Planning and acting in real world

Planning a course of action is a key requirement for an intelligent agent. The right representation for actions and states and the right algorithms can make this easier.

**Definition of Classical Planning**

* Classical planning is defined as the task of finding a sequence of actions to accomplish a goal in a discrete, deterministic, static, fully observable environment.

We have seen two approaches to this task: the problem-solving agent and the hybrid propositional logical. Both share two limitations. First, they both require ad hoc heuristics for each new domain: a heuristic evaluation function for search, and hand-written code for the hybrid wumpus agent. Second, they both need to explicitly represent an exponentially large state space.

In response to these limitations, planning researchers have invested in a factored representation using a family of languages called PDDL, the Planning Domain Definition Language (Ghallab et al., 1998), which allows us to express all 4Tn2 actions with a single action schema, and does not need domain-specific knowledge.

The agent performs three tasks in classical planning:

* Planning: The agent plans after knowing what is the problem.
* Acting: It decides what action it has to take.
* Learning: The actions taken by the agent make him learn new things.

PDLL describe 4 basic things needed in a search problem:

* In PDDL, a state is represented as a conjunction of ground atomic fluents.
* An **action schema** represents a family of ground actions. For example, here is an action

schema for flying a plane from one location to another:

*Action* (*Fly*(*p,from,to*)*,*

PRECOND: *At*(*p,from*)*^Plane*(*p*)*^Airport*(*from*)*^Airport*(*to*)

EFFECT: *¬At*(*p,from*)*^At*(*p,to*))

* Result: It is obtained by the set of actions used by the agent.
* Goal: It is same as a precondition, which is a conjunction of literals (whose value is either positive or negative).

There are various examples which will make PDLL understandable:

* Air cargo transport
* The spare tire problem
* The blocks world and many more.

Let’s discuss one of them

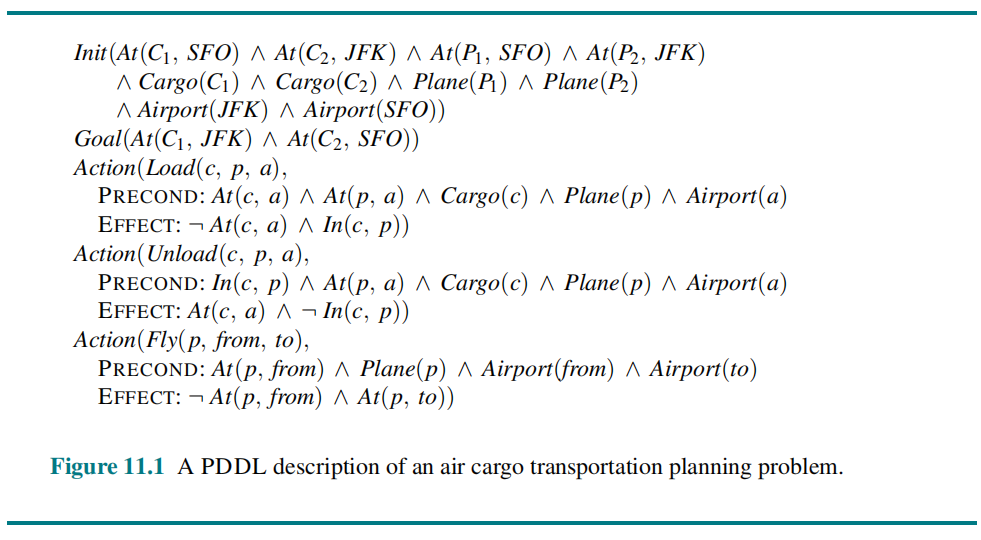
**Air cargo transport**

This problem can be illustrated with the help of the following actions:

* Load: This action is taken to load cargo.
* Unload: This action is taken to unload the cargo when it reaches its destination.
* Fly: This action is taken to fly from one place to another.

Therefore, the Air cargo transport problem is based on loading and unloading the cargo and flying it from one place to another.

Below is the PDLL description for Air cargo transport:



**The above described actions, (i.e., load, unload, and fly) affects the following two predicates:**

* **(c,p):** In this, the cargo is inside the plane **p**.
* **(x,a):** In this, the object **x** is at the airport **a**. Here, object can be the **cargo** or **plane**.

*It is to be noted that when the plan flies from one place to another, it should carry all cargo inside it. It becomes difficult with the PDLL to give solution for such a problem. Because PDLL do not have the universal quantifier.* **Thus, the following approach is used:**

* piece of cargo ceases to be**On** anywhere when it is In a plane.
* the cargo only becomes**On** the new airport when it is unloaded.

**Therefore, the planning for the solution is:**

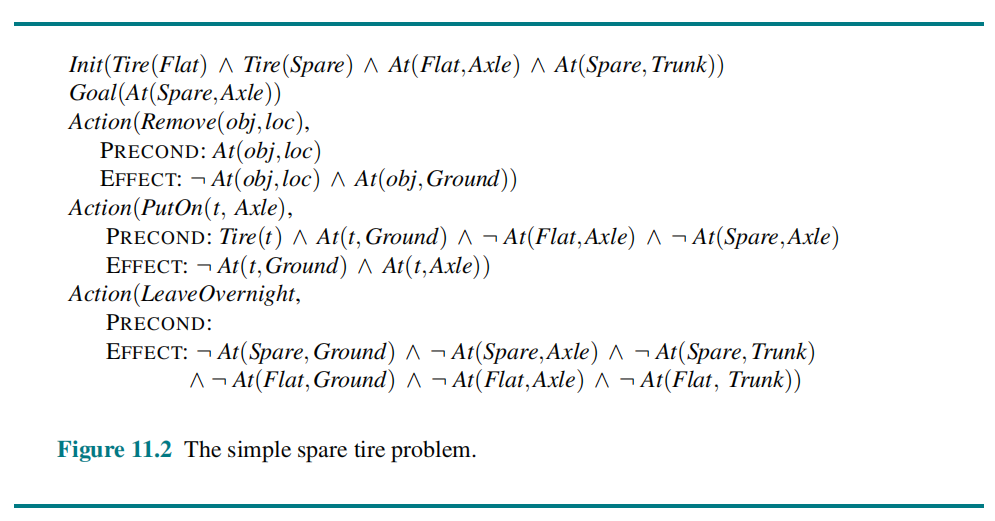


**The spare tire problem**

The problem is that the agent needs to change the flat tire. The aim is to place a good spare tire over the car’s axle. There are four actions used to define the spare tire problem:

1. Remove the spare from the trunk.
2. Remove the flat spare from the axle.
3. Putting the spare on the axle.
4. Leave the car unattended overnight. Assuming that the car is parked at an unsafe neighborhood.

The PDLL description for the spare tire problem is:



The solution to the problem is:

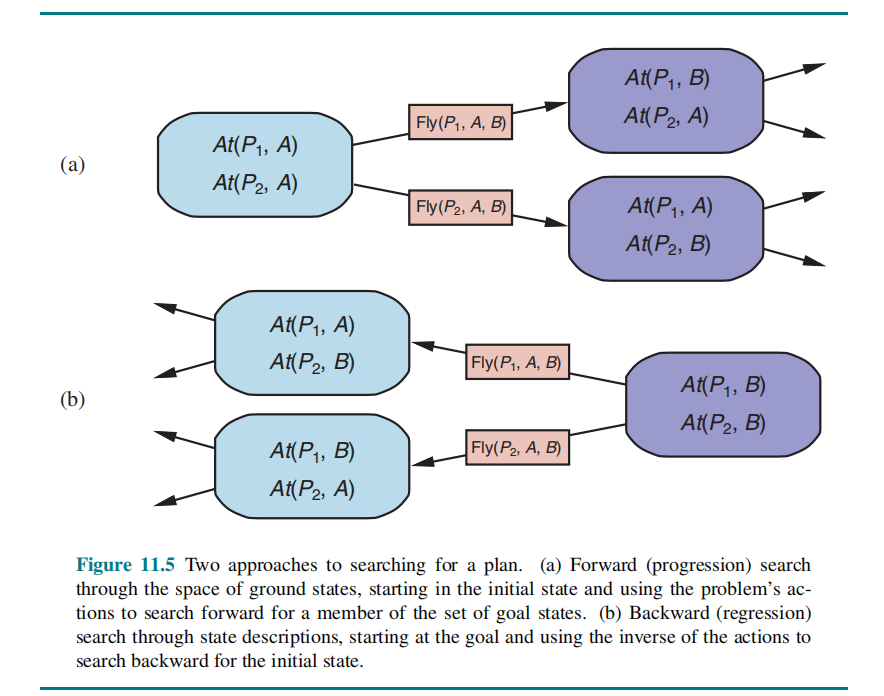
[*Remove*(*Flat,Axle*)*,Remove*(*Spare,Trunk*)*,PutOn*(*Spare,Axle*)]

**Algorithms for Classical Planning**

The description of a planning problem provides an obvious way to search from the initial state through the space of states, looking for a goal. A nice advantage of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state.

**Method # 1. Planning with State-Space Search:**

The most straight forward approach is to use state-space search. Because the descriptions of actions in a planning problem specify both preconditions and effects, it is possible to search in either direction: forward from the initial state or backward from the goal, as shown in Fig. 8.5. We can also use the explicit action and goal representations to derive effective heuristics automatically.



