UNIT 4

PLANNING AND MACHINE LEARNING

4.1 Planning With State Space Search

The agent first generates a goal to achieve and then constructs a plan to achieve it from the Current state.

Problem Solving To Planning

Representation Using Problem Solving Approach

- ✓ Forward search
- ✓ Backward search
- ✓ Heuristic search

Representation Using Planning Approach

- ✓ STRIPS-standard research institute problem solver.
- ✓ Representation for states and goals
- ✓ Representation for plans
- ✓ Situation space and plan space
- ✓ Solutions

Why Planning?

Intelligent agents must operate in the world. They are not simply passive reasons (Knowledge Representation, reasoning under uncertainty) or problem solvers (Search), they must also act on the world.

We want intelligent agents to act in “intelligent ways”. Taking purposeful actions, predicting the expected effect of such actions, composing actions together to achieve complex goals. E.g. if we have a robot we want robot to decide what to do; how to act to achieve our goals.

Planning Problem

How to change the world to suit our needs

Critical issue: we need to reason about what the world will be like after doing a few actions, not just what it is like now

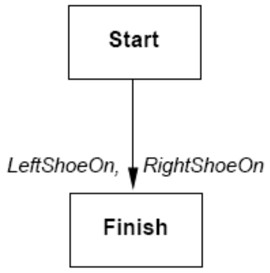
GOAL: Craig has coffee

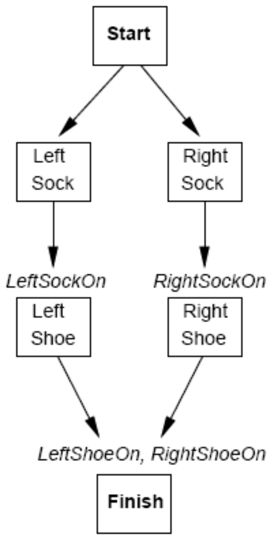
CURRENTLY: robot in mailroom, has no coffee, coffee not made, Craig in office etc.

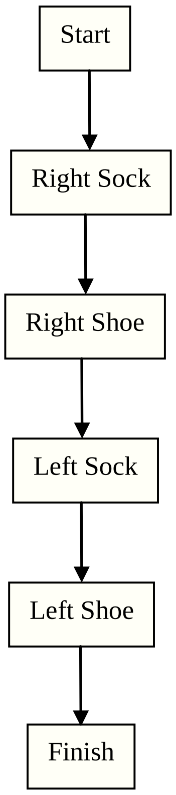
TO DO: goto lounge, make coffee

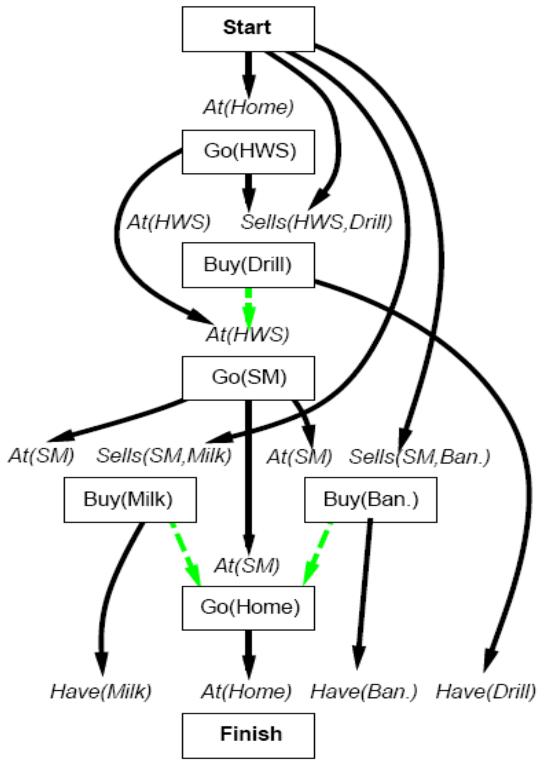
Partial Order Plan

- A partially ordered collection of steps
    - *Start step* has the initial state description and its effect
    - *Finish step* has the goal description as its precondition
    - *Causal links* from outcome of one step to precondition of another step
    - *Temporal ordering* between pairs of steps
- An open condition is a precondition of a step not yet causally linked
- A plan is ***complete*** if every precondition is achieved
- A precondition is ***achieved*** if it is the effect of an earlier step and no possibly intervening step undoes it









