B.E./B.Tech. DEGREE EXAMINATION, APRIL/MAY 2008.
Seventh Semester
Computer Science and Engineering
CS 2351 – ARTIFICIAL INTELLIGENCE
(Regulation 2004)

Time: Three hours Maximum: 100 marks

Answer ALL questions.

PART A — (10 × 2 = 20 marks)

1. Define artificial intelligence.

Artificial Intelligence is the study of how to make computers do things which, at the moment, people do better.

2. What is the use of heuristic functions?

A *heuristic* is a function, h(n) defined on the nodes of a search tree, which serves as an estimate of the cost of the cheapest path from that node to the goal node. Heuristics are used by informed search algorithms such as Greedy best-first search and A*

3. How to improve the effectiveness of a search-based problem-solving technique?

- Goal formulation
- Problem formulation
- Search
- Solution
- Execution phase

4. What is a constraint satifcation problem?

Constraint satisfaction problems (CSPs) are mathematical problems defined as a set of objects whose state must satisfy a number of constraints or limitations. CSPs represent the entities in a problem as a homogeneous collection of finite constraints over variables, which is solved by constraint satisfaction methods.

5. What is unification algorithm?

Lifted inference rules require finding substitutions that make different logical expressions UNIFICATION look identical. This process is called unification and is a key component of all first-order UNIFIER inference algorithms.

6. How can you represent the resolution in predicate logic?

• Constant symbols: a, b, c, John, …

to represent primitive objects
• Variable symbols: x, y, z, …
to represent unknown objects
• Predicate symbols: safe, married, love, …
to represent relations
married(John)
love(John, Mary)

7. List out the advantages of nonmonotonic reasoning.

The global advantages of monotonicity should not be casually tossed aside, but at the same time the computational advantages of nonmonotonic reasoning modes is hard to deny, and they are widely used in the current state of the art. We need ways for them to co-exist smoothly.

8. Differentiate between JTMS and LTMS.

In a JTMS, each sentence in the knowledge base is annotated with a justification consisting of the set of sentences from which it was inferred. It is a simple TMS where one can examine the consequences of the current set of assumptions. The meaning of sentences is not known.

Like JTMS in that it reasons with only one set of current assumptions at a time. More powerful than JTMS in that it recognises the propositional semantics of sentences, i.e. understands the relations between p and ~p, p and q and p&q, and so on**.**

9. What are framesets and instances?

An individual object of a certain class. While a class is just the type definition, an actual usage of a class is called "instance". Each instance of a class can have different values for its instance variables.

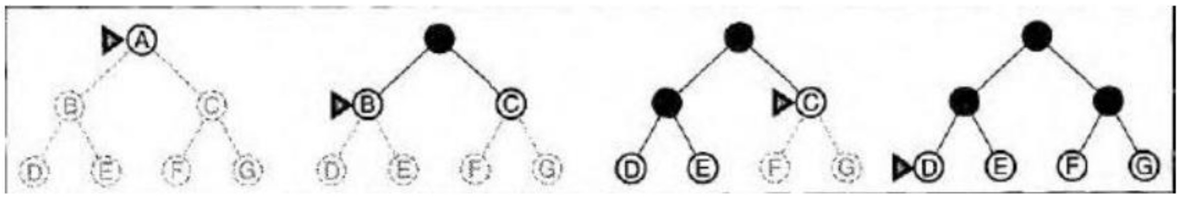
10. List out the important components of a script.

PARTB-(5 X 16 = 80 marks)

11. (a) (i) Given an example of a problem for which breadth-first search would work better than depth-first search. (8)

Breadth first search

Breadth first search is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on. In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.
Simple calling TREE-SEARCH (problem, FIFO-QUEUE ( )) results in a breadth first search.
Figure: Breadth first search on a simple binary



Time complexity, the total number of nodes generated is

B + b2 + b3 +…. +bd + bd+1 – b) = O (bd+1)

Every node that is generated must remain in memory. The space complexity is same as the time complexity.

Adv: Guaranteed to find the single solution at the shallowest depth level.

Disadv: **I)**The memory requirements are a bigger problem for breadth-first search than is the execution time.