* 1. **Forward State-Space Search**: Planning with forward state-space search is similar to the problem-solving approach. It is sometimes called progression planning, because it moves in the forward direction. We start with the problem’s initial state, considering sequences of actions until we reach a goal state.

**The formulation of planning problem as state-space search problems is as follows:**

i. The initial state of the search is the initial state from the planning problem. In general each state will be set of positive ground literals; literals not appearing are false.

ii. The actions which are applicable to a state are all those whose preconditions are satisfied. The successor state resulting from an action is generated by adding the positive effect literals and deleting the negative effect literals.

iii. The goal test checks whether the state satisfies the goal of the planning problem.

iv. The step cost of each action is typically 1. Although it would be easy to allow different costs for different actions, this was seldom done by STRIPS planners.

Since function symbols are not present, the state space of a planning problem is finite and therefore, any graph search algorithm such as A \* will be a complete planning algorithm.

Consider for example, an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo.

The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem:

load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. But finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded), or loaded into any plane at its airport (if it is unloaded).

On average, let’s say there are about 1000 possible actions, so the search tree up to the depth of the obvious solution has about 1000 nodes. It is thus clear that a very accurate heuristic will be needed to make this kind of search efficient.

* 1. **Backward State-Space Search:**

In backward search (also called regression search) we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. At each step we consider relevant actions (in contrast to forward search, which considers actions that are applicable). This reduces the branching factor significantly, particularly in domains with many possible actions.

**Planning Graphs**

Planning graphs are an efficient way to create a representation of a planning problem that can be used to

* Achieve better heuristic estimates
* Directly construct plans

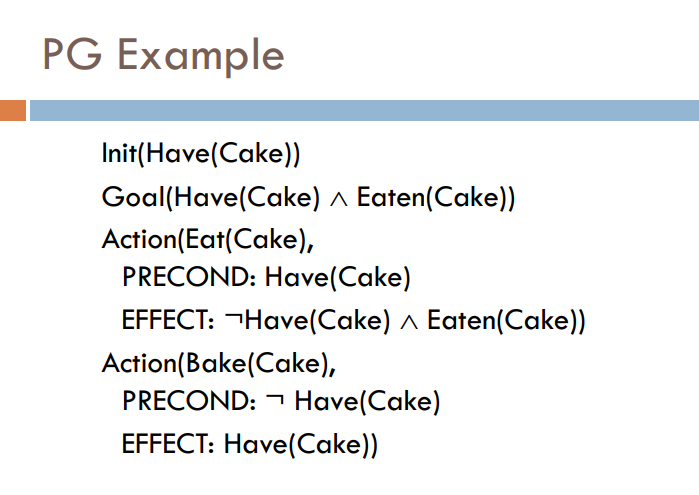
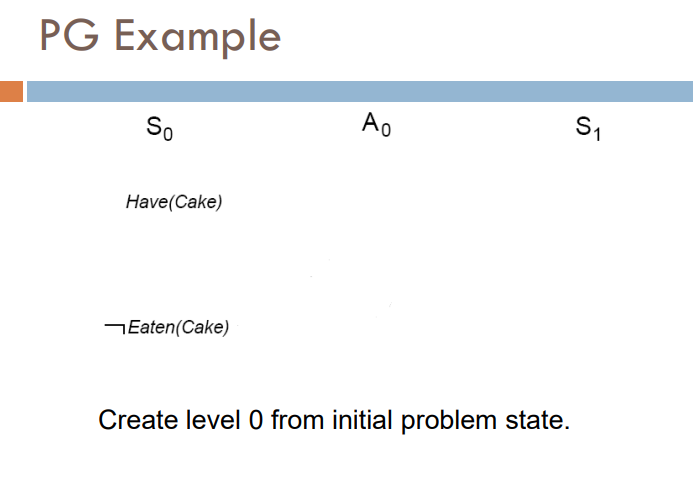
Planning graphs only work for propositional problems.

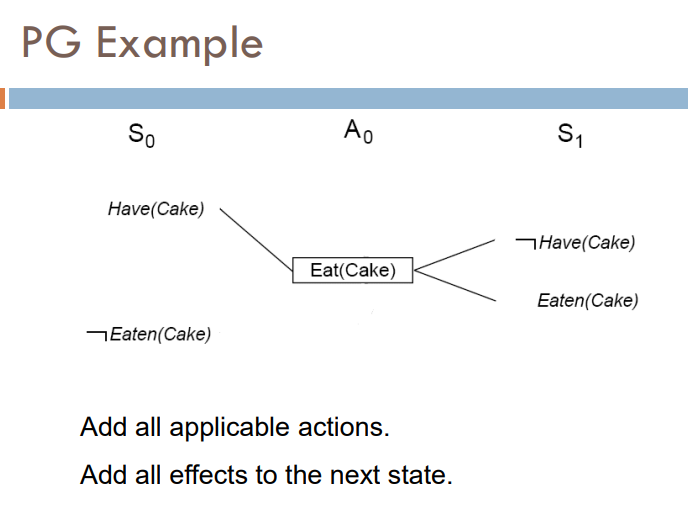
Planning graphs consists of a seq of levels that correspond to time steps in the plan.

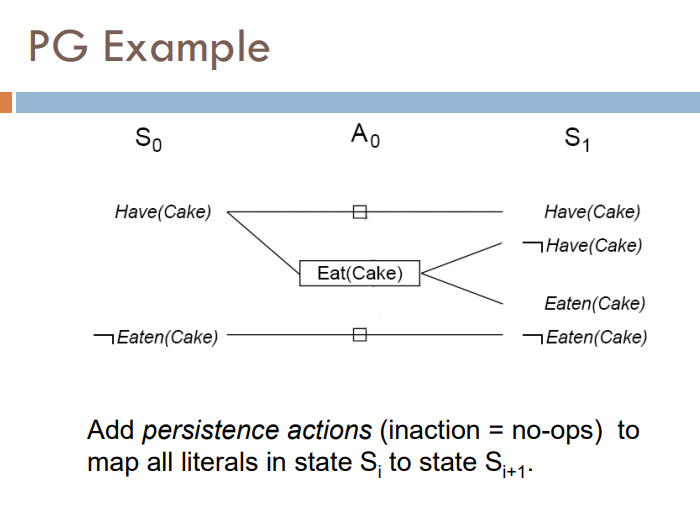
* Level 0 is the initial state.
* Each level consists of a set of literals and a set of actions that represent what might be possible at that step in the plan
* Might be is the key to efficiency
* Records only a restricted subset of possible negative interactions among actions.

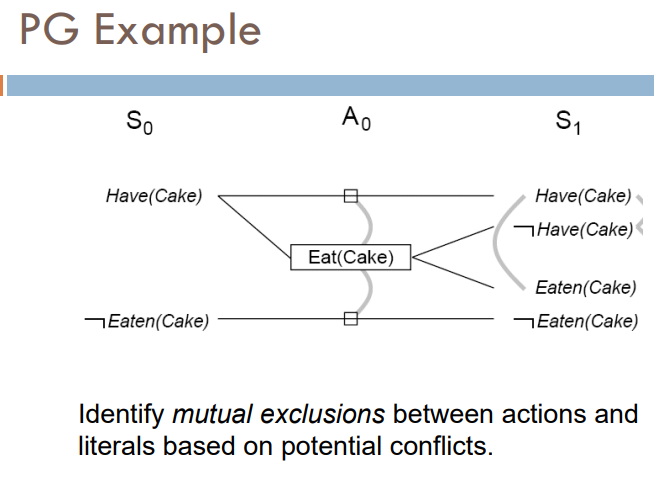
Each level consists of

* Literals = all those that could be true at that time step, depending upon the actions executed at preceding time steps.
* Actions = all those actions that could have their preconditions satisfied at that time step, depending on which of the literals actually hold.