Partial Order Plan Algorithm

```
function POP(initial, goal, operators) returns plan
   plan ← MAKE-MINIMAL-PLAN(initial, goal)
   loop do
       if SOLUTION?(plan) then return plan
       $S_{need}$, c ← SELECT-SUBGOAL(plan)
       CHOOSE-OPERATOR(plan, operators, $S_{need}$, c)
       RESOLVE-THREATS(plan)
   end

function SELECT-SUBGOAL(plan) returns $S_{need}$, c
   pick a plan step $S_{need}$ from STEPS(plan)
       with a precondition c that has not been achieved
   return $S_{need}$, c
```

```
procedure CHOOSE-OPERATOR($plan, operators, S_{need}, c$)
    choose a step $S_{add}$ from $operators$ or STEPS($plan$) that has $c$ as an effect
    if there is no such step then fail
    add the causal link $S_{add} \xrightarrow{c} S_{need}$ to LINKS($plan$)
    add the ordering constraint $S_{add} \prec S_{need}$ to ORDERINGS($plan$)
    if $S_{add}$ is a newly added step from $operators$ then
        add $S_{add}$ to STEPS($plan$)
        add $Start \prec S_{add} \prec Finish$ to ORDERINGS($plan$)
```

```
procedure RESOLVE-THREATS($plan$)
    for each $S_{threat}$ that threatens a link $S_i \xrightarrow{c} S_j$ in LINKS($plan$) do
        choose either
            Demotion: Add $S_{threat} \prec S_i$ to ORDERINGS($plan$)
            Promotion: Add $S_j \prec S_{threat}$ to ORDERINGS($plan$)
        if not CONSISTENT($plan$) then fail
    end
```

4.2 Stanford Research Institute Problem Solver (STRIPS)

STRIPS is a classical planning language, representing plan components as states, goals, and actions, allowing algorithms to parse the logical structure of the planning problem to provide a solution.

In STRIPS, state is represented as a conjunction of positive literals. Positive literals may be a propositional literal (e.g., Big ^ Tall) or a first-order literal (e.g., At(Billy, Desk)). The positive literals must be grounded – may not contain a variable (e.g., At(x, Desk)) – and must be function-free – may not invoke a function to calculate a value (e.g., At(Father(Billy), Desk)). Any state conditions that are not mentioned are assumed false.

The goal is also represented as a conjunction of positive, ground literals. A state satisfies a goal if the state contains all of the conjuncted literals in the goal; e.g., Stacked ^ Ordered ^ Purchased satisfies Ordered ^ Stacked.

Actions (or operators) are defined by action schemas, each consisting of three parts:

- The action name and any parameters.
- Preconditions which must hold before the action can be executed. Preconditions are represented as a conjunction of function-free, positive literals. Any variables in a precondition must appear in the action's parameter list.
- Effects which describe how the state of the environment changes when the action is executed. Effects are represented as a conjunction of function-free literals. Any

variables in a precondition must appear in the action's parameter list. Any world state not explicitly impacted by the action schema's effect is assumed to remain unchanged.

The following, simple action schema describes the action of moving a box from location x to location y:

Action: *MoveBox*(*x*, *y*)
Precond: *BoxAt*(*x*)
Effect: *BoxAt*(*y*), ¬ *BoxAt*(*x*)

If an action is applied, but the current state of the system does not meet the necessary preconditions, then the action has no effect. But if an action is successfully applied, then any positive literals, in the effect, are added to the current state of the world; correspondingly, any negative literals, in the effect, result in the removal of the corresponding positive literals from the state of the world.

For example, in the action schema above, the effect would result in the proposition BoxAt(*y*) being added to the known state of the world, while BoxAt(*x*) would be *removed* from the known state of the world. (Recall that state only includes positive literals, so a negation effect results in the *removal* of positive literals.) Note also that positive effects can not get duplicated in state; likewise, a negative of a proposition that is not currently in state is simply ignored. For example, if *Open*(*x*) was not previously part of the state, ¬ *Open*(*x*) would have no effect.

A STRIPS problem includes the complete (but relevant) initial state of the world, the goal state(s), and action schemas. A STRIPS algorithm should then be able to accept such a problem, returning a solution. The solution is simply an action sequence that, when applied to the initial state, results in a state which satisfies the goal.

