CS 188 Introduction to Artificial Intelligence
Spring 2022 Note 4

These lecture notes are heavily based on notes originally written by Henry Zhu. All figures are from *Artificial Intelligence: A Modern Approach*.

A Knowledge Based Agent

Imagine a dangerous world filled with lava, the only respite a far away oasis. We would like our agent to be able to safely navigate from its current position to the oasis.

In reinforcement learning, we assume that the only guidance we can give is a reward function which will try to nudge the agent in the right direction, like a game of 'hot or cold'. As the agent explores and collects more observations about the world, it gradually learns to associate some actions with positive future reward and others with undesirable, scalding death. This way, it might learn to recognize certain cues from the world and act accordingly. For example, if it feels the air getting hot it should turn the other way.

However, we might consider an alternative strategy. Instead let's tell the agent some facts about the world and allow it to reason about what to do based on the information at hand. If we told the agent that air gets hot and hazy around pits of lava, or crisp and clean around bodies of water, then it could reasonably infer what areas of the landscape are dangerous or safe based on its readings of the atmosphere. This alternative type of agent is known as a **knowledge based agent**. Such an agent maintains a **knowledge base**, which is a collection of logical **sentences** that encodes what we have told the agent and what it has observed. The agent is also able to perform **logical inference** to draw new conclusions.

The Language of Logic

Just as with any other language, logic sentences are written in a special **syntax**. Every logical sentence is code for a **proposition** about a world that may or may not be true. For example the sentence "the floor is lava" may be true in our agent's world, but probably not true in ours. We can construct complex sentences by stringing together simpler ones with **logical connectives** to create sentences like "you can see all of campus from the Big C *and* hiking is a healthy break from studying". There are five logical connectives in the language:

- $\neg$, **not**: $\neg P$ is true *if and only if (iff)* P is false. The atomic sentences P and $\neg P$ are referred to as **literals**.
- $\wedge$, **and**: $A \wedge B$ is true *iff* both A is true and B is true. An 'and' sentence is known as a **conjunction** and its component propositions the **conjuncts**.
- $\vee$, **or**: $A \vee B$ is true *iff* either A is true or B is true. An 'or' sentence is known as a **disjunction** and its component propositions the **disjuncts**.
- $\Rightarrow$, **implication**: $A \Rightarrow B$ is true unless A is true and B is false.

- $\Leftrightarrow$, **biconditional**: $A \Leftrightarrow B$ is true *iff* either both A and B are true or both are false.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
$false$	$false$	$true$	$false$	$false$	$true$	$true$
$false$	$true$	$true$	$false$	$true$	$true$	$false$
$true$	$false$	$false$	$false$	$true$	$false$	$false$
$true$	$true$	$false$	$true$	$true$	$true$	$true$

Figure 7.8 Truth tables for the five logical connectives. To use the table to compute, for example, the value of $P \vee Q$ when P is true and Q is false, first look on the left for the row where P is $true$ and Q is $false$ (the third row). Then look in that row under the $P \vee Q$ column to see the result: $true$.

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.

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

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

  $clauses \leftarrow$ the set of clauses in the CNF representation of s
  $symbols \leftarrow$ 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** $false$
  $P, value \leftarrow$ FIND-PURE-SYMBOL($symbols, clauses, model$)
  if P is non-null **then return** DPLL($clauses, symbols - P, model \cup \{P$=$value\}$)
  $P, value \leftarrow$ FIND-UNIT-CLAUSE($clauses, model$)
  if P is non-null **then return** DPLL($clauses, symbols - P, model \cup \{P$=$value\}$)
  $P \leftarrow$ FIRST($symbols$); $rest \leftarrow$ REST($symbols$)
  return DPLL($clauses, rest, model \cup \{P$=$true\}$) **or**
    DPLL($clauses, rest, model \cup \{P$=$false\}$))

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.

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 $Q \Rightarrow P$.

First-Order Logic

The second dialect of logic, **first-order logic (FOL)**, is more expressive than propositional logic and uses objects as its basic components. With first-order logic we can describe relationships between objects and apply functions to them. Each object is represented by a **constant symbol**, each relationship by a **predicate symbol**, and each function by a **function symbol**.

