Computer Science 161 Notes

Textbook notes

—

Chapter 1 (What is AI & History)

AI encompasses a huge variety of subfields, ranging from general learning and perception to specific tasks like chess. **AI is relevant to any intellectual task.** This makes AI an universal field of study.

The definitions of AI can be split up based on whether they focus on how systems should think (thought processes and reasoning) or behavior. They can also be split in terms of whether they are measured with the respect to human performance or ideal performance (rationality)

To pass a turing test, a computer would need to have NLP, knowledge representation (to store what it knows or hears), automated reasoning (using stored information to answer questions), machine learning (to adapt to new circumstances), computer vision, and robotics to manipulate objects.

Hebbian Learning: any computable function could be computed by some network of connected neurons and that all gates (and, or, not, etc) can be implemented by such a net style structure.

Genetic Algorithms: making an appropriate series of small mutations to machine code once can generate a program with good performance for any particular task.

Rational Agent: A computer that acts to achieve the best outcome or, when there is uncertainty, the best expected outcome.

Chapter 2 (Defining Agents)

An **agent** is anything that can be perceived viewing its environment through sensors and acting upon that environment through actuators. The term percept refers to the agent's perceptual inputs at any given instant. The agent function is what maps any given percept sequence to an action. The agent program is a concrete implementation of the agent function, while the agent function is an abstract mathematical description.

Chapter 3 (Search Agents)

Reflex agents base their actions on a direct mapping from states to actions. This does not work well in cases where the environment is very large and therefore it would have to store too much information

Problem-Solving Agents avoid this problem by considering the states of the world as wholes with no internal structure visible to the algorithms

A problem is defined formally by five components:

- **Initial State:** the start state of the agent
- **Actions**: a set of valid actions the agent can take (Actions[s] return the set of actions that can be executed in s)
- **Transition model:** returns the new state s that results from taking an action. [Successor is a term used to refer to any state reachable from a given state by a single action]
    - State Space: is implicitly defined by the above 4 it is all of the possible states that can explored from the initial state by taking any valid action repeatedly
    - Path: a sequence of states connected by actions
- **Goal Test:** determine whether a given state is a goal state
- **Path Cost:** assigns a numeric cost to each path (We assume that step costs are nonnegative)

Measuring Correctness of an Algorithm:

- Completeness: is the algorithm guaranteed to find a solution if there is one
- Optimality: does it find the optimal solution

Chapter 4 (Local Search Algorithms):

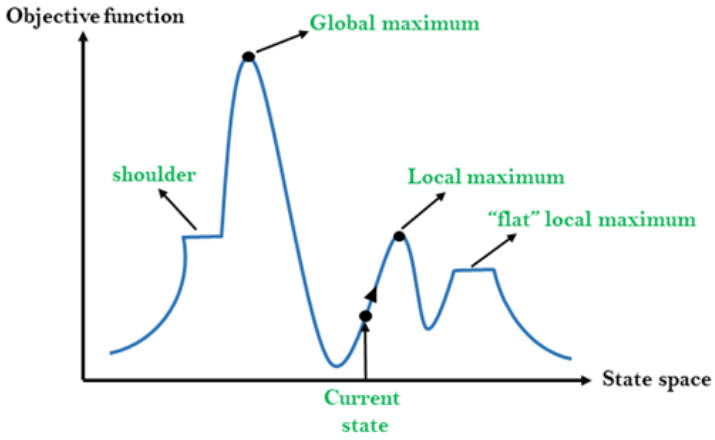
In the case that the path to a goal does not matter we can consider a different class of algorithms called local search algorithms.

Such Algorithms have two key advantages:

(1) They use very little memory – (usually constant)
(2) They can find reasonable solutions in large or infinite state spaces in which algorithms like BFS and Uniform Cost Search fail

Hill Climbing (Steepest-Ascent):

The is a greedy approach that intends to find the local maximum of the state space. It will keep moving in the increasing direction, until none of the neighboring states have a higher value.



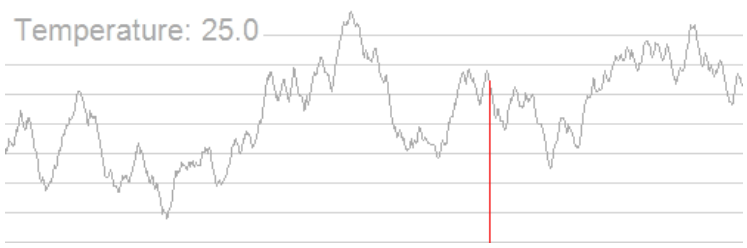
(1) It can get stuck at local maximums
(2) It can get lost on a "flat" area

To solve the above issues we can add random sideways movements, or to make it more even more accurate we can conduct it with a random start state multiple times, restarting the algorithm multiple times at different initial states slowly moves the probability of success closer and closer to 1

Simulated Annealing:

We allow for some bad moves, and slowly reduce the frequency with which we allow those bad moves. We gradually decrease the size and frequency of the bad moves allowed. We call this the temperature T and with the right temperature T the probability of finding a global optimum approaches 1.

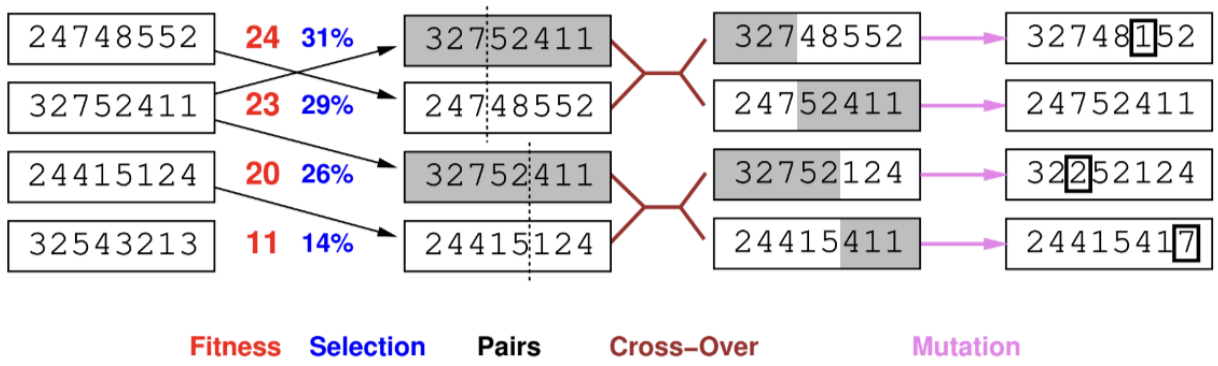
(1) If a move improves a situation it is always accepted, if it makes the situation worse it is accepted with some probability less than 1.
(2) The probability that a move is accepted is decreases exponentially based on how bad it is ΔE represents that difference



Local Beam Search:

Instead of just keeping one state we will keep track of k states, and then continually choose the k best states to continue our search. [This is not the same as running k-searches in parallel]. One issue is that all k states will end up on the same local hill → use stochastic beam search to take k-random states so that they start in different areas of the state space.