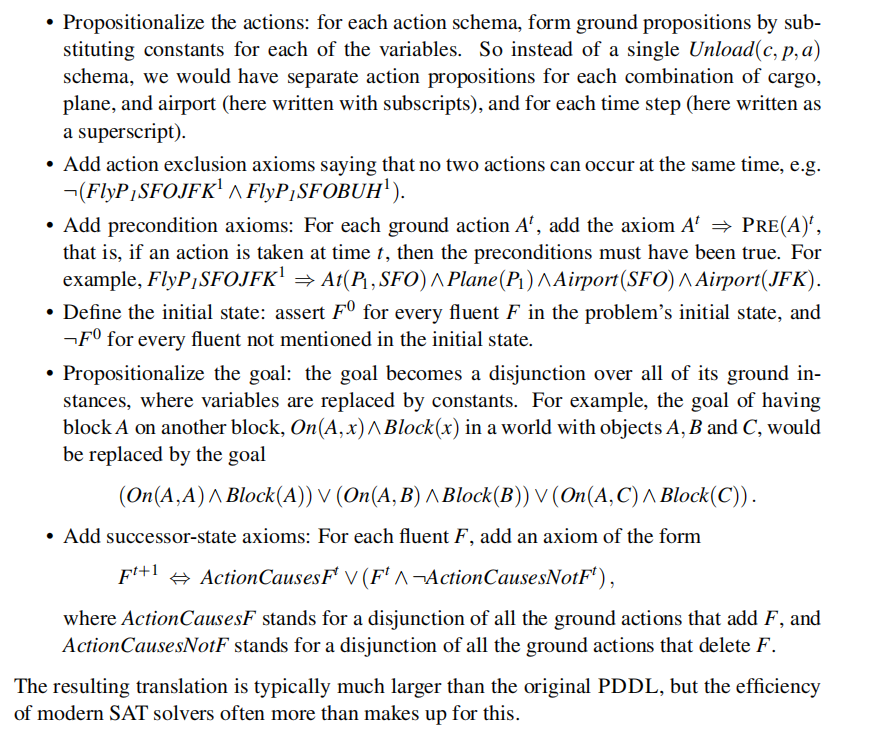


**Classical Planning as Boolean satisfiability**

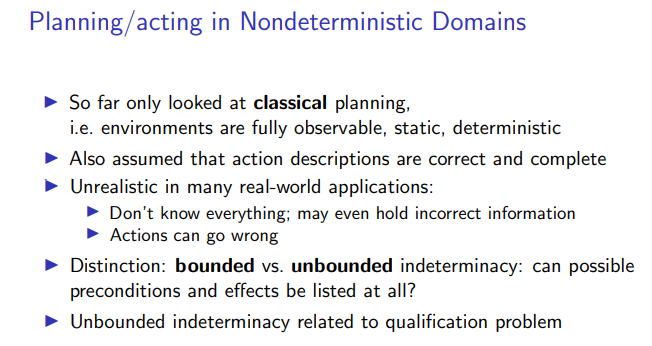
This is yet another approach to improve the expressiveness and complexity of the classical planning approaches. In this technique what we do is convert our classical planning problems representation into a well-known representation called Propositional Satisfiability Problem, also called Boolean Satisfiability Problem, or simply SAT.

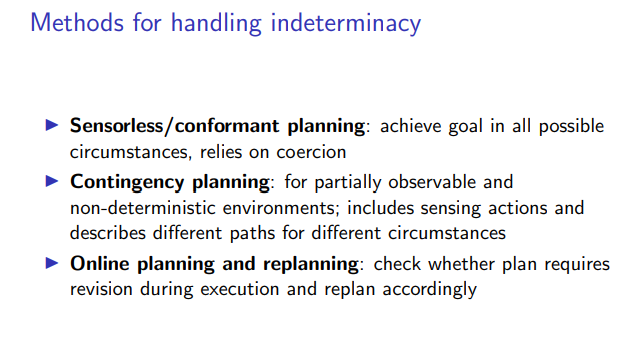
If we can convert it to a well-known problem representation, we can use existing algorithms to solve the problems.

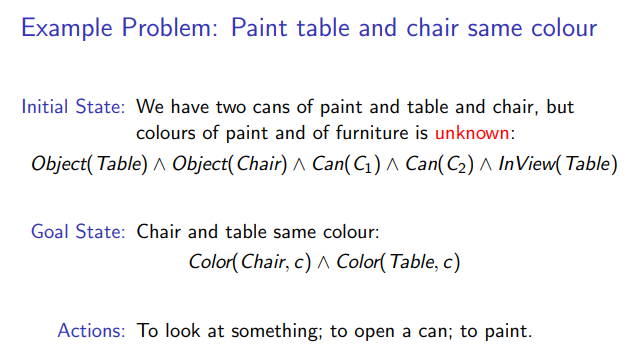
SAT-based planners such as SATPLAN operate by translating a PDDL problem description into propositional form. The translation involves a series of steps:

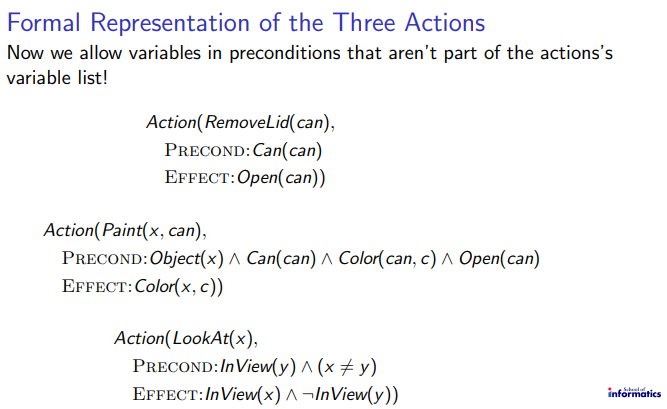


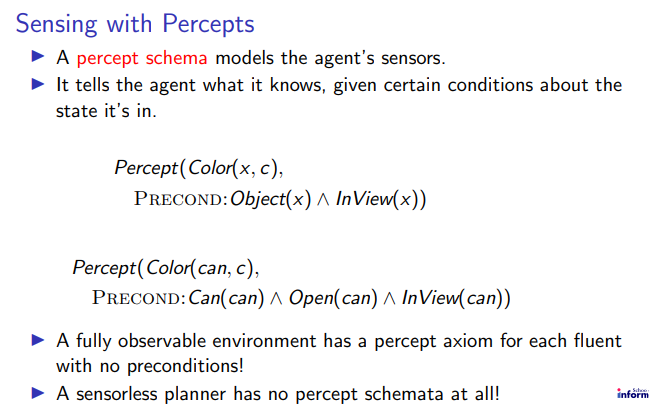
**Planning and Acting in Nondeterministic Domains**

