UNIT-III

Knowledge and Reasoning

3.1 Building a Knowledge Base

Knowledge:-

Knowledge representation and reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to intelligent behavior of agents. It is also a way which describes how we can represent knowledge in artificial intelligence.

Reasoning:-

Reasoning: ... Or we can say, "Reasoning is a way to infer facts from existing data." It is a general process of thinking rationally, to find valid conclusions. In artificial intelligence, the reasoning is essential so that the machine can also think rationally as a human brain, and can perform like a human.

A knowledge-based system (KBS) is a form of artificial intelligence (AI) that aims to capture the knowledge of human experts to support decision-making. ... Some systems encode expert knowledge as rules and are therefore referred to as rule-based systems. Another approach, case-based reasoning, substitutes cases for rules.

3.2 Propositional Logic

Propositional Logic (sometimes called Propositional Calculus) is a branch of logic that defiens way to combine statements or sentences together.

In previous systems seen above there is no way to say that something is **not**. For example, **Not** CompetesInPremierLeauge(LiverpoolFC) which denotes that LiverpoolFC does not play in the premier leauge is not possible in previous systems.

You also cannot connect sentences using “or”. Today I will sleep **or** I will study.

A proposition is a statement that can be true or false, but not both at the same time.

3.2.1 Compound Propositional Statements

Given knowledge, which is the below paragraph:

The meeting can take place if all members have been informed in advance, and it is quorate. It is quorate provided that there are at least 15 people present. Members will have been informed in advance if there is not a postal strike. Therefore, if the meeting was cancelled, we conclude that there were fewer than 15 members present, or there was a postal strike

We can turn it into a propositional statement. Atomic propositions are created from statements, the simplest statements from the paragraph are shown below:
M: The meeting takes place
A: all members have been informed
P: There is a postal strike
Q: The meeting is quorate.
F: there are at least 15 members present

Remember, the above statement was changed into atomic propositions to allow the computer to read it easily.

We can turn this into a statement
If A and Q then M. If F then Q. If not P then A.

And we can conclude
If not M then not F or P.

Propositions may be combinded with other propositions to form compound propositions. The connects used to combind propositions together are:

$\neg$	not	negation	$(\sim)$
$\wedge$	and	conjunction	(& or .)
$\vee$	or	disjunction	(| or $+$)
$\Leftrightarrow$	if and only if	equivalence	$(\leftrightarrow)$
$\Rightarrow$	if . . . then	implication	$(\rightarrow)$

Some authors use different notations, these are given in brackets.

3.2.2 Propositional formulae

An atomic proposition is a proposition that cannot be broken down anymore. For example, the proposition
I really like pizza or I hate bridges

Contains a connective and 2 propositional statements. An atomic proposition is one that has no sub-propositions and no connectives such as “I like pizza”. If P and Q are atomic propositions then the following can apply:

P ^ Q

P v Q

P <=> Q

P => Q

The meanings of each symbol are

P ^ Q = P and Q

p v Q = P or Q

P <=> Q = P is true if and only if Q is true

P => Q = P entails, implies, or has a consequence Q.

¬P = Not P.

If P is a propositional formula then ¬P is a propositional formula as well.

3.2.3 Truth Values of Propositonal Statements

All propositional statements have a truth value, either True or False. An interpertation “i” gives a truth value to every atomic proposition. Symbolically this is shown as $I(p) \in \{0,1\}$. The value p can either be 0 (false) or 1 (true).

3.2.3.1 Truth Tables

A trurth table is a table showing the truth value of an atomic proposition under certain circumstances.

p	q	p∧q
T	T	T
T	F	F
F	T	F
F	F	F

Seen above is an example of a truth table. T represents True, F represents False.

3.2.3.2Conjunction

Conjunction is the name used to mean “and” in propositional logic. True and True would equal True whereas True and False would equal False.

Below is a truth table for the statement (P ^ Q) which is (P and Q).

The conjunction $(P \wedge Q)$ of P and Q.

both P and Q are true

Truth table:

P	Q	$(P \wedge Q)$
1	1	1
1	0	0
0	1	0
0	0	0

Below are propositional statements in example format.

p: Logic is easy.

q: I eat toast.

$(p \wedge q)$: Logic is easy and I eat toast

$(\neg p \wedge q)$: Logic is not easy and I eat toast

$(\neg p \wedge \neg q)$: Logic is not easy and I do not eat toast

3.2.3.3 Disjunction

Disjunction is the name used to mean “or” in propositional logic. True or False would equate to True.

Below is an image showing the disjunction of (P v Q) otherwise known as (P or Q) and the truth table for it.

The disjunction $(P \vee Q)$ of P and Q

at least one of P and Q is true

Truth table:

P	Q	$(P \vee Q)$
1	1	1
1	0	1
0	1	1
0	0	0

Below is an image showing different examples of the disjunction

p: I'll have tea.

q: I'll have coffee.

$(p \vee q)$: I'll have tea or I'll have coffee

3.2.3.4 Equivilance

Given P < = > Q, True is only returned if P is the same as Q. In other words, they have to hold the same value for it to be true.

Below is the truth table for equivilance.

The equivalence $(P \Leftrightarrow Q)$ of P and Q

P and Q take the same truth value

Truth table:

P	Q	$(P \Leftrightarrow Q)$
1	1	1
1	0	0
0	1	0
0	0	1

Below is an image showing different examples of equivilance

p: You can take a flight.
q: You buy a ticket.
$(p \Leftrightarrow q)$: You can take a flight if and only if you buy a ticket

3.2.3.5 Implication

Logical Implication is a type of relationship between two statements where one implies the other. The statement can be read as "logically implies" or just "Implies". If A and B represent statements then A implies B or A → B.

Many students of logic find implication the hardest, so let's start with what implication is not.

- **It doesn't mean "suggests".**

If you say Bill doesn't want to go to dinner with you friday night implies Bill doesn't like you, this may be a perfectly good statement in English but in logic it's not a good statement. All it implies is some sort of suggestion of something, such as Bill not liking you.

