negation? It contains two sentences, $P(x)$ and $\neg P(y)$.
- Now we want to prove that contradiction occurred: we **want to derive a sentence False** from the sentences in KB.
Resolution with unifier $\{x/y\}$ produces the desired result.
- Since this can be done, our sentence **must be true**—assuming that KB is originally satisfiable.
- Note that unlike chaining, **the query is posed in FOL**, so variables are **universally quantified**.

AI(0270)-11.16

Example

If we use refutation, we get the following proof tree:



But there is a big practical problem... we can't do backward chaining!!

AI(0270)-11.17

How good is it in practice?

- Because **we don't know which predicate will contradict**, we cannot use backward chaining.
- So we are **forced to use forward chaining**.
Of course, we will need an extension that allows us to combine two sentences even though neither is atomic.
- As we have seen, **forward chaining is aimless**: it can generate **any sentences** into KB, usually ones that are pretty useless.
- This results in a **large branching factor**, even in cases where the KB is not really very large.
Of course, it can't be expected to be as fast as chaining using Modus Ponens. But it must not be so slow that it becomes useless.
- So the first problem to solve: **how to make the search more efficient in average case?**
Or, how to do something that looks like backward chaining?

AI(0270)-11.18

Unit Preference

- One very simple observation is that **the final sentence contains no terms at all**.
- To **reduce** the number of terms in a sentence, one has to **combine with an atomic sentence**, i.e., those which has only one term.
- So it seems to be a good strategy to try all applications of these sentences **before all other sentences**.
- This is called the **unit preference**: we prefer combining with sentences with only one term (unit sentence).
- It can be viewed as a special case of a general strategy to **prefer shorter sentences**—an A* search using the number of terms as the heuristic function.
- But branching factor is still large. We need some strategy to **make it unnecessary to generate some sentences**.

AI(0270)-11.19

Set of Support

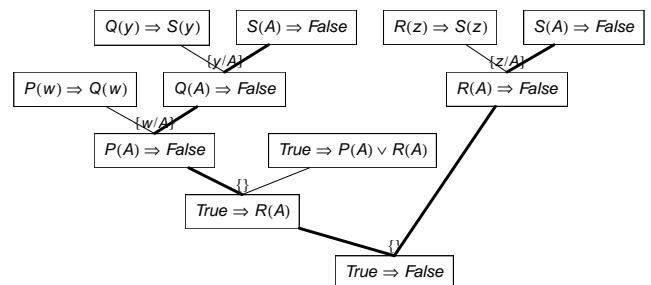
To get something that looks like backward chaining, we observe that **not all sentences have equal chance of getting contractions**.

- The original KB can be assumed to be **not contradicting**. Then...
- **The negated query must be a sentence** combined using resolution.
- By **searching only these sentences**, the search space is significantly reduced.
 - We keep **two separate sets** of sentences. One set, the **set of support**, is the **newly created sentences**. The other known facts in the KB are known as **axioms**.
 - Whenever applying resolution, **the first one always come from the set of support**. The second sentence may be in either set.
Now the search is quite similar to backward chaining, esp. if query is short.
- More importantly, the strategy is still **complete**.

AI(0270)-11.20

Example

The last proof we see is not using set of support. If we do, we get something like this:



The thick line show the main “trunk” of the proof: it contains the new sentences created during the search.

AI(0270)-11.21

Subsumption

- The set of support algorithm works quite well at the beginning, **when the set of support contains only a few sentences**.
- But when we proves more and more sentences and add them into the set of support, the branching factor escalates quickly.
- So it is important to **remove generated sentences** if they are not useful.
- One strategy, called **subsumption**, is to remove a specialized sentence from KB altogether if a more general sentence is known.
- Some examples:
 - If we know $P(x)$, remove $P(A)$.
 - If we know $Q(x)$ or $\neg P(x)$, remove $\neg P(x) \vee Q(x)$.

AI(0270)-11.22

Input Resolution

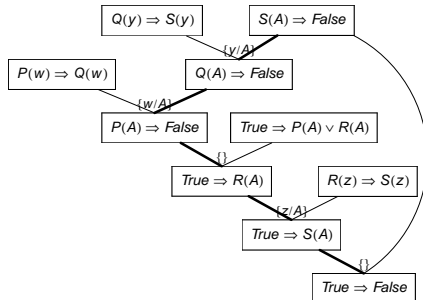
- But it is more effective if **we can avoid the expanding set of support**.
- Note that our sentence to prove contains **just one disjunct**. If the “proof tree” is really a list, then we can use a simpler algorithm.
- Our algorithm **doesn't add anything to KB**, but instead do searching with **state space** representing the current sentence.
- Our starting state is **that represented by the negation of the sentence to prove**. Our goal state is the sentence *False*.
- Operator: **apply Resolution with one axiom**.
I.e., we don't apply axiom with axiom, or state with another state.
- Derivation then becomes a **simple BFS**.
- Now the question: **is the procedure still complete?**
- Answer: **incomplete!** E.g., fails for our set of sentences.

AI(0270)-11.23

Linear Resolution: Restoring completeness

One modification can restore completeness: **linear resolution**.

- We also allow application of resolution of the current sentence α with **any ancestor sentence of α** . Our case:



AI(0270)-11.24

Answering existential questions

- What if our query is not $S(A)$, but is instead $\exists x S(x)$?
- Of course, we can still **negate** it to get $\neg S(x)$ and **perform refutation**.
- The question is, **how to get the binding?**
- During the refutation proof, there will be a **substitution made for x** . If not, then we can conclude that x can really be anything.
- This gives us the binding required.
- If we want a **list of bindings**, we can **continue the search** for alternative proof of **False**, remembering not to bind x to the values already found.

AI(0270)-11.25

Dealing with equality

There is one thing that we have ignored from the beginning: **how to deal with equality**.

Strategy 1: **Make a lot of new axioms for equality**. E.g.,

- $x = y \Rightarrow F(x) = F(y)$
- $x = y \Rightarrow R(x, \dots) \Leftrightarrow R(y, \dots)$
- $x = x$
- $x = y \Rightarrow y = x$
- $x = y, y = z \Rightarrow x = z$
- ...

- But this results in a lot of new symbols and sentences in KB...
- And every sentence needs many copies!

AI(0270)-11.26

Another solution

Strategy 2: Add some hacks to the resolution procedure:

- Replace all equivalent symbols by a single symbol.
- Demodulation**: have a separate rule that performs substitution to all sentences whenever equality of form $A = F(B)$ is established.

Note: this needs unification. E.g., when $A = F(B, C)$ is added, and a known sentence says $U(F(x, y), y) \vee P(x)$, we must unify $F(x, y)$ with $F(B, C)$, and add $U(A, C) \vee P(B)$.

- Paramodulation**: have a separate rule that performs substitution to all sentences whenever sentence of form $A = F(B) \vee P$ is established.

Details? Have to go into the reference mentioned in the book.

AI(0270)-11.27