The following table summarizes the first order logic syntax.

$$
\begin{array}{rcl}
\textit{Sentence} & \rightarrow & \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} & \rightarrow & \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} & \rightarrow & (\ \textit{Sentence}\ ) \mid [\ \textit{Sentence}\ ] \\
& \mid & \neg\ \textit{Sentence} \\
& \mid & \textit{Sentence} \wedge \textit{Sentence} \\
& \mid & \textit{Sentence} \vee \textit{Sentence} \\
& \mid & \textit{Sentence} \Rightarrow \textit{Sentence} \\
& \mid & \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
& \mid & \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\\
\textit{Term} & \rightarrow & \textit{Function}(\textit{Term}, \ldots) \\
& \mid & \textit{Constant} \\
& \mid & \textit{Variable} \\
\\
\textit{Quantifier} & \rightarrow & \forall \mid \exists \\
\textit{Constant} & \rightarrow & A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} & \rightarrow & a \mid x \mid s \mid \cdots \\
\textit{Predicate} & \rightarrow & \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} & \rightarrow & \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} & : & \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{array}
$$

Figure 8.3 The syntax of first-order logic with equality, specified in Backus–Naur form (see page 1060 if you are not familiar with this notation). Operator precedences are specified, from highest to lowest. The precedence of quantifiers is such that a quantifier holds over everything to the right of it.

Terms in first-order logic are logical expressions that refer to an object. The simplest form of terms are constant symbols. However we don't want to define distinct constant symbols for every possible object. For example, if we want to refer to John's left leg and Richard's left leg, we can do so by using **function symbols** like $Leftleg(John)$ and $Leftleg(Richard)$. Function symbols are just another way to name objects and are not actual functions.

Atomic sentences in first-order logic are descriptions of relationships between objects, and are true if the relationship holds. An example of an atomic sentence is $Brother(John, Richard)$ which is formed by a predicate symbol followed by a list of terms inside the parentheses. **Complex sentences** of first order logic are analogous to those in propositional logic and are atomic sentences connected by logical connectives.

Naturally we would like ways to describe entire collections of objects. For this we use **quantifiers**. The **universal quantifier** $\forall$, has the meaning "for all," and the **existential quantifier** $\exists$, has the meaning "there exists."

For example, if the set of objects in our world is the set of all debates, the sentence $\forall a,\ \mathit{TwoSides}(a)$ could be translated as "there are two sides to every debate". If the set of objects in our world is people, the sentence $\forall x, \exists y,\ \mathit{SoulMate}(x,y)$ would mean "for all people, there is someone out there who is their soulmate. The **anonymous variables** a, x, y are stand-ins for objects, and can be **substituted** for actual objects, for example, substituting Laura for x into our second example would result in a statement that "there is someone out there for Laura."

The universal and existential quantifiers are convenient ways to express a conjunction or disjunction, respectively, over all objects. It follows that they also obey De Morgan's laws (note the analogous relationship between conjunctions and disjunctions):

$$\begin{array}{llll} \forall x\ \neg P & \equiv & \neg\exists x\ P & \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\ \neg\forall x\ P & \equiv & \exists x\ \neg P & \qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x\ P & \equiv & \neg\exists x\ \neg P & \qquad P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x\ P & \equiv & \neg\forall x\ \neg P & \qquad P \vee Q \equiv \neg(\neg P \wedge \neg Q) \end{array}$$

Finally, we use the **equality symbol** to signify that two symbols refer to the same object. For example, the incredible sentence $(\mathit{Wife}(\mathit{Einstein}) = \mathit{FirstCousin}(\mathit{Einstein}) \wedge \mathit{Wife}(\mathit{Einstein}) = \mathit{SecondCousin}(\mathit{Einstein}))$ is true!

Unlike with propositional logic, where a model was an assignment of true or false to all proposition symbols, a model in first-order logic is a mapping of all constant symbols to objects, predicate symbols to relations between objects, and function symbols to functions of objects. A sentence is true under a model if the relations described by the sentence are true under the mapping. While the number of models of a propositional logical system is always finite, there may be an infinite number of models of a first order logical system if the number of objects is unconstrained.

These two dialects of logic allow us to describe and think about the world in different ways. With propositional logic, we model our world as a set of symbols that are true or false. Under this assumption, we can represent a possible world as a vector, with a 1 or 0 for every symbol. This binary view of the world is what is known as a **factored representation**. With first-order logic, our world consists of objects that relate to one another. This second object-oriented view of the world is known as a **structured representation**, is in many ways more expressive and is more closely aligned with the language we naturally use to speak about the world.