II) Exponential-complexity search problems cannot be solved by uninformed methods for any but only suitable for smallest instances problem (i.e.) (number of levels to be minimum (or) branching factor to be minimum)

(ii) Explain the algorithm for steepest hill climbing. (8)

In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighbouring configuration) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization and tabu search, on memory-less stochastic modifications, like simulated annealing.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends.

In simple hill climbing, the first closer node is chosen, whereas in steepest ascent hill climbing all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space which are not solutions. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.

(b) Explain the following search strategies.

(i) Best-first search (8)

Best first search

Best first search is an instance of the general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an evaluation function, f(n).
A Key component of these algorithms is a heuristic function, h (n):

h (n) = estimated cost of the cheapest path from node n to a goal node.

The two types of evaluation functions are:

Expand the node closest to the goal state using estimated cost as the evaluation is called **Greedy best-first search**.
Expand the node on the least cost solution path using estimated cost and actual cost as the evaluation function is called **A* search**.

(ii) A* search (8)

A* search: Minimizing the total estimated solution cost
Expand the node on the least cost solution path using estimated cost and actual cost as the evaluation function is called **A* search.** It evaluates nodes by combining g (n) , the cost to reach the node, and h (n), the cost to get from the node to the goal:
f (n) = g (n) + h (n).
since g (n) gives the path cost from the start node to node n, and h (n) is the estimated cost of the cheapest path from n to the goal, we have
f (n) = estimated cost of the cheapest solution through n.
A* search is both complete and optimal.

Monotonicity (consistency): In search tree any path from the root, the f-cost never decreases. This condition is true for almost all admissible heuristics. A heuristic which satisfies this property is called monotonicity.
Optimality: It is derived with two approaches. They are a) A* used with Tree-search b) A* used with Graph-search.
Memory bounded heuristic search
The memory requirements of A* is reduced by combining the heuristic function with I   terative deepening resulting an IDA* algorithm. The main difference between IDA* and standard iterative deepening is that the cutoff used is the f-cost(g+h) rather than the depth; at each iteration, the cutoff value is the smallest f-cost of any node that exceeded the cutoff on the previous iteration. The main disadvantage is, it will require more storage space in complex domains.

The two recent memory bounded algorithms are:

1. Recursive best-first search(RBFS)

2. Memory bounded A* search (MA*)

Recursive best-first search (RBFS)
RBFS is simple recursive algorithm uses only linear space. The algorithm is as follows:

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
RBFS (problem, MAKE-NODE (INITIAL-STATE [problem]), ∞)
function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
if GOAL-TEST[problem](state) then return node
successors ← EXPAND (node, problem)
if successors is empty then return failure, ∞

for each s in successors do
f[s] ← max(g(s) + h(s).f[node])
repeat
best ← the lowest f-value node in successors
if f[best] > f_limit then return failure, f[best]
alternative ← the second-lowest f-value among successors
result, f[best] ← RBFS(problem, best, min(f_limit, alternative))
if result ≠ failure then return result
```

Figure 2.1 The algorithm for recursive best-first search

Its **time complexity** depends both on the accuracy of the heuristic function and on how often he best path changes as nodes are expanded. Its **space complexity** is O (bd), even though more memory is available.

Search techniques which use all available memory are:

1) MA* (Memory-bounded A*)
2) SMA* (Simplified MA*)

SMA* (Simplified MA*)
It can make use of all available memory to carry out the search.

Properties: i) It will utilize whatever memory is made available to it.
ii) It avoids repeated states as far as its memory allows.

It is **complete** if the available memory is sufficient to store the deepest solution path.
It is **optimal** if enough memory is available to store the deepest solution path. Otherwise, it returns the best solution that can be reached with the available memory.

Advantage: SMA* uses only the available memory.

Disadvantage: If enough memory is not available it leads to suboptimal solution.
Space and Time complexity: depends on the available number of nodes.

12. (a) Explain Min-Max search procedure. (16)

The Minimax algorithm

The Minimax algorithm computes the minimax decision from the current state. It performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is m, and there are b legal moves at each point, then the time complexity of the minimax algorithm is O (bm).

Minimax is for deterministic games with perfect information. The **minimax** algorithm generates the whole game tree and applies the utility function to each terminal state. Then it propagates the utility value up one level and continues to do so until reaching the start node.

The minimax algorithm is as follows:

```
function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game
v ← MAX-VALUE(state)
return the action in SUCCESSORS(state) with value v
function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
V ← -∞
for a, s in SUCCESSORS(state) do
v ← MAX(v, MIN-VALUE(s))
return v
function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
v ← ∞
for a, s in SUCCESSORS(state) do

v ← MIN(v, MAX-VALUE(s))
return v
```

Figure: An algorithm for calculating minimax decisions

(b) Describe Alpha-Beta pruning and give the other modifications to minmax procedure to improve its performance.

Alpha-Beta pruning
Pruning: The process of eliminating a branch of the search tree from consideration without examining is called pruning. The two parameters of pruning technique are:

1. **Alpha (α):** Best choice for the value of MAX along the path or lower bound on the value that on maximizing node may be ultimately assigned.

2. **Beta (β)**: Best choice for the value of MIN along the path or upper bound on the value that a minimizing node may be ultimately assigned.

Alpha-Beta Pruning: The alpha and beta values are applied to a minimax tree, it returns the same move as minimax, but prunes away branches that cannot possibly influence the final decision is called **Alpha-Beta pruning** or **Cutoff**.
Consider the two ply game tree from figure.

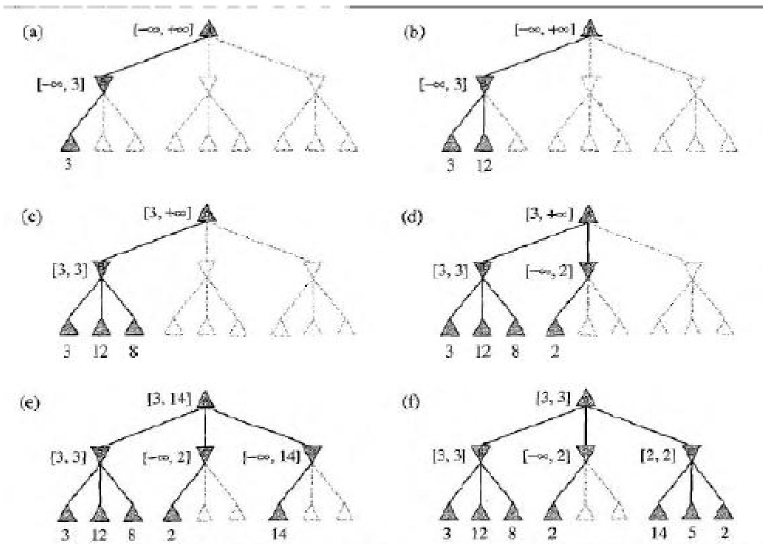