Genetic Algorithms:



GA starts with k randomly generated states (population), we run each state through a fitness function to get the probability that it will be able to reproduce. Then use these probabilities to select the different states. We then randomly pair two states, and these states will combine using a crossover point that is random. Some mutations can happen with a low probability that we define. [GA's require that states are represented as strings]

Continuous to Discrete Spaces:

All of the above algorithms besides Simulated Annealing and Hill Climbing cannot handle continuous spaces. To do this we can discretize a state space, by only changing one factor at a time.

Gradients:

∇f indicates the magnitude and direction of the steepest slope. Hence we can use the gradient to compute where a maximum is located. $x \leftarrow x + \alpha \nabla f$

In very certain cases we can compute $\nabla f = 0$ exactly, using the Newton-Raphson method where it can be easily calculated

Chapter 6 (Constraint Satisfaction):

A constraint satisfaction problem consists of three components X, D, and C:

- X is a set of Variables $\{X_1...,X_i\}$
- D is the set of domains for each variable $\{D_1...D_i\}$
- C is a set of constraints that specify allowable combinations of values

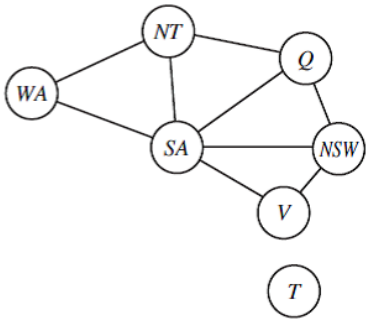
Note: it is often helpful to visualize CSP problems as graphs

Metrics of Solutions:

- Complete Assignment, each variable is assigned a value
- Partial Assignment, only some variables are assigned a value
- Consistent Solution, one that doesn't violate any constraints

Ex: Map Coloring:

- X = {WA, NA, SA, Q, NS, T}
- D_i = {red, green, blue}
- C: Adjacent regions must have different colors



CSPs are more efficient, since once we see that a partial assignment is not correct we can discard all future iterations of that process, whereas with search we only determine whether the current state is a solution

CSP <u>domains must be finite</u> if there is an infinite domain, we cannot use CSP. For a domain size d ⇒ we get $O(d^n)$ complete assignments. We also need a constraint language for CSPs. Note the only exception to this is if our domain is infinite but our constraints are linear equalities or inequalities we can solve this.

Indicating, which constraints are more desirable will lead to form of a constraint optimization problem (COP)

<u>Types of constraints:</u>

- Unary: constraints require only one variable
- Binary: constraints require pair of variables
- Higher order constraints: require 3 more variables

Node Consistency: a single variable is node consistent if it satisfies all of it's unary constraints

Arc Consistent: if very in its domain satisfies the variable's binary constraints. For a variable X_i and X_j they are arc consistent if for every value in D_i there is some value in D_j such that it satisfies the constraint

```
function AC-3( csp) returns the CSP, possibly with reduced domains
inputs: csp, a binary CSP with variables {X_1, X_2, ..., X_n}
local variables: queue, a queue of arcs, initially all the arcs in csp

while queue is not empty do
(X_i, X_j) ← REMOVE-FIRST(queue)
if REMOVE-INCONSISTENT-VALUES(X_i, X_j) then
for each X_k in NEIGHBORS[X_i] do
add (X_k, X_i) to queue
return true
```

```
function REMOVE-INCONSISTENT-VALUES( X_i, X_j) returns true iff succeeds
removed ← false
for each x in DOMAIN[X_i] do
if no value y in DOMAIN[X_j] allows (x,y) to satisfy the constraint X_i ↔ X_j
then delete x from DOMAIN[X_i];  removed ← true
return removed
```

The above code makes a CSP problem Arc-consistent, the time complexity of the algorithm is $O(n^2d^3) \Rightarrow O(cd)$

SEARCH SPACE FORMULATION FOR CSP:

- Initial State = []
- Successor Function: assign a value to an unassigned variable that does not conflict with the current assignment ⇒ fail if there are no legal assignments
- Goal Test: if the current assignment is complete (we already have consistency since we choose values that do not conflict!)

Every Solution appears at depth n with n variables so we use depth first search. However our branching factor b = (n-L)(d) so we get $n!*d^n$ leaves

Assignment Properties: variable assignments are commutative i.e it does not matter in which order we will assign the different variables.

- WA = {Green} then SA = {Red} is the same as SA = {Red} then WA = {Green}

Backtracking Search:

It is DFS that chooses values for one variable at a time and then backtracks when a variable has no legal values left to assign. ⇒ Since CSPs are standardized there is no need to supply Backtracking-search with a domain specific initial state, action function, transition model, or goal test.

- CSPs can be solved efficiently, *without* such domain-specific knowledge compared to the uninformed search strategies.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
if assignment is complete then return assignment
var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
if value is consistent with assignment given CONSTRAINTS[csp] then
add {var = value} to assignment
result ← RECURSIVE-BACKTRACKING(assignment, csp)
if result ≠ failure then return result
remove {var = value} from assignment
return failure
```

Selecting the Next Variable (Select-Unassigned-Variable) **Minimum Remaining Values:** we choose the value which is the "most constrained variable" since this increases the likelihood that this search tree will fail, thereby pruning our search tree earlier.

- What happens when all regions have the same number of constraints ⇒ we can use the degree heuristic: choose the variable that is involved in the largest number of constraints on other unassigned variables.
- Other Option: Least Constrained choose the least constrained value to leave maximum availability for future assignments

Forward Tracking:

In-order to detect failure early we can keep track of the remaining possible values of all variables and once any variable doesn't have any possibilities we can then terminate.

This does not account for the fact that two other variables with constraints on each other may only have the same value. To account and find this we can use an algorithm called Maintaining Arc Consistency (MAC)

Breaking Down CSP Problems:

If we assume that there are multiple csp problems with c variables from the total of the n variables and each problem takes at most d work to solve. The complexity of solving the CSP would be: $(n/c)*d^c$.

Tree Structured CSPs:

A constraint graph is a tree when any two variables are connected at most by only one path

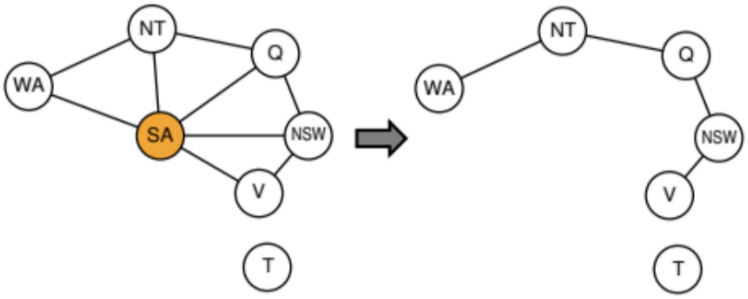
In such a case the time complexity to solve it reduces to: $O(nd^2)$ ⇒ Remember that a tree will have at most n-1 arcs.

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
  inputs: *csp*, a CSP with components X, D, C

  $n \leftarrow$ number of variables in X
  assignment $\leftarrow$ an empty assignment
  root $\leftarrow$ any variable in X
  $X \leftarrow$ TOPOLOGICALSORT(X, *root*)
  for $j = n$ **down to** 2 **do**
    MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
    if it cannot be made consistent **then return** *failure*
  for $i = 1$ **to** n **do**
    assignment$[X_i] \leftarrow$ any consistent value from D_i
    if there is no consistent value **then return** *failure*
  return *assignment*

What about cases where there are cycles?

The removal of SA forms a new tree-CSP structure. In this case SA is part of the cutset. (it is important to note that a cutset can contain more than one variable), let us call this set S