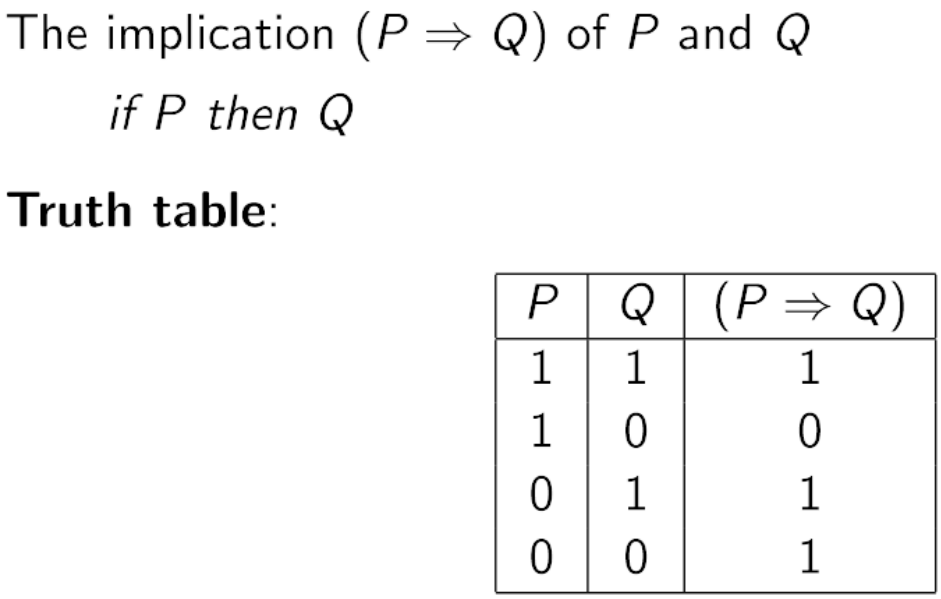
- **It doesn't mean "causes"**

There is no caustation in implication. One could say that green is a colour implies January is a month and that is a logical true implication even though there is no causaility in that. Just because green is a colour does not cause January to be a month.

- **Statements with implications must always hold a truth value**

We have to give truth values even in cases where you might not think it needs a truth value

Below is an image showing the truth table of implications.



P	Q	$(P \Rightarrow Q)$
1	1	1
1	0	0
0	1	1
0	0	1

One might care to notice that if Q is false then the statement is true.

Logical implication doesn't work both ways. A → B does not equal B → A. However, the negation of the statement is equal to the original statement.
A -> B = -A -> -B

Below is an image showing different examples for implication

Examples

p: You study hard.
q: You get an A.
$(p \Rightarrow q)$: If you study hard, you get an A

p: I am king of England.
q: $2 \times 2 = 5$.
$(p \Rightarrow q)$: If I am king of England, then $2 \times 2 = 5$

$(q \Rightarrow p)$: If $2 \times 2 = 5$, then I am king of England

p: I am king of England.
q: $2 \times 2 = 4$.
$(p \Rightarrow q)$: If I am king of England, then $2 \times 2 = 4$

$(q \Rightarrow p)$: If $2 \times 2 = 4$, then I am king of England

Logical implication when shown in this format is weak. A → B has little meaning for readers. Instead I invite the reader to look at a real world example of implication.

It is often possible to assert a universal statement using logical implication, as seen below.
$x > 2 \wedge x$ is prime $\Longrightarrow x$ is odd.

X is more than 2 and X is prime implies that X is odd. What makes this statement useful is that there isn't a single number that holds for X where it is greater than 2 and prime, but **not** odd.

Truth under an interpretation

Given an interpretation (that is, an assignment of meaning to the symbols. Often giving meaning such as words or numbers to algebraic symbols) we can compute the truth value of any proposition under I (interpretation).
I(P) = 1, therefore under this interpretation P is true.
I(p) = 0, therefore under this interpretation P is false.

Satisfiability

A proposition is considered satisfiable if there exists an interpretation under which it is true. The formula
I(p∧¬p)

Is not satisfiable because it will always result in 0 (try making p 1 or 0 yourself)

Again, a proposition is satisfiable if there exists an input that results in the proposition being true.

I(P) is satisfiable if for some interpretation P is True. In this case, P is true when it is equal to 1, or it is true when it is true.

**** There are 2^n interpretations if P has n propositional atoms. ****

Thus to generate a truth table a table with 2^n rows is needed. This is not practical and will take forever to compute, even with a small number of interpretations such as 100. This will cause a combinatorial explosion.

Knowledge Bases

Let's go back to the knowledge base question first purposed at the start of this article:
The meeting can take place if all members have beeninformed in advance, and it is quorate. It is quorateprovided that there are at least 15 people present.Members will have been informed in advance if there isnot a postal strike. Therefore, if the meeting wascancelled, we conclude that there were fewer than 15members present, or there was a postal strike

Then that gives us the propositional statements:
m : "the meeting takes place"a: "all members have been informed"p: "there is a postal strike"q: "the meeting is quorate"f: "there are at least 15 members present"

Which we can formalise as:
((a∧q)⇒m),(f⇒q),(¬p⇒a)

This was simply to remind you of knowledge bases.

Propositional Knowledge Bases and Reasoning

A propositional knowledge base is a finite set of propositional formulae, just like normal knowledge bases.

Suppose a propositional knowledge base, X, is given to us. Then a propositional formula, P, follows from X, if the following holds for every interpretation, I.
If $I(Q) = 1$ for all $Q \in X$, then $I(P) = 1$.

This can be read as:
$X \models P$

Which just means that P is provable from the knowledge base X. The proposition is provable from the knowledge base X. Propositional knowledge bases and reasoning is pretty much the same as ordinary knowledge base and reasoning discussed earlier.

3.3 First Order Logic

- The prepositional logic only deals with the facts, that may be true or false.
- The first order logic assumes that the world contains objects, relations and functions.

Propositional logic provides a good start at describing the general principles of logical reasoning, but it does not go far enough. Some of the limitations are apparent even in the "Malice and Alice" example from Chapter 2. Propositional logic does not give us the means to express a general principle that tells us that if Alice is with her son on the beach, then her son is with Alice; the general fact that no child is older than his or her parent; or the general fact that if someone is alone, they are not with someone else. To express principles like these, we need a way to talk about objects and individuals, as well as their properties and the relationships between them. These are exactly what is provided by a more expressive logical framework known as *first-order logic*, which will be the topic of the next few chapters.

Functions, Predicates, and Relations

Consider some ordinary statements about the natural numbers:

- Every natural number is even or odd, but not both.
- A natural number is even if and only if it is divisible by two.
- If some natural number, xx, is even, then so is x2x2.
- A natural number xx is even if and only if x+1x+1 is odd.
- Any prime number that is greater than 2 is odd.
- For any three natural numbers xx, yy, and zz, if xx divides yy and yy divides zz, then xx divides zz.