Figure Stages in the calculation of the optimal decision for the game tree in Figure 6.2. At each point, we show the range of possible values for each node. (a) The first leaf below B has the value 3. Hence, B, which is a MIN node, has a value of *at most* 3. (b) The second leaf below B has a value of 12; MIN would avoid this move, so the value of B is still at most 3. (c) The third leaf below B has a value of 8; we have seen all B's successors, so the value of *B* is exactly 3. Now, we can infer that the value of the root is *at least* 3, because MAX has a choice worth 3 at the root. (d) The first leaf below C has the value 2. Hence, *C*, which is a MIN node, has a value of *at most* 2. But we know that *B* is worth 3, so MAX would never choose C. Therefore, there is no point in looking at the other successors of C. This is an example of alpha–beta pruning. (e) The first leaf below D has the value 14, so D is worth *at most* 14. This is still higher than MAX's best alternative(i.e., 3), so we need to keep exploring D's successors. Notice also that we now have bounds on all of the successors of the root, so the root's value is also at most 14. (f) The second successor of D is worth 5, so again we need to keep exploring. The third successor is worth 2, so now D is worth exactly 2. MAX's decision at the root is to move to B, giving a value of 3.

Alpha –beta pruning can be applied to trees of any depth, and it is often possible to prune entire sub trees rather than just leaves.

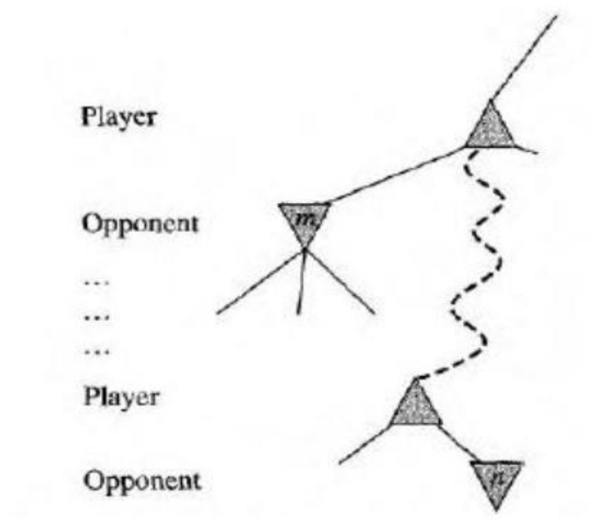
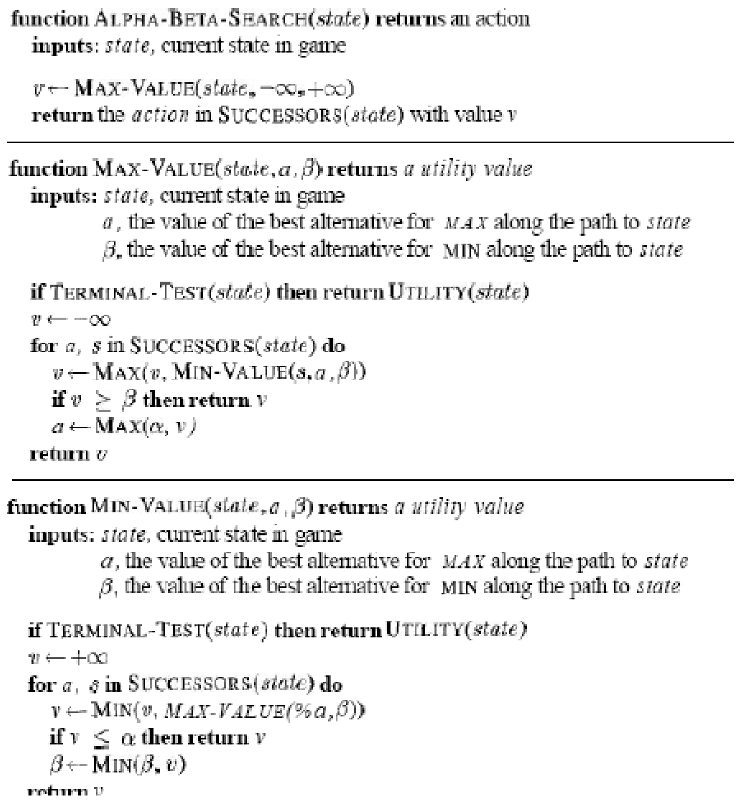


Figure Alpha-beta pruning: the general case. If m is better than n for player, we will never get to n in play.