```
# The Algorithm:
for each possible assignment in S:
      Remove inconsistent values in domains that are inconsistent with S
      if the remaining CSP has solution return that
```

Time Complexity of Cutset Methods:

Given a cutset with a size c, the time complexity becomes: $O(d^c (n-c)d^2)$. Note that for small values of c, this works well, however c can become as large as n-2, and finding the smallest cycle becomes NP-Hard

Iterative Algorithms for CSPs:

Hill Climbing and Simulated Annealing often work well in cases where we are looking for a complete solution.

To apply CSPs:

We allow states with unsatisfied constraints, and the operators will reassign variable values

Selection of Variable: Chosen at random

Value Selection: Min-conflicts heuristic

Choose a value that violates the smallest number of constraints, for example we would climb with h(n) where that is the total number of constraints.

Chapter 5 (Game Playing and Adversarial Search):

Types of Games:

Perfect information means that we have all of the available information

Imperfect information means that we have only some information but not all of it.

	deterministic	chance
perfect information	chess, checkers, go, othello	backgammon monopoly
imperfect information	battleships, blind tictactoe	bridge, poker, scrabble nuclear war

Formalizing a Game as a Search Problem:

S_0 (The Initial State): specifies how the game is set up at the start

Player(s): Defines which player has the move in a state

Action(s): Returns the set of legal moves in a state

Result(s, a): The transition model, which defines the result of a move

Terminal-Test(s): A terminal test is true when the game is over and false otherwise. (States when the game ends are called terminal states)

Utility(s p): defines the numeric value for a game that ends in a terminal state s, for a player p. (In chess this would be 0, 1, ½)

> Zero-sum game is where the total payoff for allplayers is the same for every instance of the game (in chess it is always 1)

Making optimal Decisions:

To make the optimal decision we would choose the highest achievable payoff, we assume that the other player (MIN) will always make the optimal decision

MIN-MAX:

Player #1 is referred to as MAX, and Player #2 is referred to as MIN

Player #1 (MAX) will always choose the maximum payoff value, while Player #2 (MIN) will always choose the minimum payoff value. (Best to think of it as one player trying to gain progress, while other player attempts to negate that progress)

$$\textsc{Minimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_{a \in Actions(s)} \textsc{Minimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \end{cases}$$

function MINIMAX-DECISION(*state*) **returns** *an action*
inputs: *state*, current state in game

return the *a* in ACTIONS(*state*) maximizing MIN-VALUE(RESULT(*a*, *state*))

---

function MAX-VALUE(*state*) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow -\infty$
for *a*, *s* in SUCCESSORS(*state*) **do** $v \leftarrow$ MAX(v, MIN-VALUE(*s*))
return *v*

---

function MIN-VALUE(*state*) **returns** *a utility value*
if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
$v \leftarrow \infty$
for *a*, *s* in SUCCESSORS(*state*) **do** $v \leftarrow$ MIN(v, MAX-VALUE(*s*))
return *v*

Complete: Yes, as long as the tree is finite, if it is infinite it is not complete (but a finite strategy can exist in an infinite tree)

Optimal: yes, given that the opponent plays optimally

Time Complexity: $O(b^m)$

Space Complexity: O(bm) [it basically does a dfs search]

Alpha - Beta Pruning:

α = the value of the best (highest-value) choice we have found so far at any choice point along the path for MAX.

β = the value of the best (lowest-value) choice we have found so far at any choice point along the path for MIN. n

At a MAX location if node value is $\geq \beta$ we can prune the children

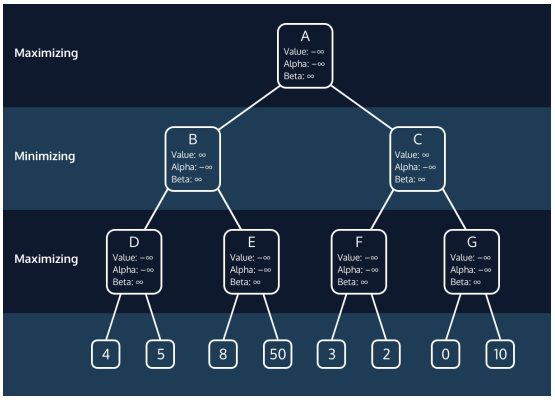
At a MIN location if node value is $\leq \alpha$ we can prune the children

```
function ALPHA-BETA-DECISION(state) returns an action
return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

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



Pruning does not affect the final result → we will still get the optimal solution.

With a good ordering of when we prune, our time complexity is: $O(b^{m/2})$

The effectiveness of alpha-beta pruning is highly dependent on the order in which the states are examined.

However, we still explore a large set of the state space even with pruning, to make it more time efficient, we can do the following (this does not provide an optimal solution)

- Cutoff Test instead of Terminal-Test (we end the search at a certain depth)
- EVAL Test instead of Utility at our stop location from cutoff

Evaluation Function:

Returns an estimate of the expected utility of the game from a given function (something similar to a heuristic function)

The evaluation function should order the terminal states in the same way as the utility function

It is also important for perfect games, the exact values do not matter as long as their relation changes...aka we can apply a monotonic transformation (is a way of transforming one set of numbers into another set of numbers in a way that the order of the numbers is preserved)

Stochastic Games:

Games that require minor unpredictability by introducing randomness but may still require some skill. → This means that we add probability branches to our game tree

Suppose we have a game of dice rolls the time complexity would be ⇒ $O(b^m n^m)$ where n is the number of distinct rolls.

Exact Values in this case do matter → we can only apply a linear transformation

```
if state is a MAX node then
    return the highest EXPECTIMINIMAX-VALUE of
SUCCESSORS(state)
if state is a MIN node then
    return the lowest EXPECTIMINIMAX-VALUE of
SUCCESSORS(state)
if state is a chance node then
    return average of EXPECTIMINIMAX-VALUE of
SUCCESSORS(state)
```

Alpha-Beta Pruning can be applied to in this case, however, some ingenuity is required.

Monte-Carlo Simulation → we start a known search algorithm and have the algorithm play thousands of games against itself, using random dice rolls. In the case of backgammon the resulting win percentage has been shown to be a good approximation of the value pf the position.

Chapter 7 (Logic Agents and Propositional Logic):

Knowledge-Based Agents: reasoning that operates on internal representations of knowledge.