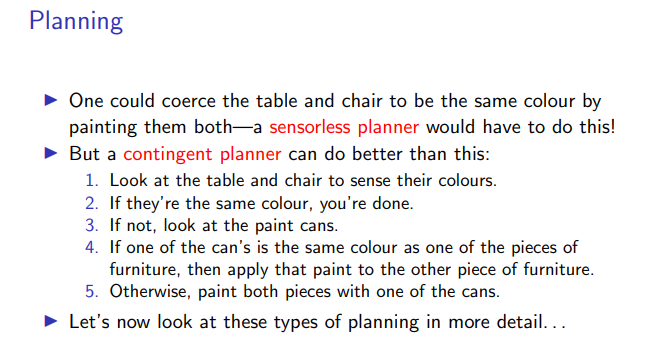




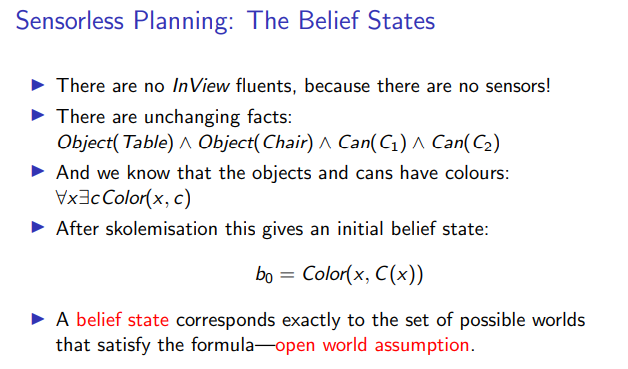


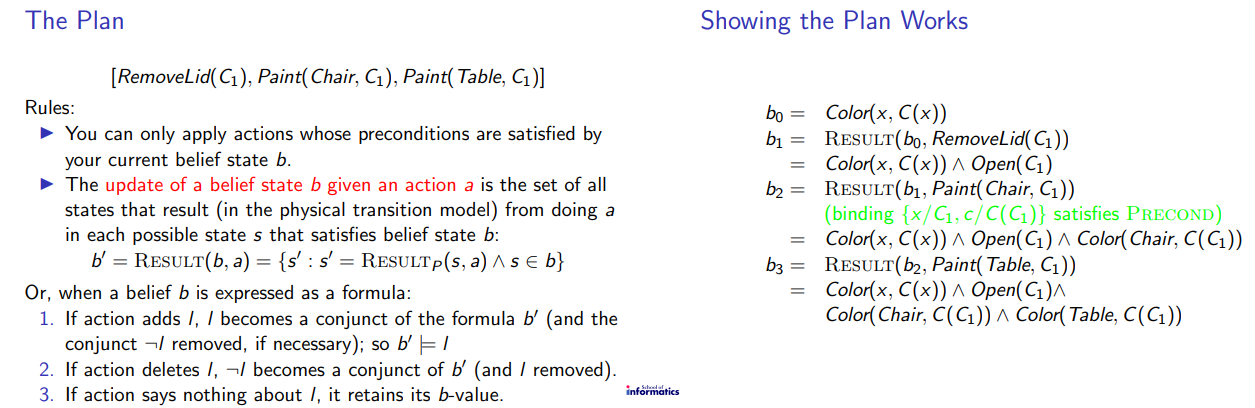


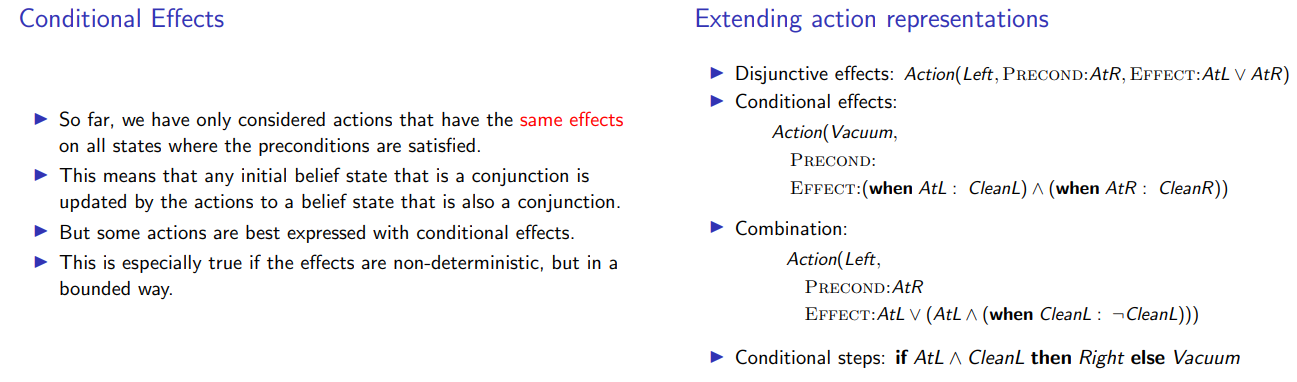




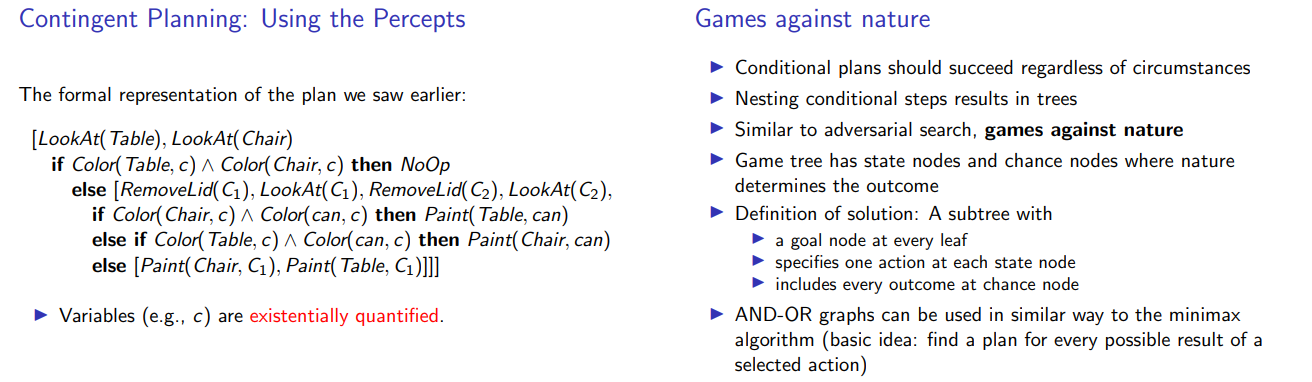
1.

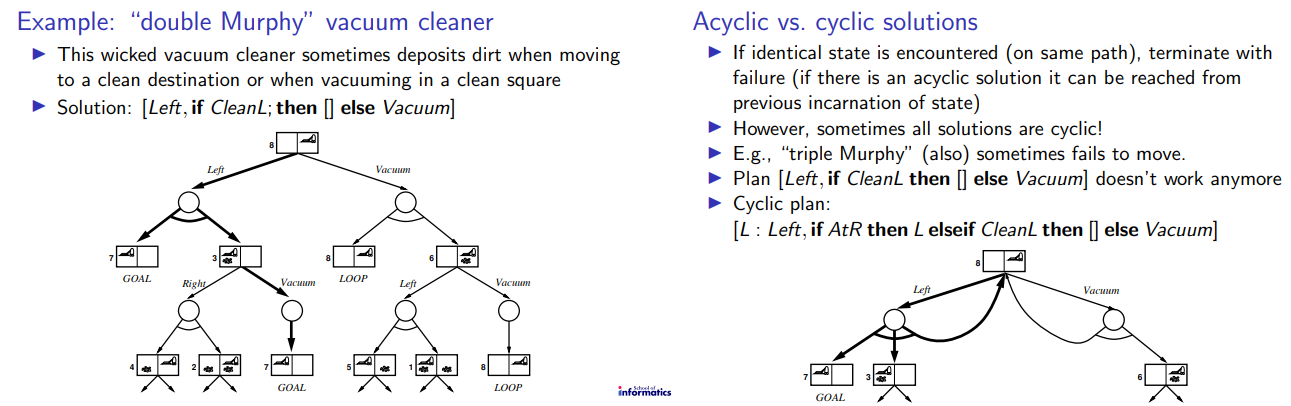


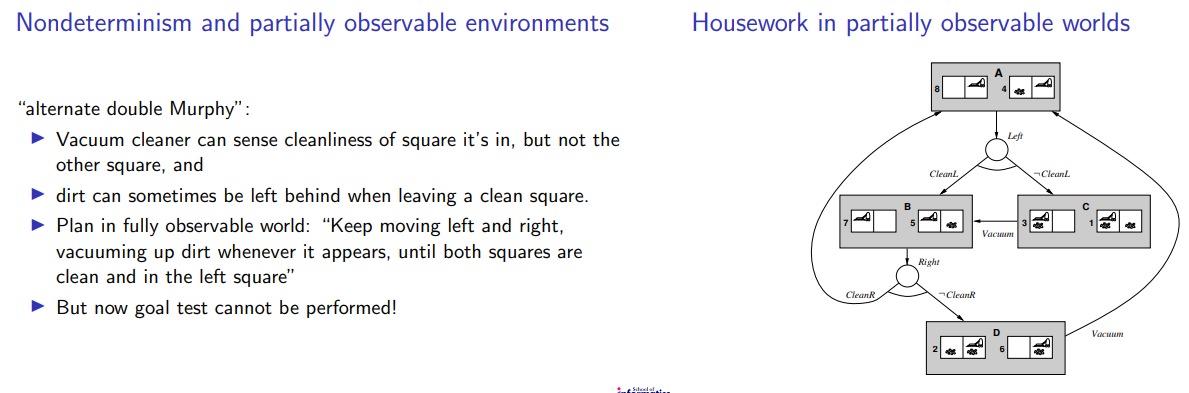




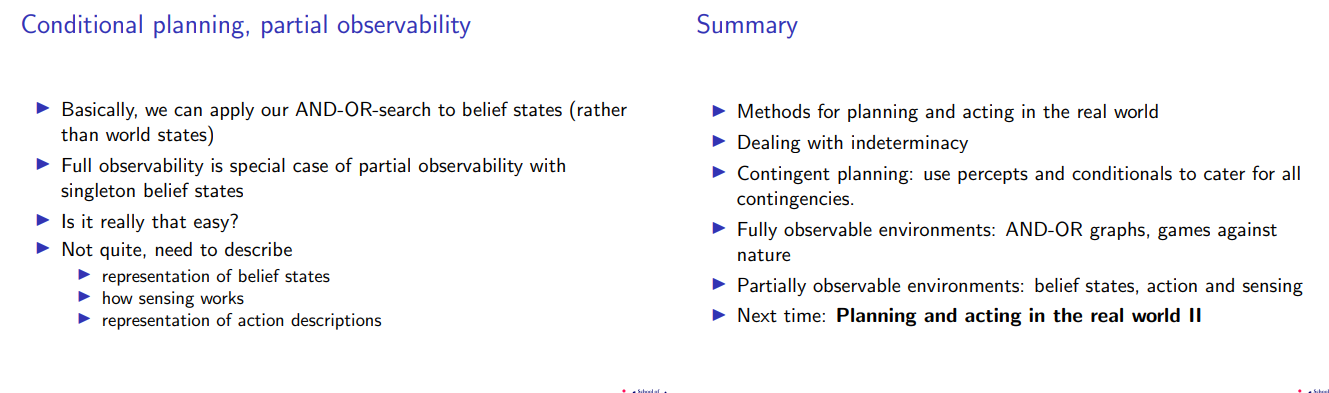
2.

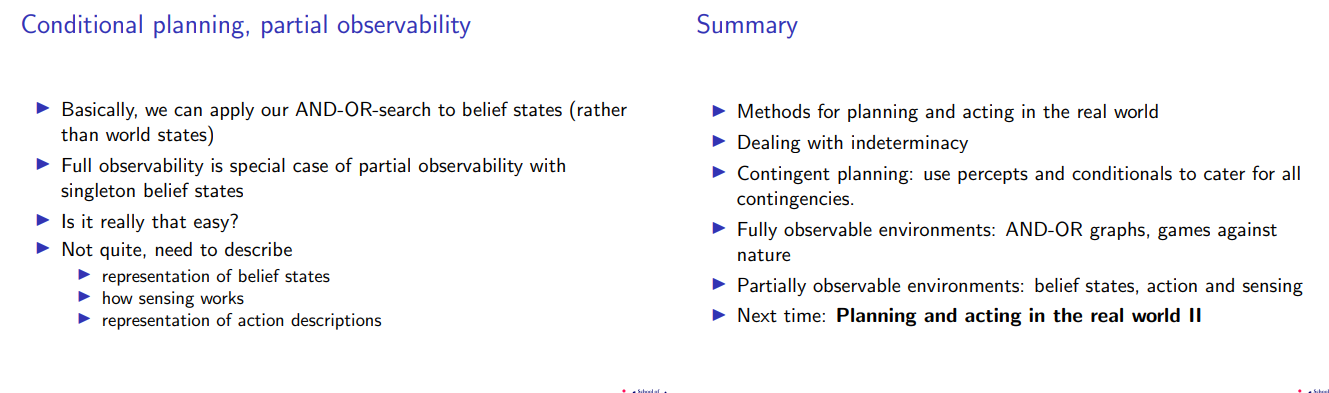






3.