4.2.1 STRIPS Planning Algorithm

As previously referenced, STRIPS began as an automated planning algorithm and has double-meaning to describe the language (described above) used to provide input to that algorithm. While the algorithm does not scale well to real-world problems, it, like the language, serves as a foundational starting point to developing and understanding more powerful automated planning algorithms. The STRIPS algorithm [3] is found below, followed by a brief commentary:

```
STRIPS($A$, $s$, $g$)
$p$ = empty plan
loop…
if $s$ satisfies $g$ then return $p$
$a$ = [an applicable action in $A$, relevant for $g$]
if $a$ = null, then return failure
$p'$ = STRIPS($A$, $s$, precond($a$))
if $p'$ = failure, then return failure
$s$ = apply $p'$ to $s$
```

```
$s$ = apply $a$ to $s$
$p = p + p' + a$
```

In the above STRIPS algorithm, *A* represents all of the possible, grounded actions (i.e., action schemas with variables replaced with values), while *s* is the current state, and *g* is the goal state.

Critically important to note is that this algorithm is a *backward-search* algorithm. In other words, the goal state of the planning problem is fed into the algorithm as *s*, while the initial state of the planning problem is provided as *g*. The returned solution, if any, simply needs to be reversed to provide the solution to the planning problem. Simply put, it works backwards from the goal to figure out a logical path to get there.

4.3 Explanation

Most expert systems have explanation facilities that allow the user to ask questions- why and how it reached some conclusion

The questions are answered by referring to the system goals, the rules being used, and existing problem solving. The rules typically reflect empirical or “compiled” knowledge. They are codes of an expert’s rule of thumb, not the expert’s deeper understanding.

Example:

Dialog with an expert system designed to give advice on car problems.

System	Does the car start?
User	No.
System	Does the engine turn over?
User	yes
System	Do you smell gas?
User	yes
System	Recommendation-wait 5 minutes and try again.
User	Why?
System	I used the rule:
	If not start, and engine_turn_over, and smell_gas
	Then recommendation is ‘wait 5 minutes and try again’

User

Note: The rule gives the correct advice for a flooded car. And knows the questions to be ask to determine if the car is flooded, but it does not contain the knowledge of what a flooded car is and why waiting will help.

Types of Explanation

There are four types of explanations commonly used in expert systems.

- ✓ Rule trace reports on the progress of a consultation;
- ✓ Explanation of how the system reached to the give conclusion;
- ✓ Explanation of why the system did not give any conclusion.
- ✓ Explanation of why the system is asking a question;

4.4 Learning

Machine Learning

- ➢ Like human learning from past experiences,a computer does not have "experiences".
- ➢ A computer system learns from data, which represent some "past experiences" of an application domain.
- ➢ Objective of machine learning : learn a target function that can be used to predict the values of a discrete class attribute, e.g., approve or not-approved, and high-risk or low risk.
- ➢ The task is commonly called: **Supervised learning, classification, or inductive learning**

Supervised Learning

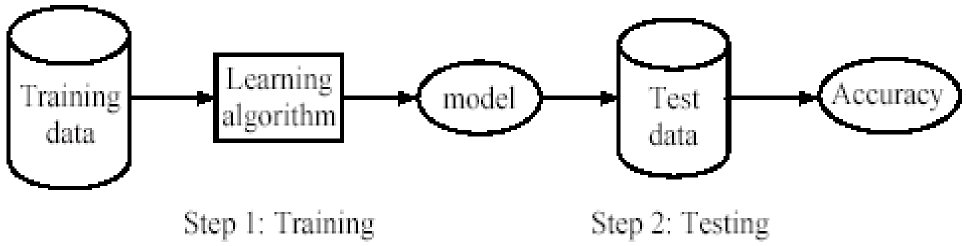
Supervised learning is a machine learning technique for learning a function from training data. The training data consist of pairs of input objects (typically vectors), and desired outputs. The output of the function can be a continuous value (called regression), or can predict a class label of the input object (called classification). The task of the supervised learner is to predict the value of the function for any valid input object after having seen a number of training examples (i.e. pairs of input and target output). To achieve this, the learner has to generalize from the presented data to unseen situations in a "reasonable" way.