```
function ALPHA-BETA-SEARCH(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state, −∞, +∞)
   return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state

   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← +∞
   for a, s in SUCCESSORS(state) do
      v ← MIN(v, MAX-VALUE(%a, β))
      if v ≤ α then return v
      β ← MIN(β, v)
   return v
```

Figure: The alpha-beta search algorithm

Effectiveness of Alpha – Beta pruning

It needs to examine only nodes to pick the best move.

13.(a) Illustrate the use of predicate logic to represent the knowledge with suitable example.

The organization of objects into **categories** is a vital part of knowledge representation. Although interaction with the world takes place at the level of individual objects, *much reasoning takes place at the level of categories*.There are two chances for representing categories in first-order logic: **predicates** and **objects**.

Measurements
Kn both scientific and commonsense theories of the world, objects have height, mass, cost,and so on. The values that we assign for these properties are called **measures.** Ordinary quantitative measures are quite easy to represent. We imagine that the universe includes abstract "measure objects," such as the *length* that is the length of this line segment:

Actions, Situations, and Events

The ontology of situation calculus
One obvious way to avoid multiple copies of axioms is simply to quantify over time-to say, *" V t* , such-and-such is the result at *t* + 1 of doing the action at *t."* Instead of dealing with explicit times like *t* 4 1, we will concentrate in this section on ***situations,*** which denote the states resulting from executing actions. This approach is called **situation calculus** and involves the following ontology:

Actions are logical terms such as ***Forward*** and ***Turn*** *(Right).* For now, we will assume that the environment contains only one agent. (If there is more than one, an additional argument can be inserted to say which agent is doing the action.)

Situations are logical terms consisting of the initial situation (usually called *So)* and all situations that are generated by applying an action to a situation. The function *Result(a, s)* (sometimes called *Do)* names the situation that results when action *a* is executed in situation *s*. Figure 3.11 illustrates this idea.

Fluent are functions and predicates that vary from one situation to the next, such as the location of the agent or the aliveness of the wumpus. The dictionary says a fluent
is something that flows, like a liquid. In this use, it means flowing or changing across situations. By convention, the situation is always the last argument of a fluent. For example, *lHoldzng(G1, So)* says that the agent is not holding the gold GI in the initial situation *So*. *Age (Wumpus, So)* refers to the wumpus's age in *So.*
A temporal or **eternal** predicates and functions are also allowed. Examples include the predicate Gold (GI) and the function *Left Leg Of* (Wumpus*).*
A situation calculus agent should be able to deduce the outcome of a given sequence of PROJECTION actions; this is the **projection** task. With a suitable constructive inference algorithm, it should also be able to join a sequence that achieves a desired effect; this is the **planning** task.

Describing actions in situation calculus

In the simplest version of situation calculus, each action is described by two axioms: a **possibility axiom** that says when it is possible to execute the action, and an **effect axiom** that says EFFECT AXIOM what happens when a possible action is executed.
The axioms have the following form:

POSSIBILITY AXIOM: *Preconditions + Poss(a, s) .*
EFFECT AXIOM: *Poss(a, s) + Changes that result from taking action.*

The problem is that the effect axioms say what changes, but don't say what stays the same.
Representing all the things that stay the same is called the **frame problem.** We must find an efficient solution to the frame problem because, in the real world, almost everything stays the same almost all the time. Each action affects only a tiny fraction of all fluent.
One approach is to write explicit **frame axioms** that *do* say what stays the same.

Solving the representational frame problem
The solution to the representational frame problem involves just a slight change in viewpoint on how to write the axioms. Instead of writing out the effects of each action

we consider how each fluent predicate evolves over time.3 The axioms we use are called **successor-state axioms.** They have the following form:
AXIOM SUCCESSOR-STATE AXIOM:
Action is possible + (Fluent is true in result state # Action S *effect made it true*
V *It was true before and action left it alone)* .The *unique names axiom* states a disqualify for every pair of constants in the knowledge base.