These statements are true, but we generally do not think of them as *logically valid*: they depend on assumptions about the natural numbers, the meaning of the terms "even" and "odd," and so on. But once we accept the first statement, for example, it seems to be a logical consequence that the number of stairs in the White House is either even or odd, and, in particular, if it is not even, it is odd. To make sense of inferences like these, we need a logical system that can deal with objects, their properties, and relations between them.

Rather than fix a single language once and for all, first-order logic allows us to specify the symbols we wish to use for any given domain of interest. In this section, we will use the following running example:

- The domain of interest is the natural numbers, NN.
- There are objects, 00, 11, 22, 33, ….
- There are functions, addition and multiplication, as well as the square function, on this domain.
- There are predicates on this domain, “even,” “odd,” and “prime.”
- There are relations between elements of this domain, “equal,” “less than”, and “divides.”

For our logical language, we will choose symbols 1, 2, 3, addadd, mulmul, squaresquare, eveneven, oddodd, primeprime, ltlt, and so on, to denote these things. We will also have variables xx, yy, and zz ranging over the natural numbers. Note all of the following.

- Functions can take different numbers of arguments: if xx and yy are natural numbers, it makes sense to write mul(x,y)mul(x,y) and square(x)square(x). So mulmul takes two arguments, and squaresquare takes only one.
- Predicates and relations can also be understood in these terms. The predicates even(x)even(x) and prime(x)prime(x) take one argument, while the binary relations divides(x,y)divides(x,y) and lt(x,y)lt(x,y) take two arguments.
- Functions are different from predicates! A function takes one or more arguments, and returns a *value*. A predicate takes one or more arguments, and is either true or false. We can think of predicates as returning propositions, rather than values.
- In fact, we can think of the constant symbols 1,2,3,…1,2,3,… as special sorts of function symbols that take zero arguments. Analogously, we can consider the predicates that take zero arguments to be the constant logical values, ⊤⊤ and ⊥⊥.
- In ordinary mathematics, we often use “infix” notation for binary functions and relations. For example, we usually write x×yx×y or x⋅yx⋅y instead of mul(x,y)mul(x,y), and we write x<yx<y instead of lt(x,y)lt(x,y). We will use these conventions when writing proofs in natural deduction, and they are supported in Lean as well.
- We will treat the equality relation, x=yx=y, as a special binary relation that is included in every first-order language.

First-order logic allows us to build complex expressions out of the basic ones. Starting with the variables and constants, we can use the function symbols to build up compound expressions like these:

- x+y+zx+y+z
- (x+1)×y×y(x+1)×y×y
- square(x+y×z)square(x+y×z)

Such expressions are called “terms.” Intuitively, they name objects in the intended domain of discourse.

Now, using the predicates and relation symbols, we can make assertions about these expressions:

- even(x+y+z)even(x+y+z)
- prime((x+1)×y×y)prime((x+1)×y×y)
- square(x+y×z)=wsquare(x+y×z)=w
- x+y<zx+y<z

Even more interestingly, we can use propositional connectives to build compound expressions like these:

- even(x+y+z)∧prime((x+1)×y×y)even(x+y+z)∧prime((x+1)×y×y)
- ¬(square(x+y×z)=w)∨x+y<z¬(square(x+y×z)=w)∨ x+y<z
- x<y∧ even(x)∧ even(y)→x+1<yx<y∧ even(x)∧ even(y)→x+1<y

The second one, for example, asserts that either (x+yz)2(x+yz)2 is not equal to ww, or x+yx+y is less than zz. Remember, these are expressions in symbolic logic; in ordinary mathematics, we would express the notions using words like “is even” and “if and only if,” as we did above. We will use notation like this whenever we are in the realm of symbolic logic, for example, when we write proofs in natural deduction. Expressions like these are called *formulas*. In contrast to terms, which name things, formulas *say things*; in other words, they make assertions about objects in the domain of discourse.

3.4 Situation Calculus:-

The idea behind situation calculus is that (reachable) states are definable in terms of the actions required to reach them. These reachable states are called situations. What is true in a situation can be defined in terms of relations with the situation as an argument. Situation calculus can be seen as a relational version of the feature-based representation of actions.
Here we only consider single agents, a fully observable environment, and deterministic actions.
Situation calculus is defined in terms of situations. A situation is either

- *init*, the initial situation, or
- *do(A,S)*, the situation resulting from doing action *A* in situation *S*, if it is possible to do action *A* in situation *S*.

Example 14.1: Consider the domain of Figure 3.1. Suppose in the initial situation, *init*, the robot, Rob, is at location *o109* and there is a key *k1* at the mail room and a package at *storage*.

do(move(rob,o109,o103), init)

is the situation resulting from Rob moving from position *o109* in situation *init* to position *o103*. In this situation, Rob is at *o103*, the key *k1* is still at *mail*, and the package is at *storage*.
The situation

do(move(rob,o103,mail),

do(move(rob,o109,o103),

init))

is one in which the robot has moved from position *o109* to *o103* to *mail* and is currently at mail. Suppose Rob then picks up the key, *k1*. The resulting situation is

do(pickup(rob,k1),

do(move(rob,o103,mail),

do(move(rob,o109,o103),

init))).

In this situation, Rob is at position *mail* carrying the key *k1*.
A situation can be associated with a state. There are two main differences between situations and states:

- Multiple situations may refer to the same state if multiple sequences of actions lead to the same state. That is, equality between situations is not the same as equality between states.
- Not all states have corresponding situations. A state is reachable if a sequence of actions exists that can reach that state from the initial state. States that are not reachable do not have a corresponding situation.

Some *do(A,S)* terms do not correspond to any state. However, sometimes an agent must reason about such a (potential) situation without knowing if *A* is possible in state *S*, or if *S* is possible.
Example 14.2: The term *do(unlock(rob,door1),init)* does not denote a state at all, because it is not possible for Rob to unlock the door when Rob is not at the door and does not have the key.

A static relation is a relation for which the truth value does not depend on the situation; that is, its truth value is unchanging through time. A dynamic relation is a relation for which the truth value depends on the situation. To represent what is true in a situation, predicate symbols denoting dynamic relations have a situation argument so that the truth can depend on the situation. A predicate symbol with a situation argument is called a fluent.
Example 14.3: The relation *at(O,L,S)* is true when object *O* is at location *L* in situation *S*. Thus, *at* is a fluent.

