9.3 Forward Chaining

- As before, let us consider knowledge bases in Horn normal form
- A definite clause either is atomic or is an implication whose body is a conjunction of positive literals and whose head is a single positive literal

Student(John)
EagerToLearn(x)
Student(y) ∧ EagerToLearn(y) ⇒ Thesis_2011(y)

- Unlike propositional literals, first-order literals can include variables
- The variables are assumed to be universally quantified

- As in propositional logic we start from facts, and by applying Generalized Modus Ponens are able to do forward chaining inference
- One needs to take care that a “new" fact is not just a renaming of a known fact

Likes(x, Candy)
Likes(y, Candy)

- Since every inference is just an application of Generalized Modus Ponens, forward chaining is a sound inference algorithm
- It is also complete in the sense that it answers every query whose answers are entailed by any knowledge base of definite clauses

Datalog

- In a Datalog knowledge base the definite clauses contain no function symbols at all
- In this case we can easily prove the completeness of inference
- Let in the knowledge base
  - p be the number of predicates,
  - k be the maximum arity of predicates (= the number of arguments), and
  - n the number of constants
- There can be no more than pn^k distinct ground facts
- So after this many iterations the algorithm must have reached a *fixed point*, where new inferences are not possible

- In Datalog a polynomial number of steps is enough to generate all entailments
- For general definite clauses we have to appeal to Herbrand's theorem to establish that the algorithm will find a proof
- If the query has no answer (is not entailed by the KB), forward chaining may fail to terminate in some cases
- E.g., if the KB includes the Peano axioms, then forward chaining adds facts

$$NatNum(S(0)).$$
$$NatNum(S(S(0))).$$
$$NatNum(S(S(S(0)))).$$
$$\ldots$$

- Entailment with definite clauses is semidecidable

9.4 Backward Chaining

- In predicate logic backward chaining explores the bodies of those rules whose head unifies with the goal
- Each conjunct in the body recursively becomes a goal
- When the goal unifies with a known fact – a clause with a head but no body – no new (sub)goals are added to the stack and the goal is solved
- Depth-first search algorithm
- The returned substitution is composed from the substitutions needed to solve all intermediate stages (subgoals)
- Inference in Prolog is based on backward chaining

9.4.2 Logic programming

- Prolog, Alain Colmerauer 1972
- Program = a knowledge base expressed as definite clauses
- Queries to the knowledge base
- Closed world assumption: we assume ¬φ to be true if sentence φ is not entailed by the knowledge base
- Syntax:
  - Capital characters denote variables,
  - Small character stand for constants,
  - The head of the rule precedes the body,
  - Instead of implication use `:-`,
  - Comma stand for conjunction,
  - Period ends a sentence

```
thesis_2011(X):- student(X), eager_to_learn(X).
```

- Prolog has a lot of syntactic sugar, e.g., for lists and artihmetics

- Prolog program `append(X,Y,Z)` succeeds if list `Z` is the result of appending (catenating) lists `X` and `Y`

```
append([ ],Y,Y).
append([A|X],Y,[A|Z]):- append(X,Y,Z).
```

- Query: `append([1],[2],Z)?`

```
Z=[1,2]
```

- We can also ask the query

  `append(A,B,[1,2])?`:

  Appending what two lists gives the list `[1,2]`?
- As the answer we get back all possible substitutions

```
A=[ ]        B=[1,2]
A=[1]        B=[2]
A=[1,2]      B=[ ]
```


- The clauses in a Prolog program are tried in the order in which they are written in the knowledge base
- Also the conjuncts in the body of the clause are examined in order (from left to right)
- There is a set of built-in functions for arithmetic, which need not be inferred further
  - E.g., `X is 4+3` → `X=7`
- For instance I/O is taken care of using built-in predicates that have side effect when executed
- Negation as failure

```
alive(X):- not dead(X).
```

  "Everybody is alive if not provably dead"

- The negation in Prolog does not correspond to the negation of logic (using the closed world assumption)

```
single_student(X):-
        not married(X), student(X).
student(peter).
married(john).
```

- By the closed world assumption, `X=peter` is a solution to the program
- The execution of the program, however, fails because when `X=john` the first predicate of the body fails
- If the conjuncts in the body were inverted, it would succeed