Solving the inferential frame problem
To solve the inferential frame problem, we have two possibilities. First, we wo*uld* discard situation calculus and invent a new formalism for writing axioms. This has been done with formalisms such as *1* his *fluent calculus.* Second, we could alter the inference mechanism to handle frame axioms rnose efficiently.

Time and event calculus
The Initiates and Terminates relations play a role similar to the Result relation in situation calculus; Initiates(e, f , t) means that the occurrence of event e at time t causes fluent f to become true, while Terminates (w , f , t) means that f ceases to be true. We use Happens(e, t) to mean that event e happens at time t , and we use Clipped(f , t , t2) to mean that f is terminated by some event sometime between t and t2. Formally, the axiom is:

Generalized events
A generalized event is composed from aspects of some "space-time chunk"--a piece of this multidimensional space-time universe. This extraction generalizes most of the concepts we have seen so far, including actions, locations, times, fluent, and physical objects.

(b) Consider the following sentences:
- **John likes all kinds of food.**
- **Apples are food.**
- **Chicken is food.**

- **Anything anyone eats and isn't killed by is food.**
- **Bill eats peanuts and is still alive.**
- **Sue eats everything Bill eats.**

(i) Translate these sentences into formulas in predicate logic.
(ii) Prove that John likes peanuts using backward chaining.
(iii) Convert the formulas of a part in to clause form
(iv) Prove that John likes peanuts using resolution.

14. (a) With an example explain the logics for nonmonotonic reasoning.

The definite clause logic is monotonic in the sense that anything that could be concluded before a clause is added can still be concluded after it is added; adding knowledge does not reduce the set of propositions that can be derived.
A logic is non-monotonic if some conclusions can be invalidated by adding more knowledge. The logic of definite clauses with negation as failure is non-monotonic. Non-monotonic reasoning is useful for representing defaults. A default is a rule that can be used unless it overridden by an exception.
For example, to say that *b* is normally true if *c* is true, a knowledge base designer can write a rule of the form
b ←c ∧~ab_a.
where ab_a is an atom that means abnormal with respect to some aspect *a*. Given *c*, the agent can infer *b*unless it is told ab_a. Adding ab_a to the knowledge base can prevent the conclusion of *b*. Rules that implyab_a can be used to prevent the default under the conditions of the body of the rule.

Example: Suppose the purchasing agent is investigating purchasing holidays. A resort may be adjacent to a beach or away from a beach. This is not symmetric; if the resort was adjacent to a beach, the knowledge provider would specify this. Thus, it is reasonable to have the clause
away_from_beach ←~on_beach.
This clause enables an agent to infer that a resort is away from the beach if the agent is not told it is adjacent to a beach.
A cooperative system tries to not mislead. If we are told the resort is on the beach, we would expect that resort users would have access to the beach. If they have access to a beach, we would expect them to be able to swim at the beach. Thus, we would expect the following defaults:
$beach\_access \leftarrow on\_beach \wedge \sim ab_{beach\_access}.$
$swim\_at\_beach \leftarrow beach\_access \wedge \sim ab_{swim\_at\_beach}.$
A cooperative system would tell us if a resort on the beach has no beach access or if there is no swimming. We could also specify that, if there is an enclosed bay and a big city, then there is no swimming, by default:
$ab_{swim\_at\_beach} \leftarrow enclosed\_bay \wedge big\_city \wedge \sim ab_{no\_swimming\_near\_city}.$
We could say that British Columbia is abnormal with respect to swimming near cities:
$ab_{no\_swimming\_near\_city} \leftarrow in\_BC \wedge \sim ab_{BC\_beaches}.$
Given only the preceding rules, an agent infers *away_from_beach*. If it is then told *on_beach*, it can no longer infer *away_from_beach*, but it can now infer *beach_access* and *swim_at_beach*. If it is also

told *enclosed_bay* and *big_city*, it can no longer infer *swim_at_beach*. However, if it is then told*in_BC*, it can then infer *swim_at_beach*.
By having defaults of what is normal, a user can interact with the system by telling it what is abnormal, which allows for economy in communication. The user does not have to state the obvious.
One way to think about non-monotonic reasoning is in terms of arguments. The rules can be used as components of arguments, in which the negated abnormality gives a way to undermine arguments. Note that, in the language presented, only positive arguments exist that can be undermined. In more general theories, there can be positive and negative arguments that attack each other.