- Knowledge Base: is a set of sentences (each will represent some assertion about the world).
- Tell: add sentences to the knowledge base
- Ask: query what is known
- Inference: deriving new sentences based on old knowledge or current sentences
    - Both TELL and ASK may use inference to get the correct result

```
function KB-AGENT( percept) returns an action
static: KB, a knowledge base
        t, a counter, initially 0, indicating time

TELL(KB, MAKE-PERCEPT-SENTENCE( percept, t))
action ← ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t + 1
return action
```

A knowledge agent must be able to do the following: Represent states & actions, incorporate new precepts, update internal representations of the world, deduce hidden properties of the world, deduce appropriate actions.

- Declarative Approach: agent designed can TELL sentences one by one to the agent
- Procedural Approach: encodes desired behaviors directly as program code

Fundamental Property of logical reasoning: in each case for which the agent draws a conclusion from the available information, that conclusion is guaranteed to be correct if the available information is correct.

Logic: are the formal languages for representing information

Syntax: defines the sentences in the language

Semantics: define the meaning of the sentences

If a sentence a is true in a model m, we say that m satisfies a (or m is a model of a) ⇒ The notation M(a) is used to represent the set of all models of a

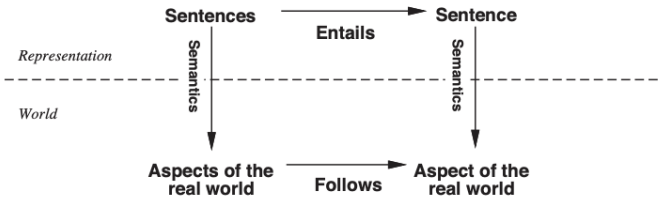
Entailment: a sentence follows logically from another sentence (KB $\models \alpha$)

- A knowledge base KB entails α if an only if α is true in all worlds where KB is true
    - Eg a knowledge base where “The Giants Won” and “The Reds Won” entails that the giants or the reds won.
- $\alpha \models \beta$ if and only if M(α) ⊆ M(β) note that α is a stronger assertion than β: it rules out more possible worlds
- The KB is false in models that contradict what the agent knows, and the are true when the models do not contradict what the agent knows

Inference:

- Entailment is like the needle being in the haystack, inference is finding the needle in the haystack
- KB $\vdash_i \alpha$ ⇒ a sentence α can be derived from from the KB by an inference algorithm i
  - Soundness: an inference algorithm that only derives entailed sentences
    - KB $\vdash_i \alpha$, then KB $\models \alpha$
  - Completeness: when an inference algorithm can derive any sentence that is entailed
    - KB $\models \alpha$, then KB $\vdash_i \alpha$

If the KB is true in the real world, then any sentence α derived from the KB by a sound inference procedure is also true in the real world.



Propositional Logic:

Atomic Sentences: consist of a single proposition symbol (each symbol can be either true or false)

Complex Sentences: are constructed using parentheses and logical connectives

¬ (not): Is the negation of a symbol (makes something a positive literal or negative literal)

∧ (and): conjunction made up of conjuncts

∨ (or): disjunction made up of disjuncts

⇒ Called an implications (can be thought as rules or if-then statements)

⇔ (if and only if) the sentence is biconditional (≡, if p ⇔ q, then if p then q and if q then p)

$$
\begin{aligned}
Sentence &\rightarrow AtomicSentence \mid ComplexSentence \\
AtomicSentence &\rightarrow True \mid False \mid P \mid Q \mid R \mid \ldots \\
ComplexSentence &\rightarrow (\ Sentence\ ) \mid [\ Sentence\ ] \\
&\mid \neg\ Sentence \\
&\mid Sentence \wedge Sentence \\
&\mid Sentence \vee Sentence \\
&\mid Sentence \Rightarrow Sentence \\
&\mid Sentence \Leftrightarrow Sentence \\
\text{OPERATOR PRECEDENCE} &: \neg, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

The semantics are just that each proposition symbol has a truth value (true or false)

For the operators, below are the truth tables, and they are with respect to a model m (so P and Q would be true or false within a model m)

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

Knowledge Bases in Propositional Logic: Recall that knowledge bases is a collection of sentences and we can represent this collection of sentences in a single complex sentence by conjunction of each other individual sentences together. KB = Sentence ∧ Sentence ∧ Sentence ∧ Sentence…

Inference Algorithms: given that we have a set of rules and variables how can we determine if the knowledge base entails a sentence:

- A DFS iteration through all possible values of the truth table to determine if there is entailment ⇒ this is sound and complete
- Enumerates through different models (an assignment to some of the symbols) and stops when any model is true
- $O(2^n)$ where n is the number of symbols (this is NP-complete)

Logical Equivalence: two sentences α and β are equivalent iff $\alpha \vDash \beta$ and $\beta \vDash \alpha$, this means that they are true in the same set of models

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

Validity: a sentence is valid if it is true in *all* models. (Called tautologies→they are necessarily true)

Eg: A ∨ ⌐A

Satisfiability: a sentence is satisfiable if it is true in *some model*

Eg: A ∨ B

Unsatisfiability: a sentence is unsatisfiable if it is true in no model

Eg: A ∧ ⌐A

Deduction Theorem:

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \Rightarrow \beta)$ is *valid*.

For any sentences α and β, $\alpha \models \beta$ if and only if the sentence $(\alpha \wedge \neg\beta)$ is *unsatisfiable*.

Inference rules: can be derived to a proof – a chain of conclusions that lead to the desired goal. (Any logical equivalence statement can be used as an inference rules) these rules are sound.

Modus Ponens: $\frac{\alpha \rightarrow \beta,\ \alpha}{\beta}$, this means that if $\alpha \Rightarrow \beta$ and α is given, then the sentence β can be inferred

And-Elimination: $\frac{\alpha \wedge \beta}{\alpha}$, this means that if we have $\alpha \wedge \beta$ then the sentence α can be inferred

Searching for proofs is an alternative to enumerating models

Monotonicity: that the set of entailed sentences can only increase as information is added to the knowledge base.

Conjunctive Normal Form: every sentence of propositional logic is logically equivalent to a conjunction of clauses. CNF is a conjunction of clauses, where a clause is a disjunction of literals, and a literal is a propositional symbol and its negation. **conjunction of disjunctions of literals** (clauses)

$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$

1. Eliminate $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$.

$$(B_{1,1} \Rightarrow (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \Rightarrow B_{1,1})$$

2. Eliminate $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Move $\neg$ inwards using de Morgan's rules:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Apply distributivity law ($\vee$ over $\wedge$) and flatten:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

Resolution: if we have a set of literals that are being OR with each other and another smaller set of literals m that are being OR together we can remove it from our rule. Here m and l_i should be complementary literals (one is the negation of the other)

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

Resolution Algorithm: resolution algorithms work based on proof by contradiction, using the deduction algorithm of proving that $(\alpha \wedge \neg\beta)$ is *unsatisfiable*. Firstly (KB $\wedge \neg\alpha$) is converted to CNF form. Then the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause.

- If there are no new clauses that can be added, then the KB does not entail α
- Two clauses resolve to yield an empty clause, then the KB does entail α

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

Definite Clause: a disjunction of literals of which at most is positive

Horn Clause: a disjunction of literals of which *at most one* is positive (all definite clauses are horn clauses, Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause).