**Time, Schedules, and Resources**

Classical planning talks about what to do, in what order, but does not talk about time: how long an action takes and when it occurs. For example, in the airport domain we could produce a plan saying what planes go where, carrying what, but could not specify departure and arrival times. This is the subject matter of scheduling.

The real world also imposes resource constraints: an airline has a limited number of staff, and staff who are on one flight cannot be on another at the same time. This section introduces techniques for planning and scheduling problems with resource constraints.

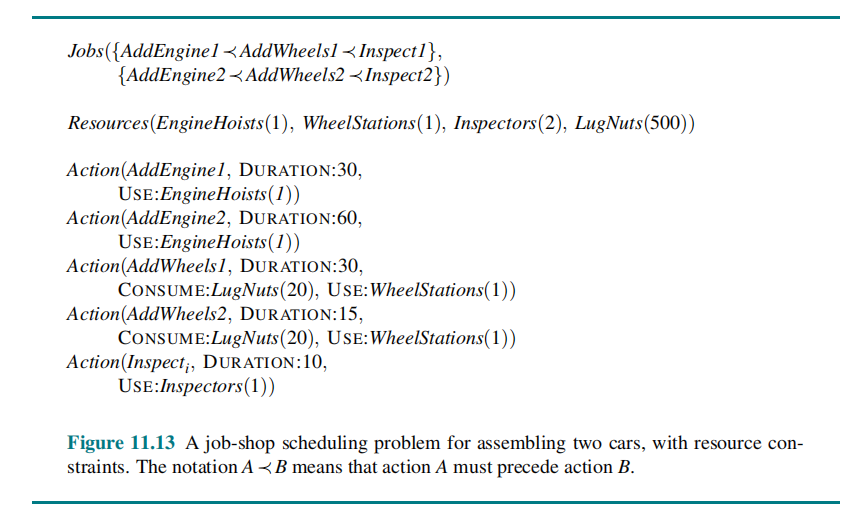
The approach we take is “plan first, schedule later”: divide the overall problem into a planning phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later scheduling phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

**Representing temporal and resource constraints**

A typical job-shop scheduling problem consists of a set of jobs, each of which has a collection of actions with ordering constraints among them. Each action has Job a duration and a set of resource constraints required by the action. A constraint specifies a type of resource (e.g., bolts, wrenches, or pilots), the number of that resource required, and whether that resource is consumable (e.g., the bolts are no longer available for use) or reusable (e.g., a pilot is occupied during a flight but is available again when the flight is over). Actions can also produce resources (e.g., manufacturing and resupply actions).

A solution to a job-shop scheduling problem specifies the start times for each action and must satisfy all the temporal ordering constraints and resource constraints. As with search and planning problems, solutions can be evaluated according to a cost function; this can be quite complicated, with nonlinear resource costs, time-dependent delay costs, and so on. For simplicity, we assume that the cost function is just the total duration of the plan, which is called the makespan.

The representation of resources as numerical quantities, such as Inspectors(2), rather than as named entities, such as Inspector(I1) and Inspector(I2), is an example of a technique called aggregation: grouping individual objects into quantities when the objects are all in-distinguishable.



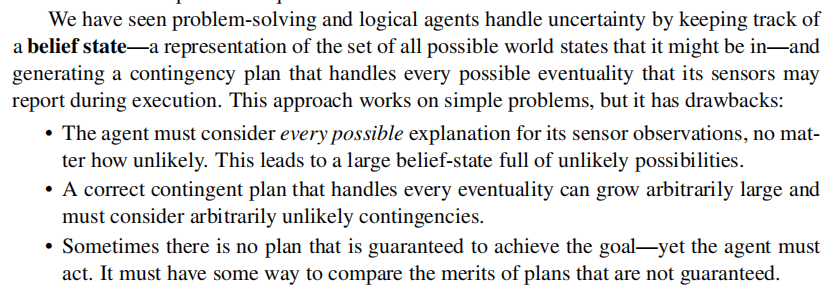
**Knowledge Representation** 🡪 *read online*

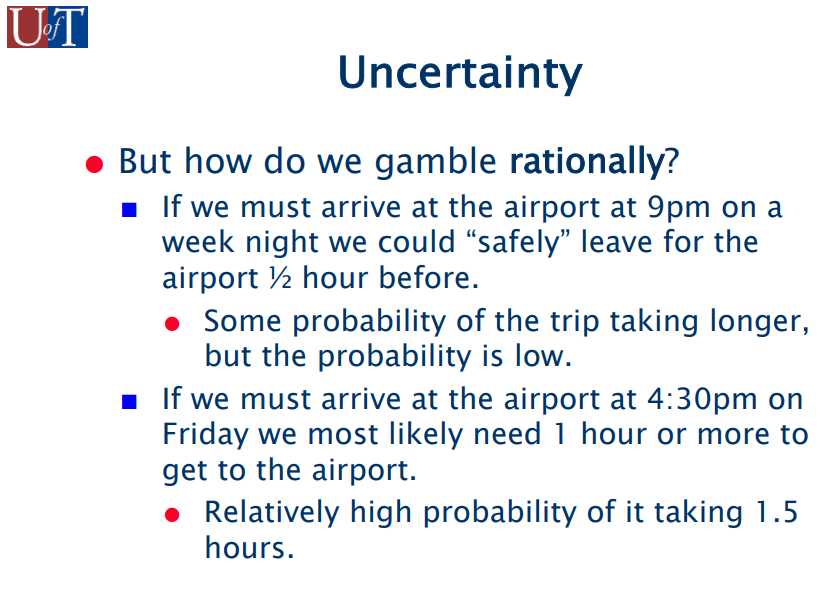
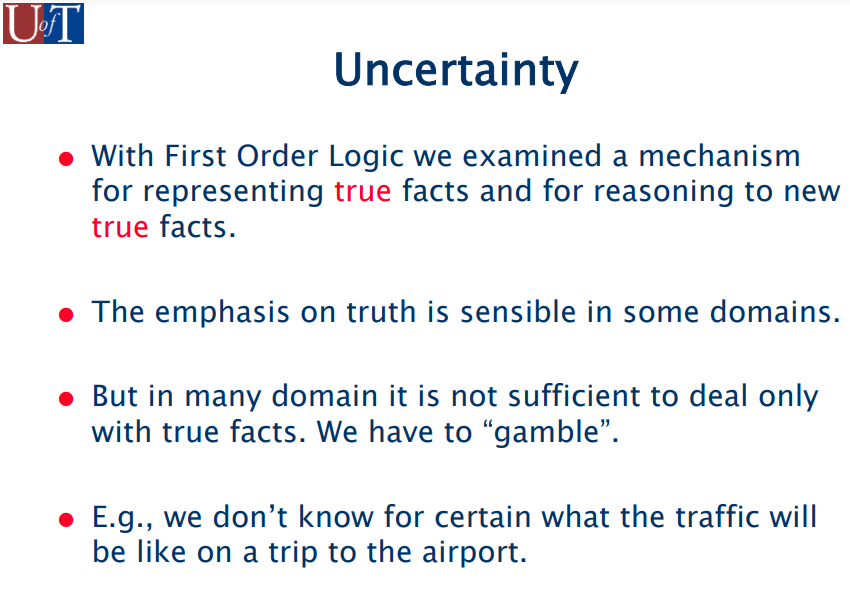
**Acting under Uncertainty**

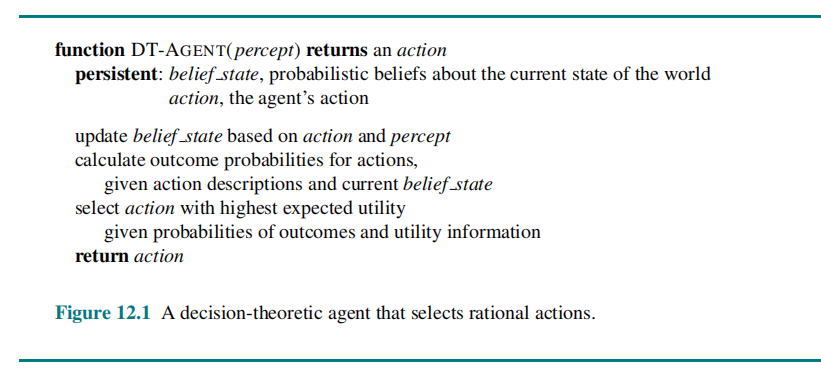
Agents in the real world need to handle uncertainty, whether due to partial observability, nondeterminism, or adversaries. An agent may never know for sure what state it is in now or

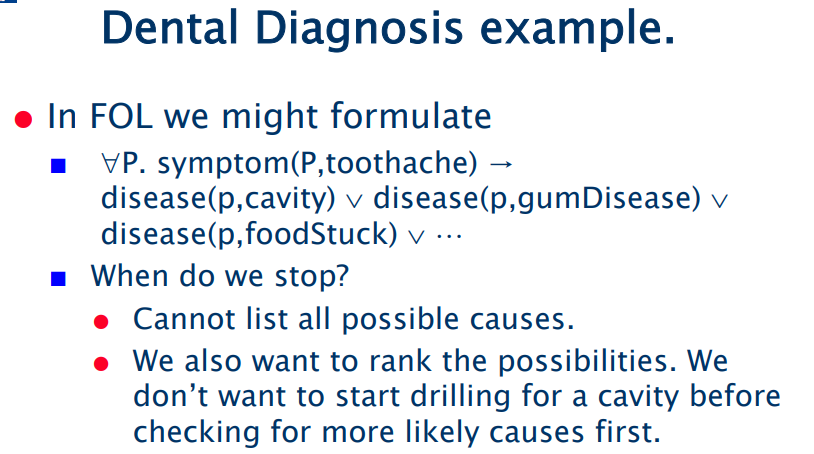
where it will end up after a sequence of actions.