(b) Explain how Bayesian statistics provides reasoning under various kinds of uncertainty.

A statistical learning method begins with the simplest task: parameter learning with complete data. A parameter learning task involves finding the numerical parameters for a probability model whose structure is fixed.

Maximum-likelihood parameter learning: Discrete models

In fact, though, we have laid out one standard method for maximum-likelihood parameter learning:
1. Write down an expression for the likelihood of the data as a function of the parameter(s).
2. Write down the derivative of the log likelihood with respect to each parameter.
3. Find the parameter values such that the derivatives are zero.

A significant problem with maximum-likelihood learning in general: ―*when the data set is small enough that some events have not yet been observed-for instance, no cherry candies-the maximum Likelihood hypothesis assigns zero probability to those events".*
The most important point is that, *with complete data, the maximum-likelihood parameter learning problem for a Bayesian network decomposes into separate learning problems, one for each parametez3*. The second point is that the parameter values for a variable, given its parents, are just the observed frequencies of the variable values for each setting of the parent values. As before, we must be careful to avoid zeroes when the data set is small.

Naive Bayes models

Probably the most common Bayesian network model used in machine learning is the **naïve Bayes** model. In this model, the "class" variable C (which is to be predicted) is the root and the "attribute" variables *Xi* are the leaves. The model is "naive7' because it assumes that the attributes are conditionally independent of each other, given the class.

Maximum-like likelihood parameter learning: Continuous models

Continuous probability models such as the **linear-Gaussian** model. The principles for maximum likelihood learning are identical to those of the discrete case. Let us begin with a very simple case: learning the parameters of a Gaussian density function on a single variable. That is, the data are generated also follows:
The parameters of this model are the mean, Y and the standard deviation *a*.

The quantity (yj - (B1xj + 02)) is the **error** for (zj, yj)-that is, the difference between the actual value yj and the predicted value (01 x j + $2)-SO E is the well-known **sum of squared errors.** This is the quantity that is minimized by the standard **linear regression** procedure. Now we can understand why: minimizing the sum of squared errors gives the maximum likelihood straight-line model, *provided that the data are generated with Gaussian noise of* fixed *variance.*

Bayesian parameter learning

The Bayesian approach to parameter learning places a hypothesis prior over the possible values of the parameters and updates this distribution as data arrive. This formulation of learning and prediction makes it clear that Bayesian learning requires no extra "principles of learning." Furthermore, *there is, in essence, just one learning algorithm,* i.e., the inference algorithm for Bayesian networks.

Learning net structures

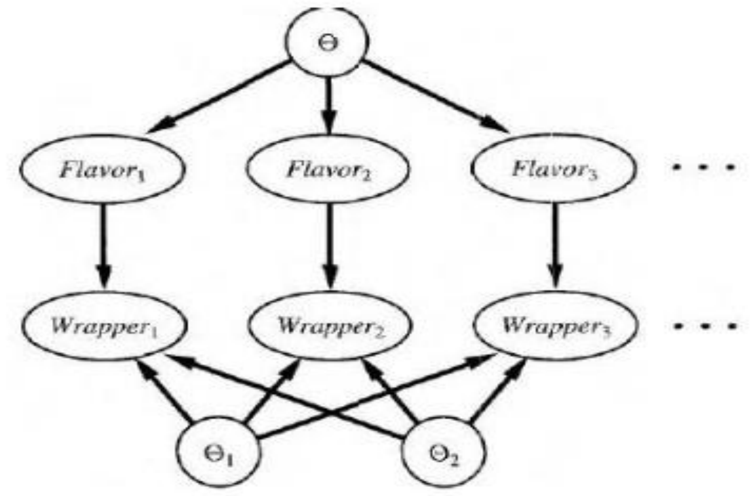


Figure: A Bayesian network that corresponds lo a Bayesian learning process. Posterior distributions for the parameter variables 0, el, and *e2* can be inferred from their prior distributions and the evidence in the *Flavor,* and *Wrapper* variables.

There are two alternative methods for deciding when a good structure has been found. The first is to test whether the conditional independence assertions implicit in the structure are actually satisfied in the data

15.(a) (i) Construct semantic net representation for the following:

- **Pomepeian(Marcus),Blacksmith(Marcus)**

- **Mary gave the green flowered vase to her favorite cousin.**

(ii) Construct partitioned semantic net representations for the following:

- **Every batter hit a ball**
- **All the batters like pitcher.**
  (2x4=8)

Or

(b) Illustrate the learning from examples by induction with suitable examples.

An **example** is a pair *(x, f* (z)), where x is the input and *f(x)* is the output of the function applied to *x*. The task of **pure inductive inference** (or **induction)** is this: Given a collection of examples of *f*, return a function *h* that approximates f. The function h is called a **hypothesis.**

For nondeterministic functions, there is an inevitable tradeoff between the complexity of the hypothesis and the degree of jit to the data. There is a tradeoff between the expressiveness of a hypothesis space and the complexity of finding simple, consistent hypotheses within that space.