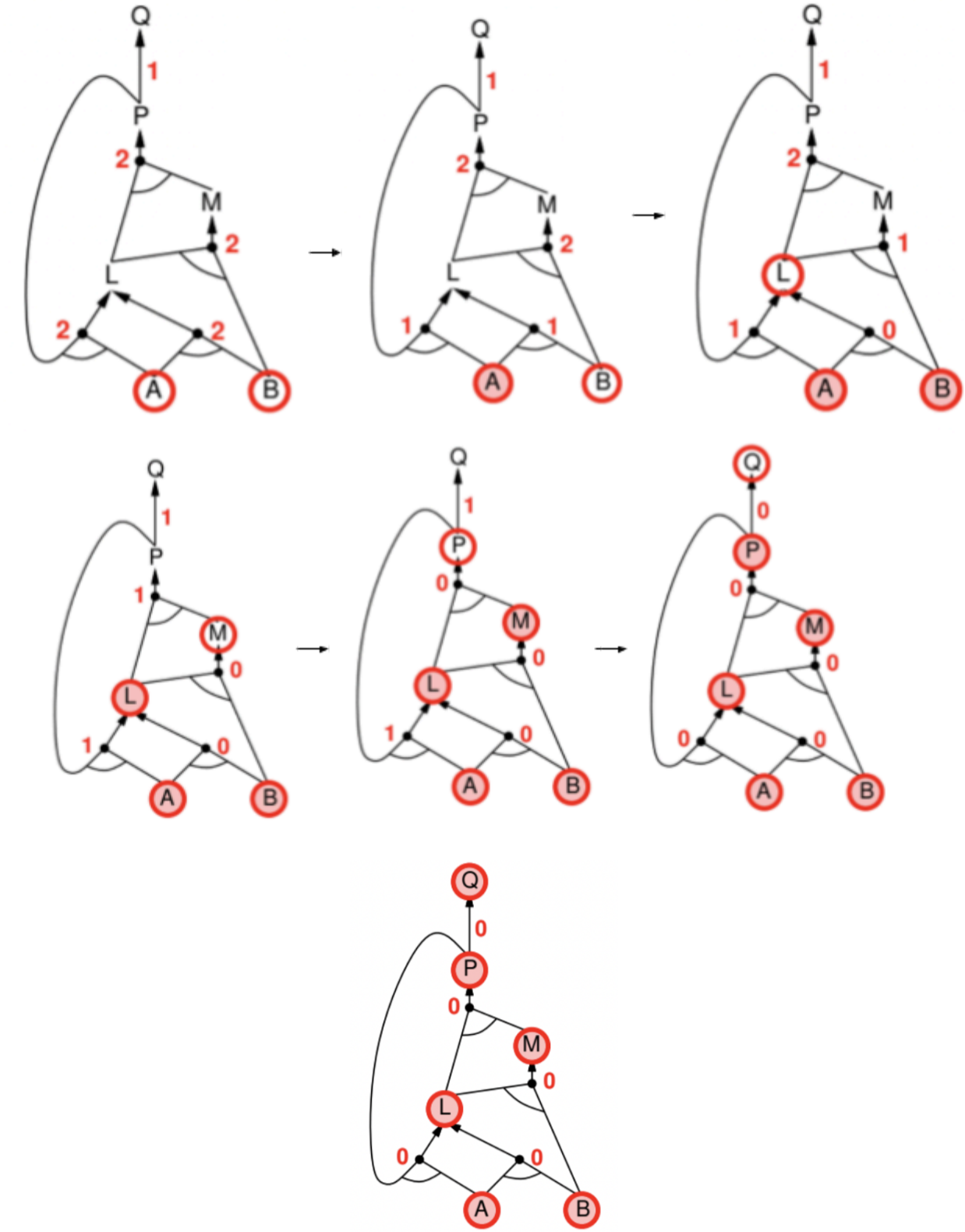
- If we are given (Conjunction of Symbols) ⇒ Literal, then it is also a Horn clause, suppose that some of the symbols are negated, then we would need to expand this into a disjunctive form to truly determine if it is a horn clause.

Forward Chaining: only works with Horn Clauses, but it works in linear time with respect to the knowledge base. However, forward chaining is data-driven which means that it may end up doing work that is irrelevant to the goal.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p ← POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

Forward Chaining Example:



Chapter 8 (First-Order Logic):

First-order logic contains:

- Objects (nouns and noun phrases) we also need to ensure that that domain of objects is non-empty, there cannot be zero objects in a world
- Relations among objects (verbs)
- Functions (which are a subset of relations)

Symbols stand for objects (constant symbols → objects, predicate symbols → relations, function symbols → functions)

First Order Logic consists of a set of objects and an interpretation that amps constant symbols to objects, predicate symbols to relations on those objects, and function symbols to functions on those objects

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

Term: A logical expression that refers to an object (hence constant symbols are terms), we can even have a function as a Term.

Atomic Sentence (atom): is formed from a predicate symbol *optionally* followed by a parenthesized list of terms:

- Atomic sentences are true in a given model if the relation referred to by the predicate symbols holds among the objects referred to by the arguments

Complex Sentences: we can use logical connectives between different atomic sentences to construct complex sentences.

Quantifiers: give the ability to express properties about entire collections of objects, instead of referring to objects solely based on their name:

Universal Quantification(∀): Means "For all", the sentence ∀x P, where P is any logical expression says that P is true for every object x.

- Should generally only be used with ⇒ (this allows us to make general rules with universal quantifiers)
- We often end up writing a rule just for those objects for whom the premise is true, and say nothing at all about the individuals for whom the premise is false

Existential Quantification(∃): "There exists an x such that". The sentence ∃x P states P is true for at least one object x.

- The natural connective to use with ∃ is ∧

Nesting Quantifiers:

∀x ∀y is the same as ∀y ∀x

∃x ∃y is the same as ∃y ∃x

∃x ∀y is not the same as ∀y ∃x (Take ∀x ∃y Loves(x,y) which means Everybody loves somebody, versus ∃y ∀x Loves(x,y) which means There is someone who is loved by everyone).

It is important to note that we can convert between ∀ and ∃ using negation, as an example

∀x ¬Likes is equivalent to ¬∃x Likes

∀x Likes is equivalent to ¬∃x ¬Likes

De Morgan's rules also apply to quantifiers:

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

Quantifiers and Kinship:

Brothers are Siblings:

$\forall$ x,y Brother(x,y) $\Rightarrow$ Sibling(x,y)

"Sibling" is symmetric:

$\forall$ x y Sibling(x,y) $\Leftrightarrow$ Sibling(y,x)

"First cousin" is child of parent's sibling:

$\forall$ x,y First_Cousing(x,y) $\Leftrightarrow$ $\exists$ p,ps Parent(p,x)$\wedge$Sibling(p,ps)$\wedge$Parent(ps,y)

Chapter 9 (First-Order Logic Inferences):

Universal Instantiation (UI): We can infer any sentence obtained by substituting a ground term (a term without variables) for a variable.

$$\frac{\forall v \ \alpha}{\text{SUBST}(\{v/g\}, \alpha)}$$

As an example take the following ∀x King(x)∧Greedy(x) ⇒ Evil(x).
This states that all Greedy Kings are evil. We can infer that if John is a king and greedy, then John is evil. Hence we can use the substitution {x/John}

Existential Instantiations: for any sentence α, variable v, and constant symbol k that does not appear elsewhere in the Knowledge base.

$$\frac{\exists v \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

Eg: ∃x Crown(c)∧OnHead(x, John) we can infer the sentence

Crown(C_1)∧OnHead(C_1, John)

The constant symbol K should not belong to another object, and this new constant symbol is called the **Skolem Constant**.