Another term for supervised learning is classification. Classifier performance depend greatly on the characteristics of the data to be classified. There is no single classifier that works best on all given problems. Determining a suitable classifier for a given problem is however still more an art than a science. The most widely used classifiers are the Neural Network (Multi-layer Perceptron), Support Vector Machines, k-Nearest Neighbors, Gaussian Mixture Model, Gaussian, Naive Bayes, Decision Tree and RBF classifiers.

Supervised learning process: two steps

- **Learning** (training): Learn a model using the training data
- **Testing**: Test the model using unseen test data to assess the model accuracy

$$Accuracy = \frac{\text{Number of correct classifications}}{\text{Total number of test cases}},$$



Supervised vs. unsupervised Learning

- **Supervised learning**:

  classification is seen as supervised learning from examples.

  - ✓ **Supervision**: The data (observations, measurements, etc.) are labeled with pre-defined classes. It is like that a “teacher” gives the classes (supervision).
  - ✓ **Test data** are classified into these classes too.

- **Unsupervised learning** (clustering)

  - ✓ Class labels of the data are unknown
  - ✓ Given a set of data, the task is to establish the existence of classes or clusters in the data

Decision Tree

- A decision tree takes as input an object or situation described by a set of attributes and returns a “decision” – the predicted output value for the input.
- A decision tree reaches its decision by performing a sequence of tests.

  Example : “HOW TO” manuals (for car repair)

A decision tree reaches its decision by performing a sequence of tests. Each internal node in the tree corresponds to a test of the value of one of the properties, and the branches from the node are labeled with the possible values of the test. Each leaf node in the tree specifies the value to be returned if that leaf is reached. The decision tree representation seems to be very natural for humans; indeed, many "How To" manuals (e.g., for car repair) are written entirely as a single decision tree stretching over hundreds of pages.

A somewhat simpler example is provided by the problem of whether to wait for a table at a restaurant. The aim here is to learn a definition for the goal predicate Will Wait. In setting this up as a learning problem, we first have to state what attributes are available to describe examples in the domain. we will see how to automate this task; for now, let's suppose we decide on the following list of attributes:

1. Alternate: whether there is a suitable alternative restaurant nearby.

2. Bar: whether the restaurant has a comfortable bar area to wait in.

3. Fri/Sat: true on Fridays and Saturdays.

4. Hungry: whether we are hungry.

5. Patrons: how many people are in the restaurant (values are None, Some, and Full).

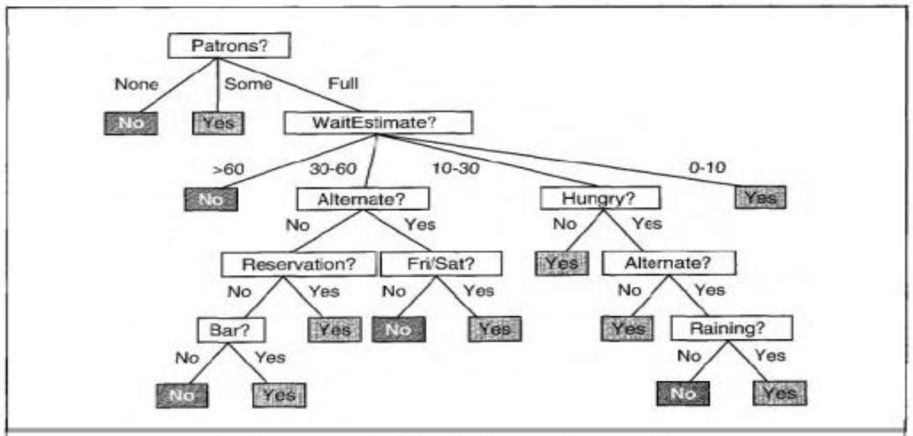
6. Price: the restaurant's price range ($, $$, $$$).