- An equality goal succeeds if the two terms are unifiable
  - E.g., `X+Y=2+3` → `X=2, Y=3`
- Prolog omits some necessary checks in connection of variable bindings → Inference is not sound
- These are seldom a problem
- Depth-first search can lead to infinite loops (= incomplete)

```
path(X,Z):- path(X,Y), link(Y,Z).
path(X,Z):- link(X,Z).
```

- Careful programming, however, lets us escape such problems

```
path(X,Z):- link(X,Z).
path(X,Z):- path(X,Y), link(Y,Z).
```

- Anonymous variable _

```
member(X,[X|_]).
member(X,[_|Y]):- member(X,Y).
```

- Works just fine

```
member(d,[a,b,c,d,e,f,g])?
yes
member(2,[3,a,4,f])?
no
```

- But queries

```
member(a,X)?
member(a,[a,b,r,a,c,a,d,a,b,r,a])?
```

do not necessarily give the intended answers

- We can explicitly prune the execution of Prolog programs by *cutting*
- Negation using cut

```
not X:- X, !, fail.
not X.
```

- `fail` causes the program to fail
- At the point of a cut all bindings that have been made since starting to examine the rule are fixed
- For the conjuncts in the body preceding the cut, no new solutions are searched for
- Neither does one examine other rules having the same head

- Prolog may come up with the same answer through several inference paths
- Then the same answer is returned more than once

```
minimum(X,Y,X):- X<=Y.
minimum(X,Y,Y):- X>=Y.
```

- Both rules yield the same answer for the query `minimum(2,2,M)?`

- One must be careful in using cut for optimizing inference

```
minimum(X,Y,X):- X<=Y, !.
minimum(X,Y,Y).
```

- This program is erroneous, for instance `minimum(2,8,8)` holds according to it

- The key question of Prolog and logic programming obviously is efficiency of execution
- Prolog implementations use a wide variety of enhancement techniques
- For example, instead of generating all possible solutions for a subgoal before examining the next subgoal, a Prolog interpreter is content (so far) with just one
- Similarly variable binding is at each instant unique; only when the search runs into a dead end, can *backing up* to a choice point lead to unbinding of variables
- A stack of history, called the *trail*, needs to be maintained to keep track of all variable bindings

9.5 Resolution

- Kurt Gödel's completeness theorem (1930) for first-order logic: any entailed sentence has a finite proof

$$T \models \varphi \Leftrightarrow T \vdash \varphi$$

- It was not until Robinson's (1965) resolution algorithm that a practical proof procedure was found
- Gödel's more famous result is the incompleteness theorem: a logical system that includes the principle of induction, is necessary incomplete
- There are sentences that are entailed, but have no finite proof
- This holds in particular for number theory, which thus cannot be axiomatized

- For resolution, we need to convert the sentences to CNF
- E.g., "Everyone who loves all animals is loved by someone"

$$\forall x: [\forall y: Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y: Loves(y, x)].$$

- **Eliminate implications**

$$\forall x: [\neg\forall y: \neg Animal(y) \vee Loves(x, y)] \vee [\exists y: Loves(y, x)].$$

- **Move negation inwards**

$$\forall x: [\exists y: Animal(y) \wedge \neg Loves(x, y)] \vee [\exists y: Loves(y, x)].$$

- **Standardize variables**

$$\forall x: [\exists y: Animal(y) \wedge \neg Loves(x,y)] \vee [\exists z: Loves(z, x)].$$

- **Skolemization**

  $\forall$ x: [Animal(F(x)) $\wedge$ ¬Loves(x, F(x))] $\vee$ Loves(G(z), x).

- **Drop universal quantifiers**

  [Animal(F(x)) $\wedge$ ¬Loves(x, F(x))] $\vee$ Loves(G(z), x).

- **Distribute $\vee$ over $\wedge$**

  [Animal(F(x)) $\vee$ Loves(G(z), x)] $\wedge$
  [¬Loves(x, F(x)) $\vee$ Loves(G(z), x)].