The atom

at(rob,o109,init)

is true if the robot *rob* is at position *o109* in the initial situation. The atom

at(rob,o103,do(move(rob,o109,o103), init))

is *true* if robot *rob* is at position *o103* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation. The atom

at(k1,mail,do(move(rob,o109,o103), init))

is *true* if *k1* is at position *mail* in the situation resulting from *rob* moving from position *o109* to position *o103* from the initial situation.
A dynamic relation is axiomatized by specifying the situations in which it is true. Typically, this is done inductively in terms of the structure of situations.

- Axioms with *init* as the situation parameter are used to specify what is true in the initial situation.
- A primitive relation is defined by specifying when it is true in situations of the form *do(A,S)* in terms of what is true in situation *S*. That is, primitive relations are defined in terms of what is true at the previous situation.
- A derived relation is defined using clauses with a variable in the situation argument. The truth of a derived relation in a situation depends on what else is true in the same situation.
- Static relations are defined without reference to the situation.

Example 14.4: Suppose the delivery robot, Rob, is in the domain depicted in Figure 3.1. Rob is at location *o109*, the parcel is in the storage room, and the key is in the mail room. The following axioms describe this initial situation:

at(rob,o109,init).
at(parcel,storage,init).
at(k1,mail,init).

The *adjacent* relation is a dynamic, derived relation defined as follows:
adjacent(o109,o103,S).
adjacent(o103,o109,S).

adjacent(o109,storage,S).
adjacent(storage,o109,S).
adjacent(o109,o111,S).
adjacent(o111,o109,S).
adjacent(o103,mail,S).
adjacent(mail,o103,S).
adjacent(lab2,o109,S).
adjacent(P_1,P_2,S)←

between(Door,P_1,P_2)∧
unlocked(Door,S).

Notice the free *S* variable; these clauses are true for all situations. We cannot omit the *S* because which rooms are adjacent depends on whether a door is unlocked. This can change from situation to situation.
The *between* relation is static and does not require a situation variable:
between(door1,o103,lab2).

We also distinguish whether or not an agent is being carried. If an object is not being carried, we say that the object is sitting at its location. We distinguish this case because an object being carried moves with the object carrying it. An object is at a location if it is sitting at that location or is being carried by an object at that location. Thus, *at* is a derived relation:
at(Ob,P,S)←

sitting_at(Ob,P,S).
at(Ob,P,S)←

carrying(Ob1,Ob,S)∧
at(Ob1,P,S).

Note that this definition allows for Rob to be carrying a bag, which, in turn, is carrying a book.
The precondition of an action specifies when it is possible to carry out the action. The relation *poss(A,S)* is true when action *A* is possible in situation *S*. This is typically a derived relation.
Example 14.5: An agent can always put down an object it is carrying:

poss(putdown(Ag,Obj),S) ←
carrying(Ag,Obj,S).

For the *move* action, an autonomous agent can move from its current position to an adjacent position:
poss(move(Ag,P_1,P_2),S) ←

autonomous(Ag) ∧
adjacent(P_1,P_2,S)∧
sitting_at(Ag,P_1,S) .

The precondition for the unlock action is more complicated. The agent must be at the correct side of the door and carrying the appropriate key:

$poss(unlock(Ag,Door),S) \leftarrow$

$autonomous(Ag) \wedge$

$between(Door,P_1,P_2) \wedge$

$at(Ag,P_1,S) \wedge$

$opens(Key,Door) \wedge$

$carrying(Ag,Key,S).$

We do not assume that the *between* relation is symmetric. Some doors can only open one way. We define what is true in each situation recursively in terms of the previous situation and of what action occurred between the situations. As in the feature-based representation of actions, causal rules specify when a relation becomes true and frame rules specify when a relation remains true.

3.6 Planning:-

What is planning in AI?

- The planning in Artificial Intelligence is about the decision making tasks performed by the robots or computer programs to achieve a specific goal.
- The execution of planning is about choosing a sequence of actions with a high likelihood to complete the specific task.

Blocks-World planning problem

- The blocks-world problem is known as **Sussman Anomaly.**
- Noninterleaved planners of the early 1970s were unable to solve this problem, hence it is considered as anomalous.
- When two subgoals G1 and G2 are given, a noninterleaved planner produces either a plan for G1 concatenated with a plan for G2, or vice-versa.
- In blocks-world problem, three blocks labeled as 'A', 'B', 'C' are allowed to rest on the flat surface. The given condition is that only one block can be moved at a time to achieve the goal.
- The start state and goal state are shown in the following diagram.

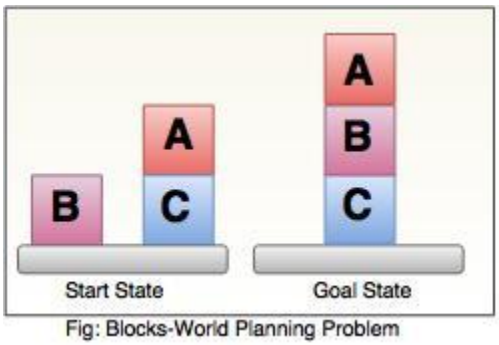


Fig: Blocks-World Planning Problem

Components of Planning System

The planning consists of following important steps:

- Choose the best rule for applying the next rule based on the best available heuristics.
- Apply the chosen rule for computing the new problem state.
- Detect when a solution has been found.
- Detect dead ends so that they can be abandoned and the system's effort is directed in more fruitful directions.
- Detect when an almost correct solution has been found.

Goal stack planning

This is one of the most important planning algorithms, which is specifically used by **STRIPS.**

- The stack is used in an algorithm to hold the action and satisfy the goal. A knowledge base is used to hold the current state, actions.
- Goal stack is similar to a node in a search tree, where the branches are created if there is a choice of an action.

The important steps of the algorithm are as stated below:

i. Start by pushing the original goal on the stack. Repeat this until the stack becomes empty. If stack top is a compound goal, then push its unsatisfied subgoals on the stack.
ii. If stack top is a single unsatisfied goal then, replace it by an action and push the action's precondition on the stack to satisfy the condition.
iii. If stack top is an action, pop it from the stack, execute it and change the knowledge base by the effects of the action.
iv. If stack top is a satisfied goal, pop it from the stack.

Non-linear planning

This planning is used to set a goal stack and is included in the search space of all possible subgoal orderings. It handles the goal interactions by interleaving method.

Advantage of non-Linear planning

Non-linear planning may be an optimal solution with respect to plan length (depending on search strategy used).

Disadvantages of Nonlinear planning

- It takes larger search space, since all possible goal orderings are taken into consideration.
- Complex algorithm to understand.