7. Raining: whether it is raining outside.

8. Reservation: whether we made a reservation.

9. Type: the kind of restaurant (French, Italian, Thai, or burger).

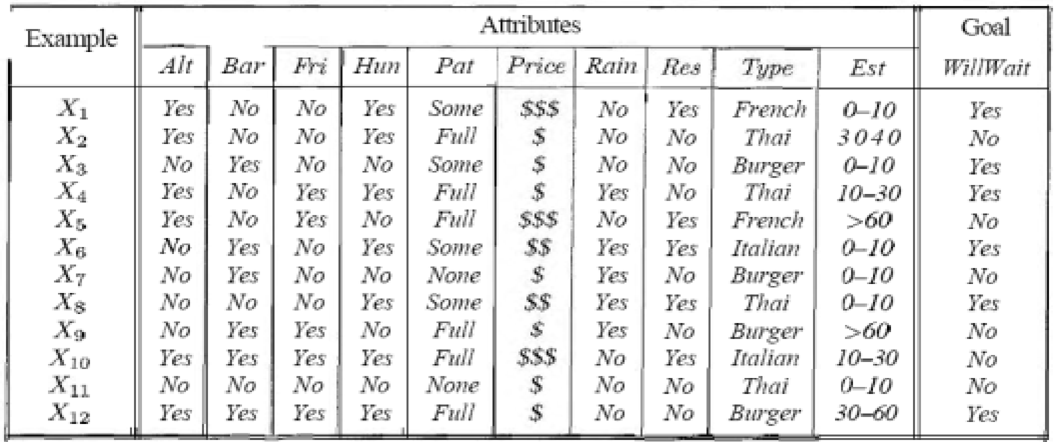
10. Wait Estimate: the wait estimated by the host (0-10 minutes, 10-30, 30-60, >60).



A decision tree for deciding whether to wait for a table.

Decision tree induction from examples

An example for a Boolean decision tree consists of a vector of' input attributes, X, and a single Boolean output value y. A set of examples (X1,Y1) . . . , (X2, y2) is shown in Figure. The positive examples are the ones in which the goal *Will Wait* is true (XI, ***X3,*** . . .); the negative examples are the ones in which it is false (X2, *X5*, . . .). The complete set of examples is called the **training set.**

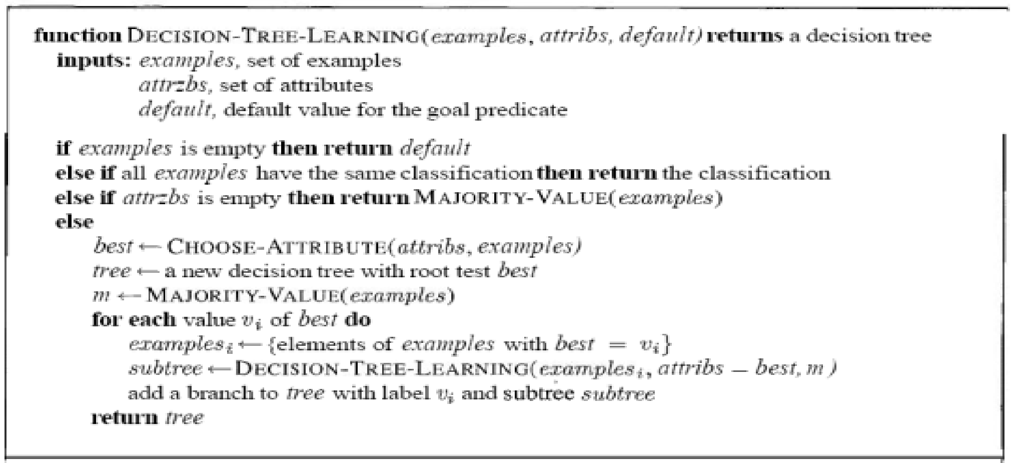


Example	Attributes										Goal
	Alt	*Bar*	*Fri*	*Hun*	*Pat*	*Price*	*Rain*	*Res*	*Type*	*Est*	*WillWait*
X_1	*Yes*	*No*	*No*	*Yes*	*Some*	*$$$*	*No*	*Yes*	*French*	*0–10*	*Yes*
X_2	*Yes*	*No*	*No*	*Yes*	*Full*	*$*	*No*	*No*	*Thai*	*3040*	*No*
X_3	*No*	*Yes*	*No*	*No*	*Some*	*$*	*No*	*No*	*Burger*	*0–10*	*Yes*
X_4	*Yes*	*No*	*Yes*	*Yes*	*Full*	*$*	*Yes*	*No*	*Thai*	*10–30*	*Yes*
X_5	*Yes*	*No*	*Yes*	*No*	*Full*	*$$$*	*No*	*Yes*	*French*	*>60*	*No*
X_6	*No*	*Yes*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Italian*	*0–10*	*Yes*
X_7	*No*	*Yes*	*No*	*No*	*None*	*$*	*Yes*	*No*	*Burger*	*0–10*	*No*
X_8	*No*	*No*	*No*	*Yes*	*Some*	*$$*	*Yes*	*Yes*	*Thai*	*0–10*	*Yes*
X_9	*No*	*Yes*	*Yes*	*No*	*Full*	*$*	*Yes*	*No*	*Burger*	*>60*	*No*
X_{10}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$$$*	*No*	*Yes*	*Italian*	*10–30*	*No*
X_{11}	*No*	*No*	*No*	*No*	*None*	*$*	*No*	*No*	*Thai*	*0–10*	*No*
X_{12}	*Yes*	*Yes*	*Yes*	*Yes*	*Full*	*$*	*No*	*No*	*Burger*	*30–60*	*Yes*