Universal Instantiation can be applied many times to produce different consequences, Existential Instantiation can be applied once, and then the existentially quantified sentence can be discarded.

First-order Inference to Propositional inference: we can replace an existentially quantified sentence can be replaced by one instantiation, and a universally quantified sentence can be replaced by the set of *all possible* instantiations:

$\forall x \ \ King(x) \wedge Greedy(x) \ \Rightarrow \ Evil(x)$
$King(John)$
$Greedy(John)$
$Brother(Richard, John)$.

→

$King(John) \wedge Greedy(John) \ \Rightarrow \ Evil(John)$
$King(Richard) \wedge Greedy(Richard) \ \Rightarrow \ Evil(Richard)$
$King(John) \ Greedy(John)$
$Brother(Richard, John)$

With a method like this is called propositionalization and the new KB is propositionalized.

Entailment in first-order logic is ***semidecidable*** which means that algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non-entailed sentence.

If a sentence α is entailed by a KB, then it is entailed by a finite subset of the KB.

- For $n = 0$ to ∞ do
  - create a propositional KB by instantiating with depth-n terms
  - see if α is entailed by this KB
- Works if α is entailed, loops if α is not entailed!

Unficiation: finding substitutions that make different logical expressions look identical.

Unification may fail when the same variable takes on two different values, since a variable cannot take two different values at the same.

Unification may fail during an occur check, take the following: S(x) *can't* unify with S(S(x))

Standardizing Apart: eliminates overlapping variables by renaming them to avoid name clashes.

α	β	θ
$Knows(John, x)$	$Knows(John, Jane)$	$\{x/Jane\}$
$Knows(John, x)$	$Knows(y, OJ)$	$\{x/OJ, y/John\}$
$Knows(John, x)$	$Knows(y, Mom(y))$	$\{y/John, x/Mom(John)\}$
$Knows(John, x)$	$Knows(x, OJ)$	$fail$
$Knows(John, x)$	$Knows(x_{17}, OJ)$	$\{x/OJ, x_{17}/John\}$

Most General Unifier: finds the most generalized unifier with the least number of restrictions if a unifier exists. For example take UNIFY(Knows(John,x), Knows(y,z)) the MGU would be {y/John, x/z}, however a unification algorithm may return: {y/John, x/John, z/John} however, the first one is more general since it places the least number of constraints on the possible values of x and z.

function UNIFY(x, y, θ) **returns** a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound expression
    y, a variable, constant, list, or compound expression
    θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure **then return** failure
  else if $x = y$ **then return** θ
  else if VARIABLE?(x) **then return** UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) **then return** UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) **and** COMPOUND?(y) **then**
    return UNIFY(x.ARGS, y.ARGS, UNIFY(x.OP, y.OP, θ))
  else if LIST?(x) **and** LIST?(y) **then**
    return UNIFY(x.REST, y.REST, UNIFY(x.FIRST, y.FIRST, θ))
  else return failure

function UNIFY-VAR(var, x, θ) **returns** a substitution

  if $\{var/val\} \in \theta$ **then return** UNIFY(val, x, θ)
  else if $\{x/val\} \in \theta$ **then return** UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) **then return** failure
  else return add $\{var/x\}$ to θ

Figure 9.1 The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution θ that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression such as $F(A, B)$, the OP field picks out the function symbol F and the ARGS field picks out the argument list (A, B).

Generalized Modus Pones (GMP):

$$\frac{p_1',\ p_2',\ \ldots,\ p_n',\ (p_1 \wedge p_2 \wedge \ldots \wedge p_n \Rightarrow q)}{\textsc{Subst}(\theta, q)}.$$

The GMP is a sound inference rule (note we never say that it is a complete inference rule!)

Definite Clauses: an atomic sentence, or an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal.

Conversion to CNF:

1. Eliminate Biconditionals and implications
2. Move $\neg$ inwards
3. Standardize each variable, each quantifier should use a different variable
4. Skolemize (a more general form of the existential unification)
5. Drop Universal Quantifiers
6. Distribute $\wedge$ over $\vee$

Resolution: we insert $\neg\alpha$ into our KB and then resolve.

Chapter 13 Quantifying Uncertainty:

Adding probabilities to our knowledge base account for the following factors:

- Laziness: failure to enumerate exceptions, qualifications, etc
- Ignorance: lack of facts, initial conditions or insufficient knowledge

In essence probability provides a way to summarize the uncertainty that arises from our own laziness and ignorance. Decision Theory: Utility Theory + Probability (where utility theory decides the utility).

Subjective (Baysian) Probability: probabilities relate to one's own state of knowledge and these probabilities can change with new evidence.

Probability Basics: We start with a sample space $\mathbf{\Omega}$ and let $\omega \in \mathbf{\Omega}$, where ω is a sample point/atomic event. Then we say that for all ω: $0 \leq P(\omega) \leq 1$, and across the entire sample space: $P(A) = \sum_{\omega \in \mathbf{\Omega}} P(\omega)$

- A Random variable, is a function from some sample points to some range. Let X be the random variable then: $P(X = x_i) = \sum_{\{\omega: X(\omega) = x_i\}} P(\omega)$.
- Events are called propositions
  - Propositional or Boolean random variables (eg: Cavity = True)
  - Discrete Random Variables can be finite or infinite (eg: Weather =⟨sunny, rainy, cloudy, snow⟩in the case we get Weather = sunny this would be a proposition).
    - Values must be exhaustive: at least one of the events should occur
    - Values must be mutually exclusive: two events cannot happen at the simultaneously
  - Continuous Random Variables can be unbounded or bounded (eg: Temp = 21.6 given that Temp < 22.0)

Unconditional Probability: refers to a probability given that we have no other information available.

Conditional Probability: refers to a probability given that we have other prior information or events that have occurred. This is written with the | (eg. P(cavity | toothache) means the probability that we have a cavity given that toothache is true and we have no further information).