Algorithm

1. Choose a goal 'g' from the goalset
2. If 'g' does not match the state, then

- Choose an operator 'o' whose add-list matches goal g
- Push 'o' on the opstack
- Add the preconditions of 'o' to the goalset

3. While all preconditions of operator on top of opstack are met in state

- Pop operator o from top of opstack
- state = apply(o, state)
- plan = [plan; o]

3.7 Partial-Order Planning

The forward and regression planners enforce a total ordering on actions at all stages of the planning process. The CSP planner commits to the particular time that the action will be carried out. This means that those planners have to commit to an ordering of actions that cannot occur concurrently when adding them to a partial plan, even if there is no particular reason to put one action before another.

The idea of a **partial-order planner** is to have a partial ordering between actions and only commit to an ordering between actions when forced. This is sometimes also called a **non-linear planner**, which is a misnomer because such planners often produce a linear plan.

A partial ordering is a less-than relation that is transitive and asymmetric. A **partial-order plan** is a set of actions together with a partial ordering, representing a "before" relation on actions, such that any total ordering of the actions, consistent with the partial ordering, will solve the goal from the initial state. Write $act_0 < act_1$ if action act_0 is before action act_1 in the partial order. This means that action act_0 must occur before action act_1.

For uniformity, treat *start* as an action that achieves the relations that are true in the initial state, and treat *finish* as an action whose precondition is the goal to be solved. The pseudoaction *start* is before every other action, and *finish* is after every other action. The use of these as actions means that the algorithm does not require special cases for the initial situation and for the goals. When the preconditions of *finish* hold, the goal is solved.

An action, other than *start* or *finish*, will be in a partial-order plan to achieve a precondition of an action in the plan. Each precondition of an action in the plan is either true in the initial state, and so achieved by *start*, or there will be an action in the plan that achieves it.

We must ensure that the actions achieve the conditions they were assigned to achieve. Each precondition P of an action act_1 in a plan will have an action act_0 associated with it such that act_0 achieves precondition P for act_1. The triple $\langle act_0,P,act_1\rangle$ is a **causal link**. The partial order specifies that action act_0 occurs before action act_1, which is written as $act_0 < act_1$. Any other action A that makes P false must either be before act_0 or after act_1.

Informally, a partial-order planner works as follows: Begin with the actions *start* and *finish* and the partial order *start* < *finish*. The planner maintains an agenda that is a set of $\langle P,A\rangle$ pairs, where A is an action in the plan and P is an atom that is a precondition of A that must be achieved. Initially the agenda contains pairs $\langle G,finish\rangle$, where G is an atom that must be true in the goal state.

At each stage in the planning process, a pair $\langle G,act_1\rangle$ is selected from the agenda, where P is a precondition for action act_1. Then an action, act_0, is chosen to achieve P. That action is either already in the plan - it could be the *start* action, for example - or it is a new action that is added to the plan. Action act_0 must happen before act_1 in the partial order. It adds a causal link that records that act_0 achieves P for action act_1. Any action in the plan that deletes P must happen either before act_0 or after act_1. If act_0 is a new action, its preconditions are added to the agenda, and the process continues until the agenda is empty.

This is a non-deterministic procedure. The "choose" and the "either ...or ..." form choices that must be searched over. There are two choices that require search:

- which action is selected to achieve G and
- whether an action that deletes G happens before act_0 or after act_1.

---

non-deterministic procedure *PartialOrderPlanner*(*Gs*)

Inputs
    Gs: set of atomic propositions to achieve
Output
    linear plan to achieve *Gs*
Local
    Agenda: set of $\langle P,A\rangle$ pairs where P is atom and A an action
    Actions: set of actions in the current plan
    Constraints: set of temporal constraints on actions
    CausalLinks: set of $\langle act_0,P,act_1\rangle$ triples

Agenda $\leftarrow \{\langle G,finish\rangle : G \in Gs\}$
Actions $\leftarrow \{start,finish\}$
Constraints $\leftarrow \{start<finish\}$
CausalLinks $\leftarrow \{\}$
repeat
    select and remove $\langle G,act_1\rangle$ from *Agenda*

```
17:             either
18:                 choose act0 ∈Actions such that act0 achieves G
19:             or
20:                 choose act0 ∉Actions such that act0 achieves G
21:                 Actions ←Actions ∪{act0}
22:                 Constraints ←add_const(start<act0,Constraints)
23:                 for each CL∈CausalLinks do
24:                     Constraints ←protect(CL,act0,Constraints)
25:
26:                 Agenda ←Agenda ∪{⟨P,act0⟩: P is a precondition of act0 }
27:
28 :            Constraints ←add_const(act0<act1,Constraints)
29:             CausalLinks ∪ {⟨act0,G,act1⟩}
30:             for each A∈Actions do
31:                 Constraints ←protect(⟨act0,G,act1⟩,A,Constraints)
32:
33:         until Agenda={}
34:         return total ordering of Actions consistent with Constraints
```

Figure 8.5: Partial-order planner

The function *add_const(act₀<act₁,Constraints)* returns the constraints formed by adding the constraint $act_0 < act_1$ to *Constraints*, and it fails if $act_0 < act_1$ is incompatible with *Constraints*. There are many ways this function can be implemented.
The function *protect(⟨act₀,G,act₁⟩,A,Constraints)* checks

whether $A \neq act_0$ and $A \neq act_1$ and *A* deletes *G*. If so, it returns either $\{ A < act_0 \} \cup Constraints$ or $\{ act_1 < A \} \cup Constraints$. This is a non-deterministic choice that is searched over. Otherwise it returns *Constraints*.

Example: Consider the goal $\neg swc \wedge \neg mw$, where the initial state contains *RLoc=lab*, *swc*, $\neg rhc$, *mw*, $\neg rhm$.
Initially the agenda is

⟨$\neg swc$,*finish*⟩,⟨$\neg mw$,*finish*⟩.

Suppose ⟨$\neg swc$,*finish*⟩ is selected and removed from the agenda. One action exists that can achieve $\neg swc$, namely deliver coffee, *dc*, with preconditions *off* and *rhc*. At the end of the **repeat** loop, *Agenda* contains

⟨*off*,*dc*⟩,⟨*rhc*,*dc*⟩,⟨$\neg mw$,*finish*⟩.

Constraints is {*start*<*finish*, *start* < *dc*, *dc* <*finish*}. There is one causal link, ⟨*dc*, $\neg swc$,*finish*⟩. This causal link means that no action that undoes $\neg swc$ is allowed to happen after *dc* and before *finish*.