Examples for the restaurant domain.

Decision Tree Algorithm

The basic idea behind the Decision-Tree-Learning-Algorithm is to test the most important attribute first. By "most important," we mean the one that makes the most difference to the classification of an example. That way, we hope to get to the correct classification with a small number of tests, meaning that all paths in the tree will be short and the tree as a whole will be small.



```
function DECISION-TREE-LEARNING(examples, attribs, default) returns a decision tree
  inputs: examples, set of examples
          attrzbs, set of attributes
          default, default value for the goal predicate

  if examples is empty then return default
  else if all examples have the same classification then return the classification
  else if attrzbs is empty then return MAJORITY-VALUE(examples)
  else
      best ← CHOOSE-ATTRIBUTE(attribs, examples)
      tree ← a new decision tree with root test best
      m ← MAJORITY-VALUE(examples)
      for each value $v_i$ of best do
          $examples_i$ ← {elements of examples with best = $v_i$}
          subtree ← DECISION-TREE-LEARNING($examples_i$, attribs – best, m )
          add a branch to tree with label $v_i$ and subtree subtree
      return tree
```

The decision tree learning algorithm.

Reinforcement Learning

- ➢ Learning what to do to maximize reward
  - ✓ Learner is not given training
  - ✓ Only feedback is in terms of reward
  - ✓ Try things out and see what the reward is
- ➢ Different from Supervised Learning
  - ✓ Teacher gives training examples

Examples

- ➢ Robotics: Quadruped Gait Control, Ball Acquisition (Robocup)
- ➢ Control: Helicopters
- ➢ Operations Research: Pricing, Routing, Scheduling
- ➢ Game Playing: Backgammon, Solitaire, Chess, Checkers
- ➢ Human Computer Interaction: Spoken Dialogue Systems
- ➢ Economics/Finance: Trading

Markov decision process VS Reinforcement Learning

- ➢ Markov decision process

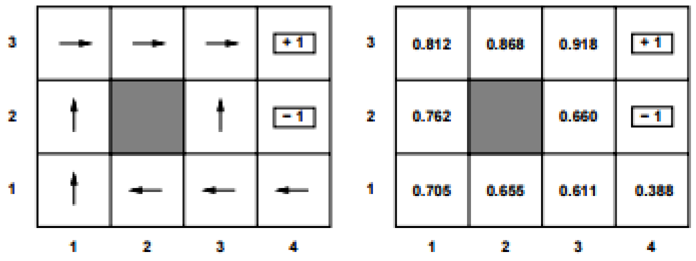
- ✓ Set of state S, set of actions A
- ✓ Transition probabilities to next states T(s, a, a')
- ✓ Reward functions R(s)

- ➢ RL is based on MDPs, but
  - ✓ Transition model is not known
  - ✓ Reward model is not known
- ➢ MDP computes an optimal policy
- ➢ RL learns an optimal policy

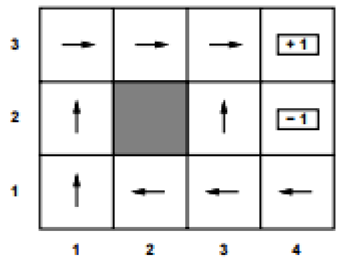
Types of Reinforcement Learning

- ➢ Passive Vs Active
  - ✓ Passive: Agent executes a fixed policy and evaluates it
  - ✓ Active: Agents updates policy as it learns
- ➢ Model based Vs Model free
- ➢ Model-based: Learn transition and reward model, use it to get optimal policy
- ➢ Model free: Derive optimal policy without learning the model