- $P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$ this also implies that $P(A \wedge B) = P(A \mid B)\, P(B) = P(B \mid A)P(A)$

Joint Probability Distributions: denotes all possible combinations of values and is denoted by a **P**. (eg. **P**(Weather, Cavity) = 4*2 = 8 possible combinations).

$$P(W = sunny \wedge C = true) = P(W = sunny|C = true)\, P(C = true)$$
$$P(W = rain \wedge C = true) = P(W = rain|C = true)\, P(C = true)$$
$$P(W = cloudy \wedge C = true) = P(W = cloudy|C = true)\, P(C = true)$$
$$P(W = snow \wedge C = true) = P(W = snow|C = true)\, P(C = true)$$
$$P(W = sunny \wedge C = false) = P(W = sunny|C = false)\, P(C = false)$$
$$P(W = rain \wedge C = false) = P(W = rain|C = false)\, P(C = false)$$
$$P(W = cloudy \wedge C = false) = P(W = cloudy|C = false)\, P(C = false)$$
$$P(W = snow \wedge C = false) = P(W = snow|C = false)\, P(C = false)\ .$$

Well Known Probability Axioms:

- P(¬A) = 1 - P(A)
- P(A v B) = P(A) + P(B) - P(A∧B)
- Pr(B) = Pr(B|A) Pr(A) + Pr(B|¬A) Pr(¬A)

If Agent 1 expresses at a set of degrees of belief that violate the axioms of probability theory then there is a combination of bets by Agent 2 that guarantees that Agent 1 will lose money every time.

Inference of Full Joint Distributions:

	$toothache$		$\neg toothache$	
	$catch$	$\neg catch$	$catch$	$\neg catch$
$cavity$	0.108	0.012	0.072	0.008
$\neg cavity$	0.016	0.064	0.144	0.576

- Marginal Probability (ignoring the influence of probability of other variables), eg: P(cavity) = 0.108 + 0.012 + 0.072 + 0.008 = 0.2
- Extending this to a set of variables we get: $P(Y) = \sum_{\{z \in \mathbb{Z}\}} P(Y, \mathbb{Z})$
- This can also be rewritten as: $P(Y) = \sum_{\{z \in \mathbb{Z}\}} P(Y|\mathbb{Z})\, P(\mathbb{Z})$ it's important to note that here we sum over all possible combinations of the set of $\mathbb{Z}$.

Observe that in the equation $P(A \mid B) = \frac{P(A \wedge B)}{P(B)}$ P(B) in the denominator could be treated as some constant α that we would not need to calculate. We get the following:

$P(X \mid e) = \alpha P(X \mid e) = \alpha \sum_{\{Y\}} P(X, e, Y)$ the time complexity for this is $O(d^n)$ for both space and time, where d is the largest domain size (for all boolean variables it would be 2^n)

Independence:

P(α | β) = P(α) or

P(β | α) = P(β) or

P($\alpha \wedge \beta$) = P(α)P(β) or

P(X | Y) = **P**(X) or

P(Y | X) = **P**(Y) or

P(X,Y) = **P**(X)**P**(Y)

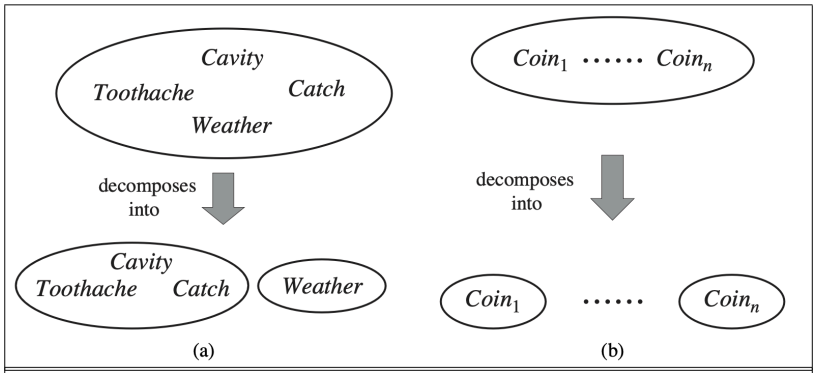


Figure 13.4 Two examples of factoring a large joint distribution into smaller distributions, using absolute independence. (a) Weather and dental problems are independent. (b) Coin flips are independent.

Bayes Rule:

$P(b \mid a) = \frac{P(a \mid b)\, P(b)}{P(a)}$, this can be applied to multivalued variables by being written as the following: $\boldsymbol{P}(Y \mid X) = \frac{\boldsymbol{P}(X \mid Y)\, \boldsymbol{P}(Y)}{\boldsymbol{P}(X)}$ and with some background evidence e, we get the following: $\boldsymbol{P}(Y \mid X, e) = \frac{\boldsymbol{P}(X \mid Y, e)\, \boldsymbol{P}(Y \mid e)}{\boldsymbol{P}(X \mid e)}$. Bayes' rule is important since if we take the example of a medical setting we get that often we see P(effect | cause), and now given this we can newly determine P(cause | effect).

- Bayes' rule with normalization: $\boldsymbol{P}(Y \mid X) = \alpha \boldsymbol{P}(X \mid Y) \boldsymbol{P}(Y)$

Conditional Independence: brought about by direct causal relationships in the domain. As an example take the following three variables: catch, cavity, toothache. If you have a toothache then a dentist may catch a cavity. However, suppose we are given that a cavity exists, then a toothache has nothing to do with whether the dentist will catch the cavity. This means that catch and toothache are conditionally independent based on cavity.

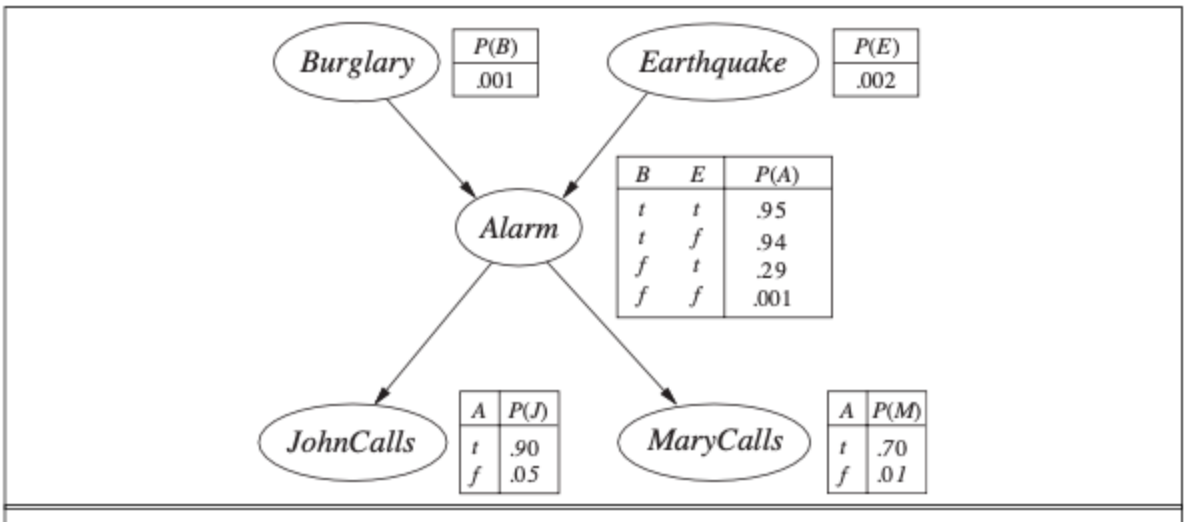
- Independent but Conditionally dependent: suppose event A is: coin 1 lands on heads, and event B is: coin 2 lands on heads. These are independent events by themselves. Add a new event C: both land on the same value. Now A and B are no longer independent events

Chapter 14 (Bayesian Network):

Bayesian Network:

1. Each node corresponds to a random variable, which may be discrete or continuous
2. A set of directed links or arrows connects pairs of nodes. If there is an arrow from node X to node Y, X is said to be a parent of Y. The graph has no directed cycles. It is a DAG.
3. Each node X_i has a conditional probability distribution **P**(X_i | Parents(X_i)) that quantifies the effect of the parents on the node.