First Order Logical Inference

With first order logic we formulate inference exactly the same way. We'd like to find out if $KB \models q$, that is if q is true in all models under which KB is true. One approach to finding a solution is **propositionalization** or translating the problem into propositional logic so that it can be solved with techniques we have already laid out. Each universal (existential) quantifier sentence can be converted to a conjunction (disjunction) of sentences with a clause for each possible object that could be substituted in for the variable. Then, we can use a SAT solver, like DPLL or Walk-SAT, (un)satisfiability of $(KB \wedge \neg q)$.

One problem with this approach is there are an infinite number of substitutions that we could make, since there is no limit to how many times we can apply a function to a symbol. For example, we can nest the function $\mathit{Classmate}(\cdots \mathit{Classmate}(\mathit{Classmate}(\mathit{Austen}))\cdots)$ as many times as we'd like, until we reference the whole school. Luckily, a theorem proved by Jacques Herbrand (1930) tells us that if a sentence is entailed

by a knowledge base that there is a proof involving just a *finite* subset of the propositionalized knowledge base. Therefore, we can try iterating through finite subsets, specifically searching via iterative deepening through nested function applications, i.e. first search through substitutions with constant symbols, then substitutions with $Classmate(Austen)$, then substitutions with $Classmate(Classmate(Austen))$, ...

Another approach is to directly do inference with first-order logic, also known as **lifted inference**. For example, we are given $(\forall x\, HasAbsolutePower(x) \wedge Person(x) \Rightarrow Corrupt(x)) \wedge Person(John) \wedge HasAbsolutePower(John)$ ("absolute power corrupts absolutely"). We can infer $Corrupt(John)$ by substituting x for $John$. This rule is known as **Generalized Modus Ponens**. The forward chaining algorithm for first order logic repeatedly applies generalized Modus Ponens and substitution to infer q or show that it cannot be inferred.

Logical Agents

Now that we understand how to formulate what we know and how to reason with it, we will talk about how to incorporate the power of deduction into our agents. One obvious ability an agent should have is the ability to figure out what state it is in, based on a history of observations and what it knows about the world (**state-estimation**). For example, if we told the agent that the air starts to shimmer near pools of lava and it observed that the air right before it is shimmering, it could infer that danger is nearby.

To incorporate its past observations into an estimate of where it currently is, an agent will need to have a notion of time and transitions between states. We call state attributes that vary with time **fluents** and write a fluent with an index for time, e.g. Hot^t = the air is hot at time t. The air should be hot at time t if something causes the air to be hot at that time, or the air was hot at the previous time and no action occurred to change it. To represent this fact we can use the general form of the **successor-state axiom** below:

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t)$$

In our world, the transition could be formulated as $Hot^{t+1} \Leftrightarrow StepCloseToLava^t \vee (Hot^t \wedge \neg StepAwayFromLava^t)$.

Having written out the rules of the world in logic, we can now actually do planning by checking the satisfiability of some logic proposition! To do this, we construct a sentence that includes information about the initial state, the transitions (successor-state axioms), and the goal. (e.g. $InOasis^T \wedge Alive^T$ encodes the objective of surviving and ending up in the oasis by time T). If the rules of the world have been properly formulated, then finding a satisfying assignment to all the variables will allow us to extract a sequence of actions that will carry the agent to the goal.

Summary

In this note, we introduced the concept of logic which knowledge-based agents can use to reason about the world and make decisions. We introduced the language of logic, its syntax and the standard logical equivalences. Propositional logic is a simple language that is based on proposition symbols and logical connectives. First-order logic is a representation language more powerful than propositional logic. The syntax of first-order logic builds on that of propositional logic, using terms to represent objects and universal and existential quantifiers to make assertions.

We further described the DPLL algorithm used to check satisfiability (SAT problem) in propositional logic. It is a depth-first enumeration of possible models, using early termination, pure symbol heuristic and unit clause heuristic to improve performance. The forward chaining algorithm can be used for reasoning when our knowledge base consists solely of literals and implications in propositional logic.

Inference in first-order logic can be done directly by using rules like Generalized Modus Ponens or by propositionalization, which translates the problem into propositional logic and uses a SAT solver to draw conclusions.