Passive Learning



- ➢ Evaluate how good a policy π is
- ➢ Learn the utility $U^{\pi}(s)$ of each state
- ➢ Same as policy evaluation for known transition & reward models



Agent executes a sequence of trials:

$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$
$(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 2) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$
$(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (4, 2)_{-1}$

Goal is to learn the expected utility $U_{\pi}(s)$

$$U^{\pi}(s) = E\left[\sum_{t=0}^{\infty} \gamma^{t} R(s_t) | \pi, s_0 = s\right]$$

Direct Utility Estimation

- Reduction to inductive learning
  - ✓ Compute the empirical value of each state
  - ✓ Each trial gives a sample value
  - ✓ Estimate the utility based on the sample values
- Example: First trial gives
  - ✓ State (1,1): A sample of reward 0.72
  - ✓ State (1,2): Two samples of reward 0.76 and 0.84
  - ✓ State (1,3): Two samples of reward 0.80 and 0.88
- Estimate can be a running average of sample values
- Example: U(1, 1) = 0.72,U(1, 2) = 0.80,U(1, 3) = 0.84, . . .
- Ignores a very important source of information

- The utility of states satisfy the Bellman equations

$$U^{\pi}(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') U^{\pi}(s')$$

- Search is in a hypothesis space for U much larger than needed
- Convergence is very slow
- Make use of Bellman equations to get $U^{\pi}(s)$
- Need to estimate T(s, π(s), s') and R(s) from trials
- Plug-in learnt transition and reward in the Bellman equations
- Solving for U^{π}: System of n linear equations
- Estimates of T and R keep changing
- Make use of modified policy iteration idea
    - ✓ Run few rounds of value iteration
    - ✓ Initialize value iteration from previous utilities
    - ✓ Converges fast since T and R changes are small
- ADP is a standard baseline to test 'smarter' ideas
- ADP is inefficient if state space is large
    - ✓ Has to solve a linear system in the size of the state space
    - ✓ Backgammon: 10^{50} linear equations in 10^{50} unknowns

Temporal Difference Learning

- Best of both worlds
    - ✓ Only update states that are directly affected
    - ✓ Approximately satisfy the Bellman equations
    - ✓ Example:

      $(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$

      $(1, 1) \rightarrow (1, 2) \rightarrow (1, 3) \rightarrow (2, 3) \rightarrow (3, 3) \rightarrow (3, 2) \rightarrow (3, 3) \rightarrow (4, 3)_{+1}$

$(1, 1) \rightarrow (2, 1) \rightarrow (3, 1) \rightarrow (3, 2) \rightarrow (4, 2)_{-1}$

- After the first trial, U(1, 3) = 0.84,U(2, 3) = 0.92
- Consider the transition (1, 3) → (2, 3) in the second trial
- If deterministic, then U(1, 3) = −0.04 + U(2, 3)
- How to account for probabilistic transitions (without a model)

➢ TD chooses a middle ground

$$U^{\pi}(s) \leftarrow (1-\alpha)U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s'))$$

➢ Temporal difference (TD) equation, α is the learning rate

➢ The TD equation

$$U^{\pi}(s) \leftarrow U^{\pi}(s) + \alpha(R(s) + \gamma U^{\pi}(s') - U^{\pi}(s))$$

➢ TD applies a correction to approach the Bellman equations

✓ The update for s' will occur T(s, π(s), s') fraction of the time

✓ The correction happens proportional to the probabilities

✓ Over trials, the correction is same as the expectation

➢ Learning rate α determines convergence to true utility

✓ Decrease α_s proportional to the number of state visits

✓ Convergence is guaranteed if

$$\sum_{m=1}^{\infty} \alpha_s(m) = \infty \qquad \sum_{m=1}^{\infty} \alpha_s^2(m) < \infty$$

✓ Decay α_s (m) = 1/m satisfies the condition

➢ TD is model free

TD Vs ADP

➢ TD is mode free as opposed to ADP which is model based

➢ TD updates observed successor rather than all successors

- ➢ The difference disappears with large number of trials
- ➢ TD is slower in convergence, but much simpler computation per observation