The topology of a bayesian network– specifies the conditional independence relationships that hold in the domain.



In the above example we can display the power of Bayesian networks, using the following formula reduction: $P(X_1 = x_1, \Lambda \dots \Lambda X_n = x_n)$ this is a conjunction of particle assignments to each variable and is abbreviated as $P(x_1, \dots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i))$.

As an example we could calculate the probability that the alarm has sounded, but neither a burglary nor an earthquake has occurred, and both John and Mary call:

- $P(j, m, a, \neg b, \neg e) = P(j \mid a)\, P(m \mid a)\, P(a \mid \neg b \Lambda \neg E)\, P(\neg B)\, P(\neg e)$

Markov Blanket: a node is conditionally independent of all other nodes in the network, given its parents, children and children's parents → this forms a Markov Blanket.

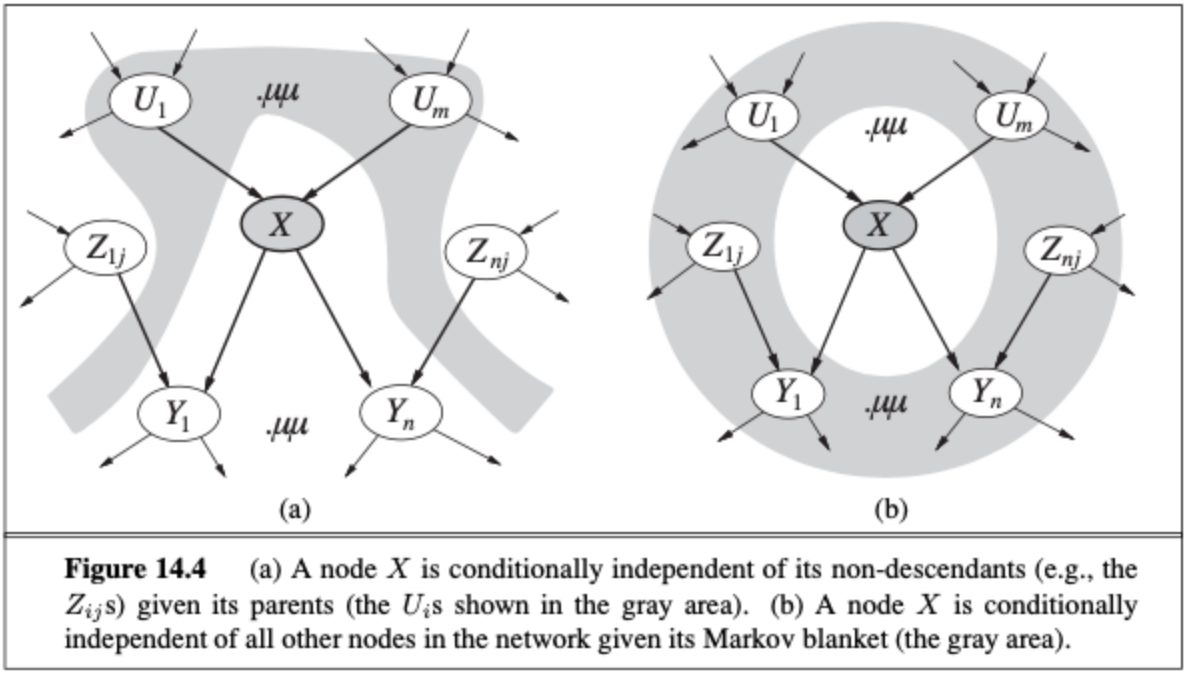


Figure 14.4 (a) A node X is conditionally independent of its non-descendants (e.g., the Z_{ij}s) given its parents (the U_is shown in the gray area). (b) A node X is conditionally independent of all other nodes in the network given its Markov blanket (the gray area).

The topological semantics (Markovian assumptions) state that each variable is conditionally independent of its non descendants, given its parents. (Eg, John is independent of Burglary, Earthquake, and Mary given Alarm.

Determining Conditional Independence from a Bayesian Network:

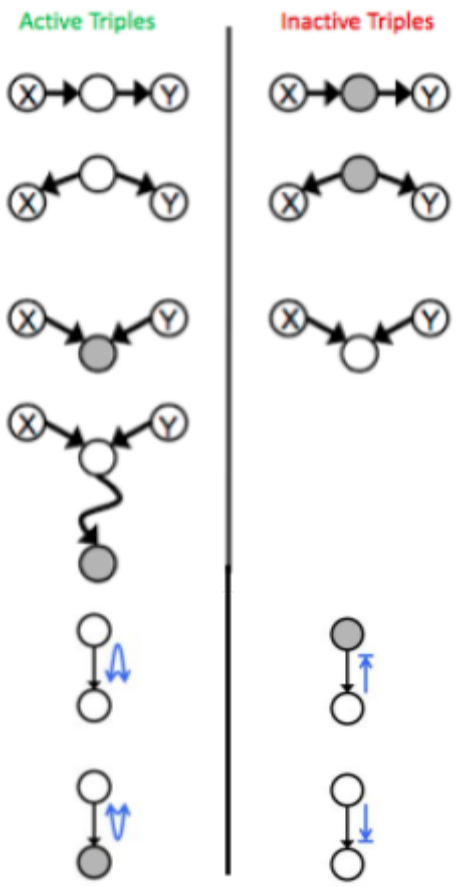
Active Tripels → do not imply conditional independence.

Inactive Triples → do imply conditional independence

(Note the bottom left case is simply an extension of the case above it).

(Gray colored nodes represent information that is given or known)

To prove or disprove independence between variables, run the following algorithm: For every path between the two variables (regardless of direction) if any path is active then the variables are not independent. If there are no active paths, then the variables are independent.



Hidden Variables: Variables that affect that probability but are not in the probability. To deduce this we look at all the parents of the variables and those are our hidden variables.

Exact Inference in Bayesian Networks: a query can be answered using a Bayesian network by computing sums of products of conditional probabilities from the network. We can deduce a P(X | e) from a bayesian network by expanding based on hidden variables according the following formula:

$$P(X \mid e) = \alpha P(X, e) = \alpha \sum_{y} P(X, e, y)$$

And then we can apply the rule $P(x_1, \ldots, x_n) = \prod_{i=1}^{n} P(x_i \mid parents(X_i))$. To get new probability values.