Suppose ⟨¬*mw,finish*⟩ is selected from the agenda. One action exists that can achieve this, *pum*, with preconditions *mw* and *RLoc=mr*. The causal link ⟨*pum*, ¬*mw,finish*⟩ is added to the set of causal links; ⟨*mw,pum*⟩ and ⟨*mr,pum*⟩ are added to the agenda.
Suppose ⟨*mw,pum*⟩ is selected from the agenda. The action *start* achieves *mw*, because *mw* is true initially. The causal link ⟨*start,mw,pum*⟩ is added to the set of causal links. Nothing is added to the agenda.
At this stage, there is no ordering imposed between *dc* and *pum*.
Suppose ⟨*off,dc*⟩ is removed from the agenda. There are two actions that can achieve *off*: *mc_cs* with preconditions *cs*, and *mcc_lab* with preconditions *lab*. The algorithm searches over these choices. Suppose it chooses *mc_cs*. Then the causal link ⟨*mc_cs,off,dc*⟩ is added.
The first violation of a causal link occurs when a move action is used to achieve ⟨*mr,pum*⟩. This action violates the causal link ⟨*mc_cs,off,dc*⟩, and so must happen after *dc* (the robot goes to the mail room after delivering coffee) or before *mc_cs*.

The preceding algorithm has glossed over one important detail. It is sometimes necessary to perform some action more than once in a plan. The preceding algorithm will not work in this case, because it will try to find a partial ordering with both instances of the action occurring at the same time. To fix this problem, the ordering should be between action instances, and not actions themselves. To implement this, assign an index to each instance of an action in the plan, and the ordering is on the action instance indexes and not the actions themselves. This is left as an exercise.

3.8 Uuncertain Knowledge and Reasoning

Why Reason Probabilistically?

- In many problem domains it isn't possible to create complete, consistent models of the world. Therefore agents (and people) must act in uncertain worlds (which the real world is).
- Want an agent to make rational decisions even when there is not enough information to prove that an action will work.
- Some of the reasons for reasoning under uncertainty:
    - **True uncertainty**. E.g., flipping a coin.
    - **Theoretical ignorance**. There is no complete theory which is known about the problem domain. E.g., medical diagnosis.
    - **Laziness**. The space of relevant factors is very large, and would require too much work to list the complete set of antecedents and consequents. Furthermore, it would be too hard to use the enormous rules that resulted.
    - **Practical ignorance**. Uncertain about a particular individual in the domain because all of the information necessary for that individual has not been collected.
- Probability theory will serve as the formal language for representing and reasoning with uncertain knowledge.

Representing Belief about Propositions

- Rather than reasoning about the truth or falsity of a proposition, reason about the belief that a proposition or event is true or false
- For each primitive proposition or event, attach a **degree of belief** to the sentence
- Use **probability theory** as a formal means of manipulating degrees of belief
- Given a proposition, A, assign a probability, P(A), such that 0 <= P(A) <= 1, where if A is true, P(A)=1, and if A is false, P(A)=0. Proposition A must be either true or false, but P(A) summarizes our degree of belief in A being true/false.
- Examples
    - P(Weather=Sunny) = 0.7 means that we believe that the weather will be Sunny with 70% certainty. In this case Weather is a random variable that can take on values in a domain such as {Sunny, Rainy, Snowy, Cloudy}.
    - P(Cavity=True) = 0.05 means that we believe there is a 5% chance that a person has a cavity. Cavity is a Boolean random variable since it can take on possible values *True* and *False*.
    - Example: P(A=a ^ B=b) = P(A=a, B=b) = 0.2, where A=My_Mood, a=happy, B=Weather, and b=rainy, means that there is a 20% chance that when it's raining my mood is happy.
- Obtaining and Interpreting Probabilities
  There are several senses in which probabilities can be obtained and interpreted, among them the following:
    - **Frequentist Interpretation**
      The probability is a property of a population of similar events. E.g., if set S = P union N, and P intersection N is the empty set, then the probability of an object

being in set P is |P|/|S|. Hence, in this interpretation probabilities come from experiments and determining the population associated with a given proposition.

- **Subjectivist Interpretation**
  A subjective degree of belief in a proposition or the occurrence of an event. E.g., the probability that you'll pass the Final Exam based on your own subjective evaluation of the amount of studying you've done and your understanding of the material. Hence, in this interpretation probabilities characterize the agent's beliefs.

- We will assume that in a given problem domain, the programmer and expert identify all of the relevant propositional variables that are needed to reason about the domain. Each of these will be represented as a **random variable**, i.e., a variable that can take on values from a set of mutually exclusive and exhaustive values called the **sample space** or **partition** of the random variable. Usually this will mean a sample space {*True*, *False*}. For example, the proposition *Cavity* has possible values *True* and *False* indicating whether a given patient has a cavity or not. A random variable that has True and False as its possible values is called a **Boolean random variable**.

  More generally, propositions can include the equality predicate with random variables and the possible values they can have. For example, we might have a random variable *Color* with possible values *red*, *green*, *blue*, and *other*. Then P(Color=red) indicates the likelihood that the color of a given object is red. Similarly, for Boolean random variables we can ask P(A=True), which is abbreviated to P(A), and P(A=False), which is abbreviated to P(~A).

3.9 Probability

Axioms of Probability Theory

Probability Theory provides us with the formal mechanisms and rules for manipulating propositions represented probabilistically. The following are the three axioms of probability theory:

- 0 <= P(A=a) <= 1 for all *a* in sample space of A
- P(True)=1, P(False)=0
- P(A v B) = P(A) + P(B) - P(A ^ B)

From these axioms we can show the following properties also hold:

- P(~A) = 1 - P(A)
- P(A) = P(A ^ B) + P(A ^ ~B)
- Sum{P(A=a)} = 1, where the sum is over all possible values *a* in the sample space of A

Joint Probability Distribution

Given an application domain in which we have determined a sufficient set of random variables to encode all of the relevant information about that domain, we can completely specify all of the possible probabilistic information by constructing the **full joint probability distribution**,

P(V1=v1, V2=v2, ..., Vn=vn), which assigns probabilities to all possible combinations of values to all random variables.

For example, consider a domain described by three Boolean random variables, Bird, Flier, and Young. Then we can enumerate a table showing all possible interpretations and associated probabilities:

Bird	Flier	Young	Probability
T	T	T	0.0
T	T	F	0.2
T	F	T	0.04
T	F	F	0.01
F	T	T	0.01
F	T	F	0.01
F	F	T	0.23
F	F	F	0.5

Notice that there are 8 rows in the above table representing the fact that there are 2^3 ways to assign values to the three Boolean variables. More generally, with *n* Boolean variables the table will be of size 2^n. And if *n* variables each had *k* possible values, then the table would be size k^n.