- The end result is quite hard to comprehend, but it doesn't matter, because the translation procedure is easily automated

- First-order literals are complementary if one unifies with the negation of the other
- Thus the binary resolution rule is

$$\frac{\ell_1 \vee \dots \vee \ell_k,\ m}{(\ell_1 \vee \dots \vee \ell_{i-1} \vee \ell_{i+1} \vee \dots \vee \ell_k)(\theta)}$$

  where Unify(ℓ_i, ¬m) = θ
- For example, we can resolve
  [Animal(F(x)) $\vee$ Loves(G(x), x)] and [¬Loves(u, v) $\vee$ ¬Kills(u, v)]
  by eliminating the complementary literals Loves(G(x), x) and ¬Loves(u, v) with unifier θ = { u/G(x), v/x } to produce the **resolvent** clause [Animal(F(x)) $\vee$ ¬Kills(u, v)]

- Resolution is a complete inference rule also for predicate logic in the sense that we can check (not generate) all logical consequences of the knowledge base
- KB ⊨ α is proved by showing that KB $\wedge$ ¬ α is unsatisfiable through a proof by refutation

- *Theorem provers (automated reasoners)* accept full first-order logic, whereas most logic programming languages handle only Horn clauses
- For example Prolog intertwines logic and control
- In most theorem provers, the syntactic form chosen for the sentences does not affect the result
- Application areas: verification of software and hardware
- In mathematics theorem provers have a high standing nowadays: they have come up with novel mathematical results
- For instance, in 1996 a version of well-known Otter was the first to prove (eight days of computation) that the axioms proposed by Herbert Robbins in 1933 really define Boolean algebra

13 QUANTIFYING UNCERTAINTY

- In practice agents almost never have full access to the whole truth of their environment and, therefore, must act under *uncertainty*
- A logical agent may fail to acquire certain knowledge that it would require
- If the agent cannot conclude that any particular course of action achieves its goal, then it will be unable to act
- Conditional planning can overcome uncertainty to some extent, but it does not resolve it

- An agent based solely on logics cannot choose rational actions in an uncertain environment

- Logical knowledge representation requires rules without exceptions
- In practice, we can typically at best provide some *degree of belief* for a proposition
- In dealing with degrees of belief we will use *probability theory*
- Probability 0 corresponds to an unequivocal belief that the sentence is false and, respectively, 1 to an unequivocal belief that the sentence is true
- Probabilities in between correspond to intermediate degrees of belief in the truth of the sentence, not on its relative truth
- Utilities that have been weighted with probabilities give the agent a chance of acting rationally by preferring the action that yields the highest expected utility
- Principle of *Maximum Expected Utility* (MEU)

13.2 Basic Probability Notation

- A random variable refers to a part of the world, whose status is initially unknown
- Random variables play a role similar to proposition symbols in propositional logic
- E.g., *Cavity* might refer whether the lower left wisdom tooth has a cavity
- The *domain* of a random variable may be of type
  - *Boolean*:
    we write Cavity = true ⇨ cavity and
    Cavity = false ⇨ ¬cavity;
  - *discrete*: e.g., Weather might have the domain
    { sunny, rainy, cloudy, snow };
  - *continuous*: then one usually examines the cumulative distribution function; e.g., $X \leq 4.02$

- Elementary propositions can be combined to form complex propositions using all the standard connectives:

  $cavity \wedge \neg toothache$

- An atomic *event* is a complete specification of the world, i.e., an assignment of values to all the variables
- Properties of atomic events:
  - They are mutually exclusive
  - The set of all atomic events is exhaustive — at least one must be the case
  - Any particular atomic event entails the truth or falsehood of every proposition
  - Any proposition is logically equivalent to the disjunction of all atomic events that entail the truth of the proposition

$$cavity \equiv (cavity \wedge toothache) \vee (cavity \wedge \neg toothache)$$

Prior probability

- Degrees of belief are always applied to propositions
- Prior probability $P(a)$ is the degree of belief accorded to proposition a in the absence of any other information

  $P(cavity) = 0.1$

- In particular, prior probability is the agent's initial belief before it receives any percepts
- The probability distribution $\underline{P}(X)$ of a random variable X is a vector of values for probabilities of the elements in its (ordered) domain
- E.g., when $P(sunny) = 0.02$, $P(rainy) = 0.2$, $P(cloudy) = 0.7$, and $P(snow) = 0.08$, then

$$\underline{P}(Weather) = [0.02, 0.2, 0.7, 0.08]$$

- The joint probability distribution of two random variables is the product of their domains
- E.g., **P**(Weather, Cavity) is a 4×2 table of probabilities
- Full joint probability distribution covers the complete set of random variables used to describe the world

- For continuous variables it is not possible to write out the entire distribution as a table, one has to examine *probability density functions* instead
- Rather than examine point probabilities (that have value 0), we examine probabilities of value ranges
- We will concentrate mostly on discrete-valued random variables