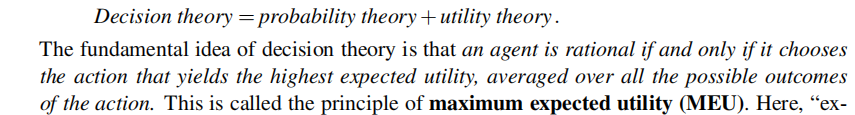
Uncertainty arises because of both laziness and ignorance. It is inescapable in complex, nondeterministic, or partially observable environments.











PROBABILISTIC REASONING

<https://www.javatpoint.com/probabilistic-reasoning-in-artifical-intelligence>

**Time and Uncertainty**

We have developed our techniques for probabilistic reasoning in the context of static worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem: treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including the actual blood sugar level and insulin level. Given this information, we can make a decision about the patient’s food intake and insulin dose. Unlike the case of car repair, here the dynamic aspects of the problem are essential. Blood sugar levels and measurements thereof can change rapidly over time, depending on recent food intake and insulin doses, metabolic activity, the time of day, and so on.

To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

*Rest read yourself.*

Unit- iv

LEARNING

An agent is learning if it improves its performance after making observations about the world.

When the agent is a computer,

we call it machine learning: a computer observes some data, builds a model based on the data, and uses the model as both a hypothesis about the world and a piece of software that can solve problems.

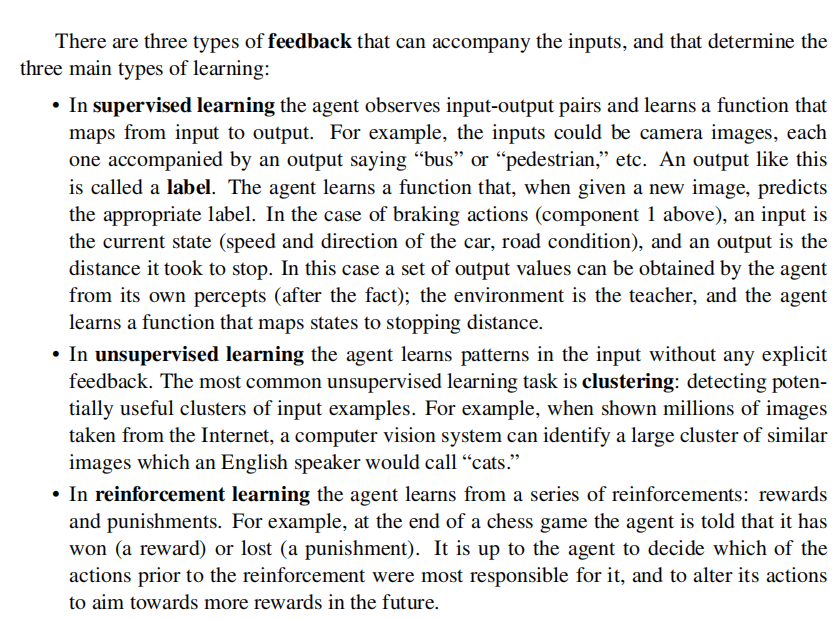
**Forms of Learning**

Any component of an agent program can be improved by machine learning. The improvements, and the techniques used to make them, depend on these factors:

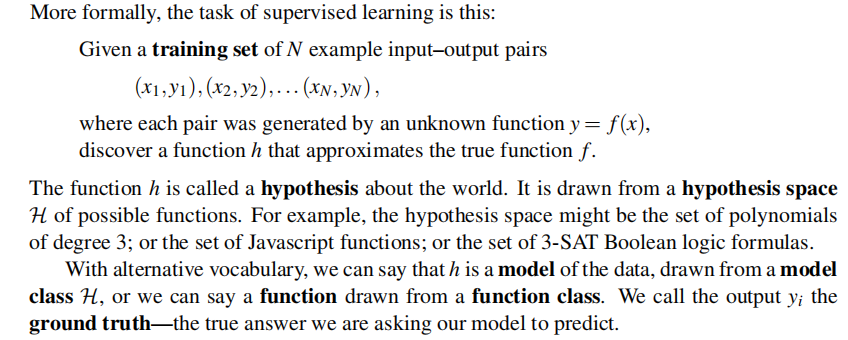
• Which component is to be improved.

• What prior knowledge the agent has, which influences the model it builds.

• What data and feedback on that data is available.



**Supervised Learning**



We say that a hypothesis is **underfitting** when it fails to find a pattern in the data. On the other hand, the piecewise linear function has **low bias**; the shape of the function is driven by the data.

We say a function is **overfitting** the data when it pays too much attention to the particular data set it is trained on, causing it to perform poorly on unseen data.

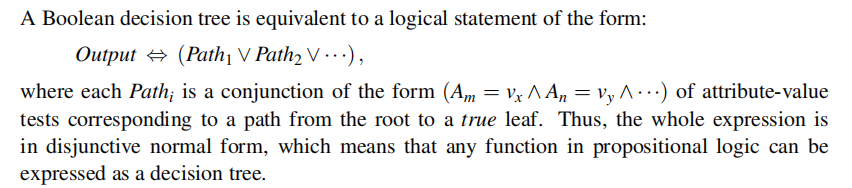
**Learning Decision Trees**

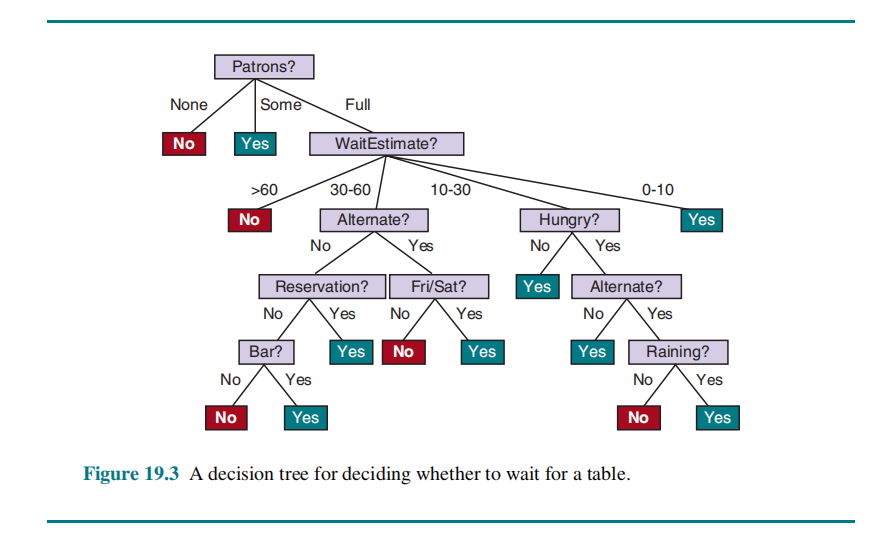
A decision tree is a representation of a function that maps a vector of attribute values a single output value—a “decision.” A decision tree reaches its decision by performing a sequence of tests, starting at the root and following the appropriate branch until a leaf is reached. Each internal node in the tree corresponds to a test of the value of one of the input attributes, the branches from the node are labeled with the possible values of the attribute, and the leaf nodes specify what value is to be returned by the function.