Also notice that the sum of the probabilities in the right column must equal 1 since we know that the set of all possible values for each variable are known. This means that for *n* Boolean random variables, the table has 2^n-1 values that must be determined to completely fill in the table.

If all of the probabilities are known for a full joint probability distribution table, then we can compute *any* probabilistic statement about the domain. For example, using the table above, we can compute

- P(Bird=T) = P(B) = 0.0 + 0.2 + 0.04 + 0.01 = 0.25
- P(Bird=T, Flier=F) = P(B, ~F) = P(B, ~F, Y) + F(B, ~F, ~Y) = 0.04 + 0.01 = 0.05

Conditional Probabilities

- Conditional probabilities are key for reasoning because they formalize the process of accumulating evidence and updating probabilities based on new evidence. For example,

if we know there is a 4% chance of a person having a cavity, we can represent this as the **prior** (aka unconditional) probability P(Cavity)=0.04. Say that person now has a symptom of a toothache, we'd like to know what is the **posterior** probability of a Cavity given this new evidence. That is, compute P(Cavity | Toothache).

- If P(A|B) = 1, this is equivalent to the sentence in Propositional Logic B => A. Similarly, if P(A|B) =0.9, then this is like saying B => A with 90% certainty. In other words, we've made implication fuzzy because it's not absolutely certain.
- Given several measurements and other "evidence", E1, ..., Ek, we will formulate queries as P(Q | E1, E2, ..., Ek) meaning "what is the degree of belief that Q is true given that we know E1, ..., Ek *and nothing else*."
- **Conditional probability is defined as: P(A|B) = P(A ^ B)/P(B) = P(A,B)/P(B)**
  One way of looking at this definition is as a normalized (using P(B)) joint probability (P(A,B)).
- Example Computing Conditional Probability from the Joint Probability Distribution
  Say we want to compute P(~Bird | Flier) and we know the full joint probability distribution function given above. We can do this as follows:
- P(~B|F) = P(~B,F) / P(F)
- = (P(~B,F,Y) + P(~B,F,~Y)) / P(F)
- = (.01 + .01)/P(F)

Next, we could either compute the marginal probability P(F) from the full joint probability distribution, or, as is more commonly done, we could do it by using a process called **normalization**, which first requires computing

P(B|F) = P(B,F) / P(F)
= (P(B,F,Y) + P(B,F,~Y)) / P(F)
= (0.0 + 0.2)/P(F)

Now we also know that P(~B|F) + P(B|F) = 1, so substituting from above and solving for P(F) we get P(F) = 0.22. Hence, P(~B|F) = 0.02/0.22 = 0.091.

While this is an effective procedure for computing conditional probabilities, it is intractable in general because it means that we must compute and store the full joint probability distribution table, which is exponential in size.

- Some important rules related to conditional probability are:
  - Rewriting the definition of conditional probability, we get the **Product Rule**: P(A,B) = P(A|B)P(B)
  - **Chain Rule**: P(A,B,C,D) = P(A|B,C,D)P(B|C,D)P(C|D)P(D), which generalizes the product rule for a joint probability of an arbitrary number of variables. Note that ordering the variables results in a different expression, but all have the same resulting value.
  - **Conditionalized version of the Chain Rule**: P(A,B|C) = P(A|B,C)P(B|C)
  - **Bayes's Rule**: P(A|B) = (P(A)P(B|A))/P(B), which can be written as follows to more clearly emphasize the "updating" aspect of the rule: P(A|B) = P(A) * [P(B|A)/P(B)] Note: The terms P(A) and P(B) are called the **prior** (or **marginal**)

probabilities. The term P(A|B) is called the **posterior** probability because it is derived from or depends on the value of B.

- **Conditionalized version of Bayes's Rule**: P(A|B,C) = P(B|A,C)P(A|C)/P(B|C)
- **Conditioning (aka Addition) Rule**: P(A) = Sum{P(A|B=b)P(B=b)} where the sum is over all possible values *b* in the sample space of B.
- P(~B|A) = 1 - P(B|A)

3.10 Bayesian Networks

• To do probabilistic reasoning, you need to know the joint probability distribution

• But, in a domain with N propositional variables, one needs 2N numbers to specify the joint probability distribution.

Introduction to Bayesian Networks

Bayesian networks are a type of probabilistic graphical model that uses Bayesian inference for probability computations. Bayesian networks aim to model conditional dependence, and therefore causation, by representing conditional dependence by edges in a directed graph. Through these relationships, one can efficiently conduct inference on the random variables in the graph through the use of factors.

Probability

Before going into exactly what a Bayesian network is, it is first useful to review probability theory.