Active Learning

- ➢ Agent updates policy as it learns
- ➢ Goal is to learn the optimal policy
- ➢ Learning using the passive ADP agent
  - ✓ Estimate the model R(s),T(s, a, s’) from observations
  - ✓ The optimal utility and action satisfies

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')U(s')$$

  - ✓ Solve using value iteration or policy iteration

- ➢ Agent has “optimal” action
- ➢ Simply execute the “optimal” action

Exploitation vs Exploration

- ➢ The passive approach gives a greedy agent
- ➢ Exactly executes the recipe for solving MDPs
- ➢ Rarely converges to the optimal utility and policy
  - ✓ The learned model is different from the true environment
- ➢ Trade-off
  - ✓ Exploitation: Maximize rewards using current estimates
  - ✓ Agent stops learning and starts executing policy
  - ✓ Exploration: Maximize long term rewards
  - ✓ Agent keeps learning by trying out new things
- ➢ Pure Exploitation

  - ✓ Mostly gets stuck in bad policies
- ➢ Pure Exploration
  - ✓ Gets better models by learning
  - ✓ Small rewards due to exploration
- ➢ The multi-armed bandit setting
  - ✓ A slot machine has one lever, a one-armed bandit
  - ✓ n-armed bandit has n levers
- ➢ Which arm to pull?
  - ✓ Exploit: The one with the best pay-off so far
  - ✓ Explore: The one that has not been tried

Exploration

- ➢ Greedy in the limit of infinite exploration (GLIE)
  - ✓ Reasonable schemes for trade off
- ➢ Revisiting the greedy ADP approach
  - ✓ Agent must try each action infinitely often
  - ✓ Rules out chance of missing a good action
  - ✓ Eventually must become greedy to get rewards
- ➢ Simple GLIE
  - ✓ Choose random action 1/t fraction of the time
  - ✓ Use greedy policy otherwise
- ➢ Converges to the optimal policy
- ➢ Convergence is very slow

Exploration Function

- A smarter GLIE
  - ✓ Give higher weights to actions not tried very often
  - ✓ Give lower weights to low utility actions
- Alter Bellman equations using optimistic utilities $U^+(s)$

$$U^+(s) = R(s) + \gamma \max_a f\left(\sum_{s'} T(s,a,s')U^+(s'), N(a,s)\right)$$

- The exploration function f (u, n)
  - ✓ Should increase with expected utility u
  - ✓ Should decrease with number of tries n
- A simple exploration function

$$f(u,n) = \begin{cases} R^+, & \text{if } n < N \\ u, & \text{otherwise} \end{cases}$$

- Actions towards unexplored regions are encouraged
- Fast convergence to almost optimal policy in practice

Q-Learning

- Exploration function gives a active ADP agent
- A corresponding TD agent can be constructed
  - ✓ Surprisingly, the TD update can remain the same
  - ✓ Converges to the optimal policy as active ADP
  - ✓ Slower than ADP in practice
- Q-learning learns an action-value function Q(a; s)
  - ✓ Utility values $U(s) = \max_a Q(a; s)$
- A model-free TD method
  - ✓ No model for learning or action selection

- Constraint equations for Q-values at equilibrium

$$Q(a,s) = R(s) + \gamma \sum_{s'} T(s,a,s') \max_{a'} Q(a',s')$$

- Can be updated using a model for T(s; a; s')
- The TD Q-learning does not require a model

$$Q(a,s) \leftarrow Q(a,s) + \alpha(R(s) + \gamma \max_{a'} Q(a',s') - Q(a,s))$$

- Calculated whenever a in s leads to s'
- The next action a_{next} = argmax$_{a'}$ f (Q(a'; s');N(s'; a'))
- Q-learning is slower than ADP
- Trade-o: Model-free vs knowledge-based methods

PART- A

1. What are the components of planning system?
2. What is planning?
3. What is nonlinear plan?
4. List out the 3 types of machine learning?
5. What is Reinforcement Learning?
6. What do you mean by goal stack planning?
7. Define machine learning.
8. What are the types of Reinforcement Learning.

PART B

1. Briefly explain the advanced plan generation systems.

2. Explain Machine Learning.
3. Explain STRIPS.
4. Explain Reinforcement Learning.
5. Briefly explain Partial Order Plan.
6. Explain in detail about various Machine learning methods.