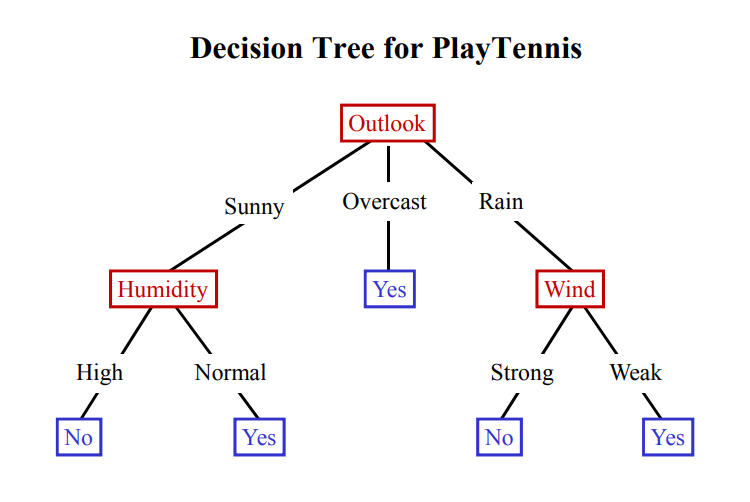
In decision tree, inputs consisting of discrete values and outputs that are either true (a positive

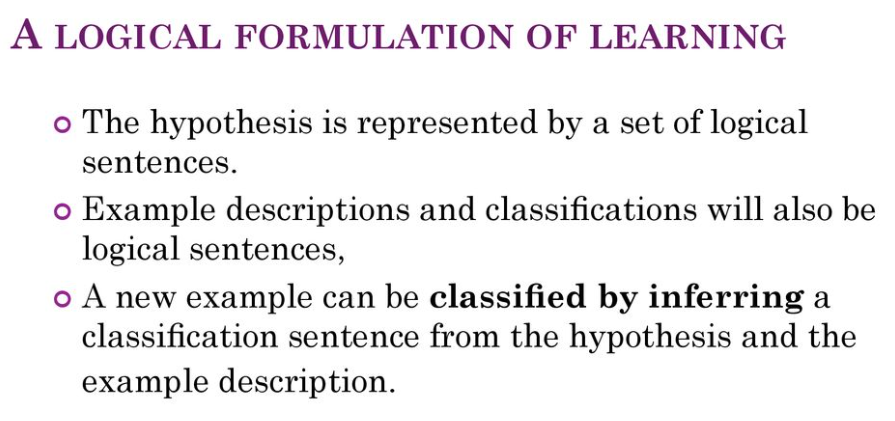
example) or false (a negative example). We call this Boolean classification.