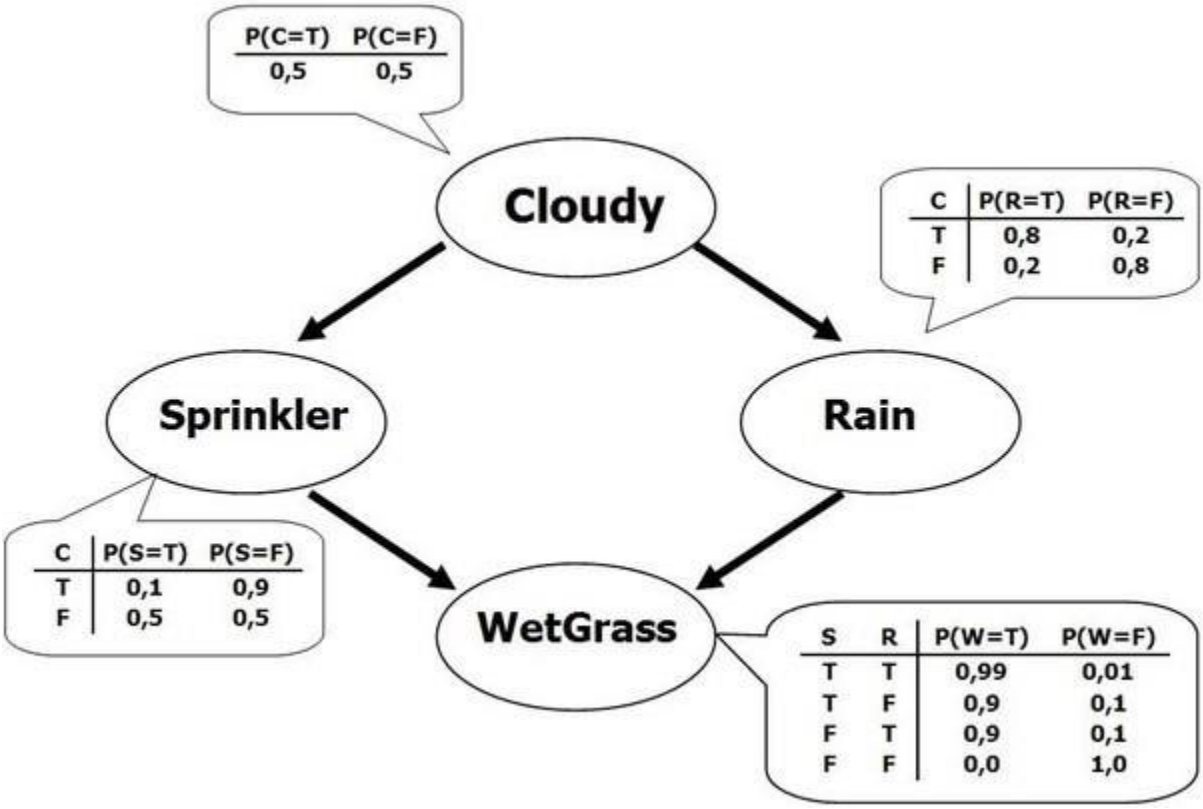
First, remember that the joint probability distribution of random variables A_0, A_1, …, A_n, denoted as P(A_0, A_1, …, A_n), is equal to P(A_1 | A_2, …, A_n) * P(A_2 | A_3, …, A_n) * … * P(A_n) by the chain rule of probability. We can consider this a **factorized** representation of the distribution, since it is a product of N factors that are localized probabilities.

$$\mathrm{P}\left(\bigcap_{k=1}^{n} A_k\right) = \prod_{k=1}^{n} \mathrm{P}\left(A_k \,\middle|\, \bigcap_{j=1}^{k-1} A_j\right)$$

Next, recall that **conditional independence** between two random variables, A and B, given another random variable, C, is equivalent to satisfying the following property: P(A,B|C) = P(A|C) * P(B|C). In other words, as long as the value of C is known and fixed, A and B are independent. Another way of stating this, which we will use later on, is that P(A|B,C) = P(A|C).

The Bayesian Network

Using the relationships specified by our Bayesian network, we can obtain a compact, factorized representation of the joint probability distribution by taking advantage of conditional independence.



A Bayesian network is a **directed acyclic graph** in which each edge corresponds to a conditional dependency, and each node corresponds to a unique random variable. Formally, if an edge (A, B) exists in the graph connecting random variables A and B, it means that P(B|A) is a **factor** in the joint probability distribution, so we must know P(B|A) for all values of B and A in order to conduct inference. In the above example, since Rain has an edge going into WetGrass, it means that P(WetGrass|Rain) will be a factor, whose probability values are specified next to the WetGrass node in a conditional probability table.

Bayesian networks satisfy the **local Markov property**, which states that a node is conditionally independent of its non-descendants given its parents. In the above example, this means that P(Sprinkler|Cloudy, Rain) = P(Sprinkler|Cloudy) since Sprinkler is conditionally independent of its non-descendant, Rain, given Cloudy. This property allows us to simplify the joint distribution, obtained in the previous section using the chain rule, to a smaller form. After simplification, the joint distribution for a Bayesian network is equal to the product of P(node|parents(node)) for all nodes, stated below:

$$P(X_1,\ldots,X_n)=\prod_{i=1}^{n}P\left(X_i \mid X_1,\ldots,X_{i-1}\right)=\prod_{i=1}^{n}P\left(X_i \mid Parents(X_i)\right)$$

In larger networks, this property allows us to greatly reduce the amount of required computation, since generally, most nodes will have few parents relative to the overall size of the network.

Inference

Inference over a Bayesian network can come in two forms.

The first is simply evaluating the joint probability of a particular assignment of values for each variable (or a subset) in the network. For this, we already have a factorized form of the joint distribution, so we simply evaluate that product using the provided conditional probabilities. If we only care about a subset of variables, we will need to marginalize out the ones we are not interested in. In many cases, this may result in underflow, so it is common to take the logarithm of that product, which is equivalent to adding up the individual logarithms of each term in the product.

The second, more interesting inference task, is to find P(x|e), or, to find the probability of some assignment of a subset of the variables (x) given assignments of other variables (our evidence, e). In the above example, an example of this could be to find P(Sprinkler, WetGrass | Cloudy), where {Sprinkler, WetGrass} is our x, and {Cloudy} is our e. In order to calculate this, we use the fact that P(x|e) = P(x, e) / P(e) = αP(x, e), where α is a normalization constant that we will calculate at the end such that P(x|e) + P(¬x | e) = 1. In order to calculate P(x, e), we must marginalize the joint probability distribution over the variables that do not appear in x or e, which we will denote as Y.

$$P(x|e)=\alpha\sum_{\forall y\in Y}P(x,e,Y)$$

For the given example, we can calculate P(Sprinkler, WetGrass | Cloudy) as follows:

$$P(Sprinkler, WetGrass \mid Cloudy) = \alpha \sum_{Rain} P(WetGrass \mid Sprinkler, Rain) P(Sprinkler \mid Cloudy) P(Rain \mid Cloudy) P(Cloudy)$$

$$= \alpha P(WetGrass \mid Sprinkler, Rain) P(Sprinkler \mid Cloudy) P(Rain \mid Cloudy) P(Cloudy) + \alpha P(WetGrass \mid Sprinkler, \neg Rain) P(Sprinkler \mid Cloudy) P(\neg Rain \mid Cloudy) P(Cloudy)$$

We would calculate P(¬x | e) in the same fashion, just setting the value of the variables in x to false instead of true. Once both P(x | e) and P(¬x | e) are calculated, we can solve for α, which equals 1 / (P(x | e) + P(¬x | e)).

Note that in larger networks, Y will most likely be quite large, since most inference tasks will only directly use a small subset of the variables. In cases like these, exact inference as shown above is very computationally intensive, so methods must be used to reduce the amount of computation. One more efficient method of exact inference is through variable elimination, which takes advantage of the fact that each factor only involves a small number of variables. This means that the summations can be rearranged such that only factors involving a given variable are used in the marginalization of that variable. Alternatively, many networks are too large even for this method, so approximate inference methods such as MCMC are instead used; these provide probability estimations that require significantly less computation than exact inference methods.