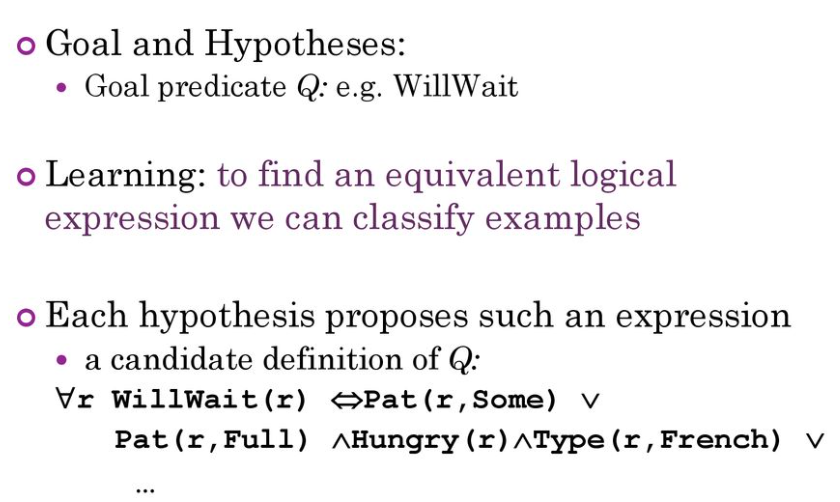
**Expressiveness of decision trees**

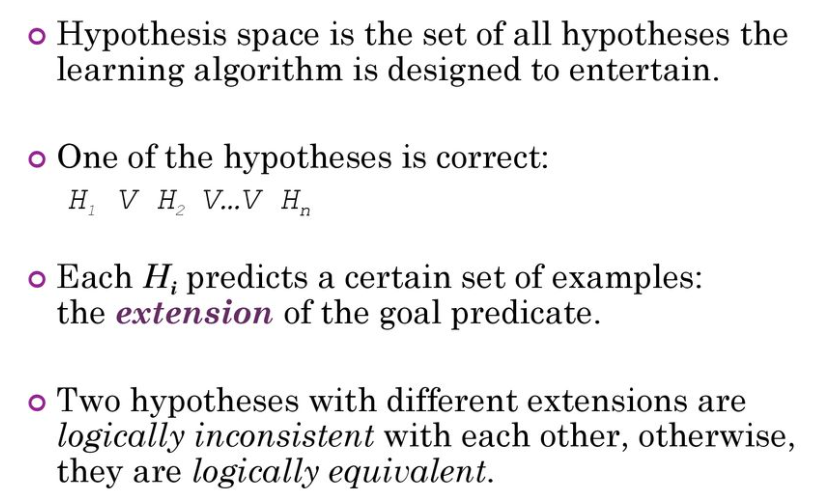


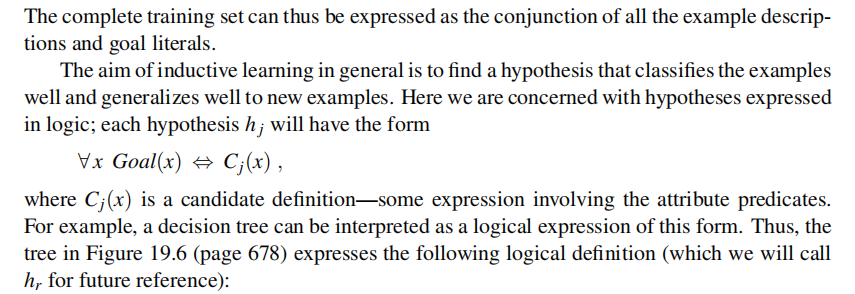


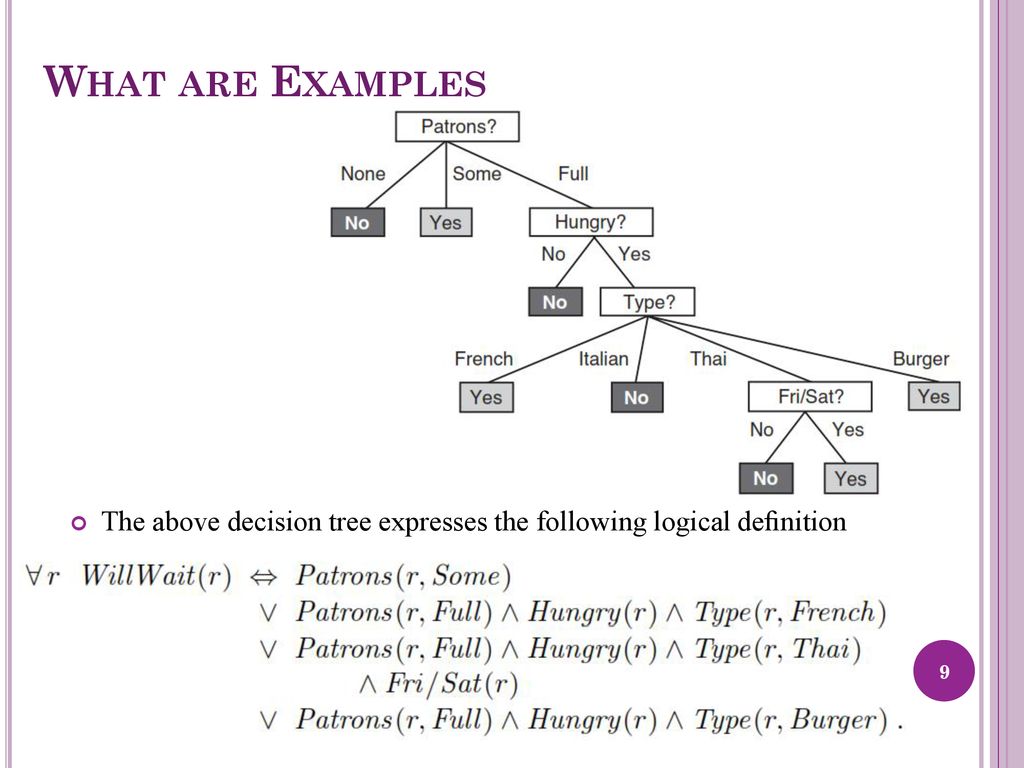


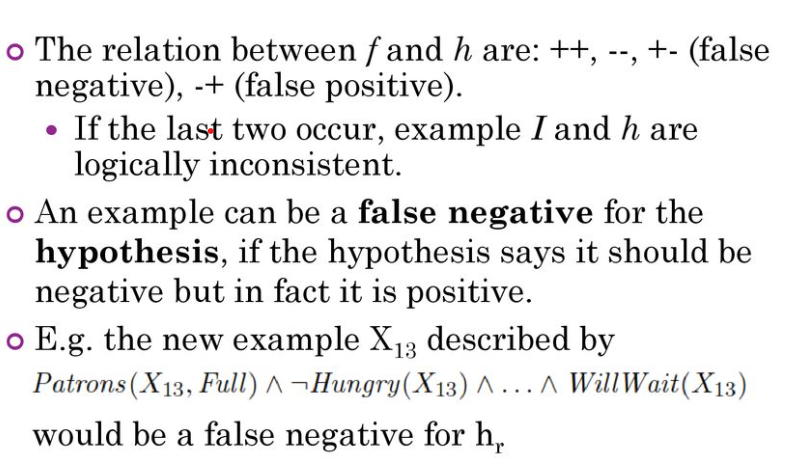


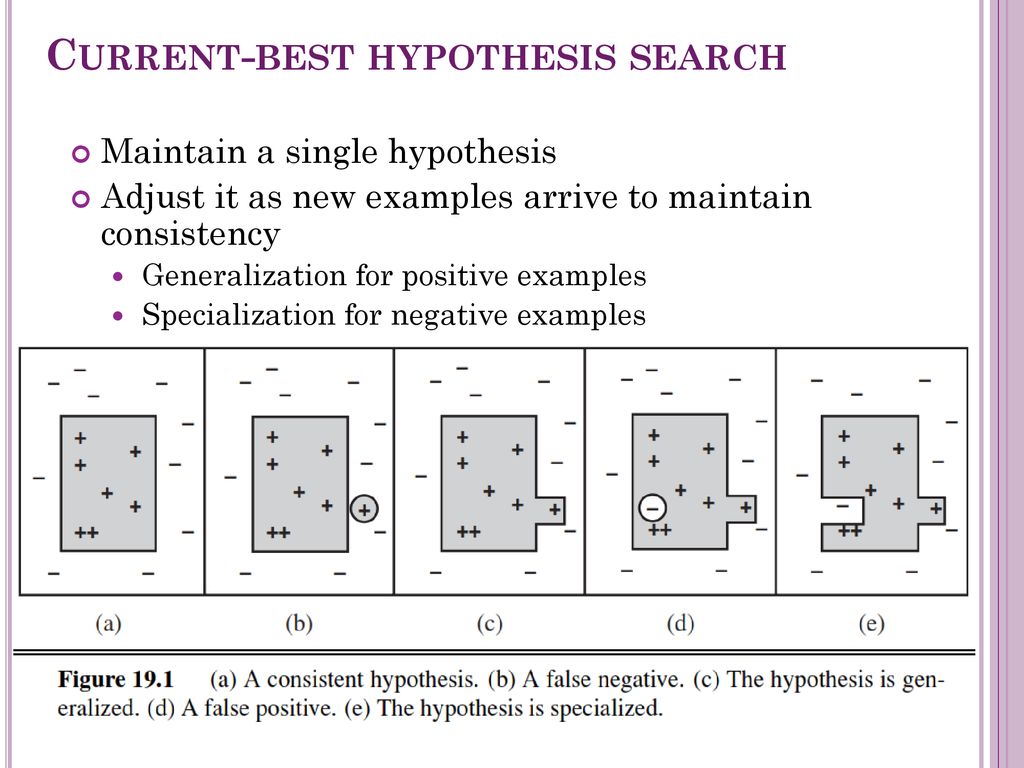


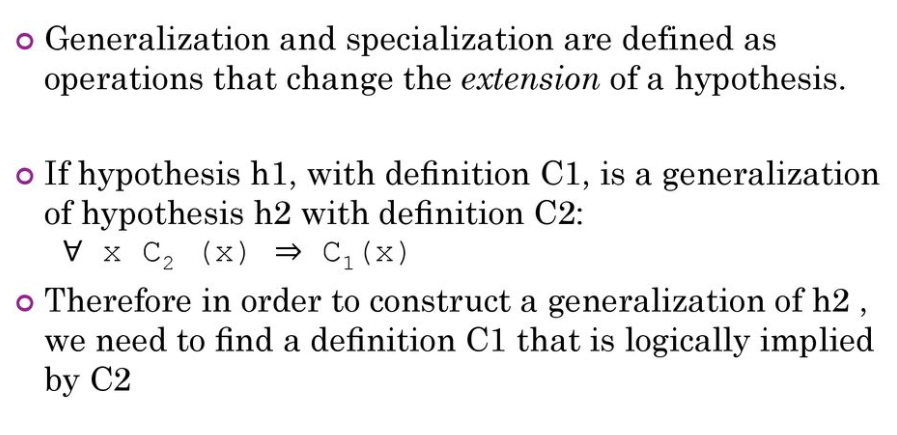


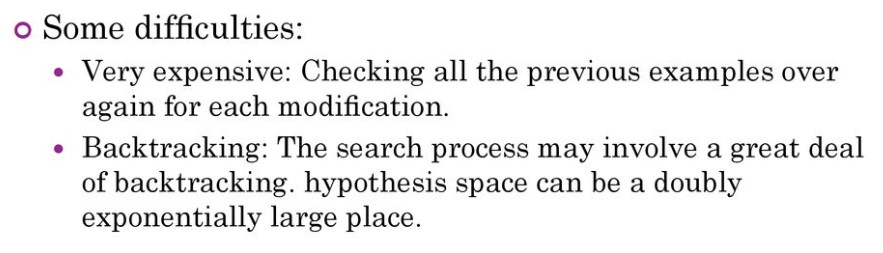














Backtracking arises because the current-best-hypothesis approach has to choose a particular

hypothesis as its best guess even though it does not have enough data yet to be sure of the

choice.

**Statistical Learning**

The key concepts are data and hypotheses. Here, the data are evidence—that is, instantiations of some or all of the random variables describing the domain. The hypotheses in this chapter are probabilistic theories of how the domain works, including logical theories as a special case.

**Learning with Complete Data**

The general task of learning a probability model, given data that are assumed to be generated

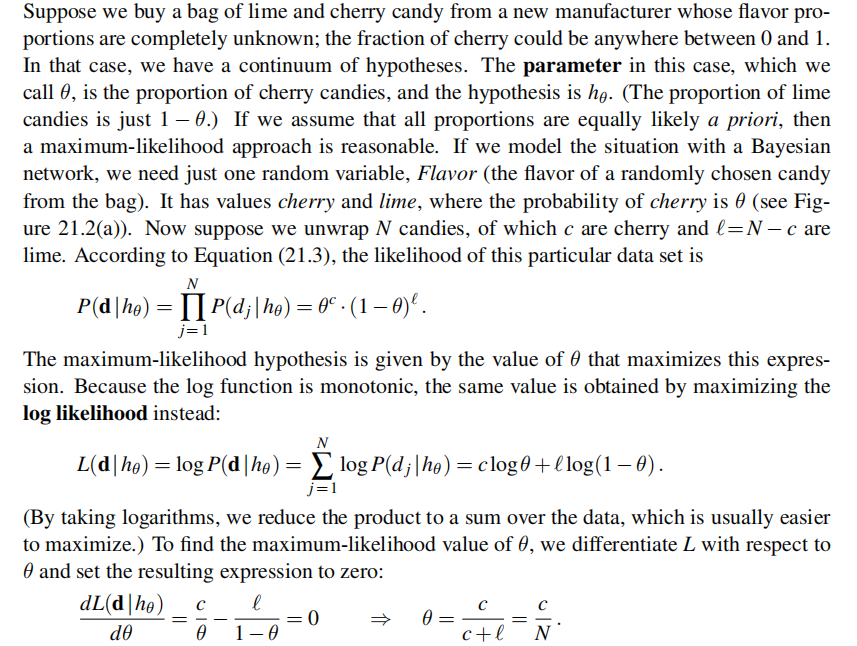
from that model, is called density estimation. (The term applied originally to probability

density functions for continuous variables, but it is used now for discrete distributions too.)

Density estimation is a form of unsupervised learning.

Data are complete when each data point contains values for every variable in the probability model being learned. We focus on parameter learning—finding the numerical parameters for a probability model whose structure is fixed. For example, we might be interested in learning the conditional probabilities in a Bayesian network with a given structure. We will also look briefly at the problem of learning structure and at nonparametric density estimation.

* **Maximum-likelihood parameter learning: Discrete model**





NATURAL LANGUAGE PROCESSING

Natural language processing strives to build machines that understand and respond to text or voice data—and respond with text or speech of their own—in much the same way humans do.

**Language Models**

Formal languages, such as first-order logic, are precisely defined.

A

grammar defines the syntax of legal sentences and semantic rules define the meaning.

Natural languages, such as English or Chinese, cannot be so neatly characterized

• Language judgments vary from person to person and time to time. Everyone agrees that

“Not to be invited is sad” is a grammatical sentence of English, but people disagree on

the grammaticality of “To be not invited is sad.”

• Natural language is ambiguous (“He saw her duck” can mean either that she owns a

waterfowl, or that she made a downwards evasive move) and vague (“That’s great!”

does not specify precisely how great it is, nor what it is).

• The mapping from symbols to objects is not formally defined. In first-order logic, two

uses of the symbol “Richard” must refer to the same person, but in natural language two

occurrences of the same word or phrase may refer to different things in the world.

We define a language model as a probability distribution describing the likelihood of any string.

**The bag-of-words model , N-gram word models**

**Word representations**

**Part-of-speech (POS) tagging**

**Grammar**

A grammar is a set of rules that defines the tree structure of allowable

phrases, and a language is the set of sentences that follow those rules.

Natural languages do not work exactly like the formal language of first-order logic—they

do not have a hard boundary between allowable and unallowable sentences, nor do they have a

single definitive tree structure for each sentence.

**Parsing**

Parsing is the process of analyzing a string of words to uncover its phrase structure, according

to the rules of a grammar.

**Natural Language Tasks**

<https://www.ibm.com/cloud/learn/natural-language-processing>