CS3243 Finals Cheatsheet v1.1 (2019-11-25)

by Julius Putra Tanu Setiaji, page 1 of 2

1 Introduction to AI

1.1 Rational Agent

- An **agent** is an entity that perceives its **env** through sensors & acts through **actuators**
- An agent's **percept sequence** is complete history of everything the agent has ever perceived.
- What is rational depends on: (1) The performane measure that defines success, (2) The agent's prior knowledge of the env, (3) The actions that the agent can perform, (4) The agent's percept sequence to date.
- For each possible percept sequence, a **Rational Agent** should select an action that is expected to maximise its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

1.2 Task Environment

- **PEAS**: Performance, Environment, Actuators, Sencsors
- **Fully observable** vs **Partially observable**:: an agent's sensors give it access to the complete state of the env at each point in time **VS** if the sensors detect all aspects that are relevant to the choice of action
- **Single agent** vs **Multiagent**: whether there are any other agent in the environment, multiagent further divided into **competitive** and **cooperative** where **communication** and **randomised behaviour** are the typical rational behaviours respectively
- **Deterministic** vs **Stochastic**: if the next state of the env is completely determined by the current state and the action executed by the agent **VS** otherwise. (partially observable env may appear stochastic)
- **Episodic** vs **Sequential**: the choice of current action does not depend on prev actions **VS** otherwise
- **Static** vs **Dynamic**: if the environment is unchanged while an agent is deliberating **VS** otherwise
- **Discrete** vs **Continuous**: in terms of state of env, time, percepts and actions

1.3 The Structure of Agents

- An **agent program** (takes in current percept) implements the **agent function** (percept sequence): mapping from percept sequence to actions.
- **Table-Driven-Agent**: persists the percept sequence from the current percept, and looks up action from table.
- Drawback: Hube table to build and store (time and space), no autonomy (impossible to learn all correct table entries from experience), no guidance on filling in the correct tabel entries

1.4 Agent Types, in increasing generality

- **Simple Reflex Agent**: passive, only selects actions on the basis of the current percept (ignoring percept history). Updates state based on percept only
- The rest updates state based on percept, current state, most recent action and model of the world.
- **Model-based Reflex Agents**: passive, (to handle partial observability, need to build model of the world)
- **Goal-based Agent**: achieve goal (binary: achieve goal/not).
- **Utility-based Agent**: maximises utility function (measure of happiness: more than binary).
- **Learning Agent**: **Learning element** responsible for making improvements with feedback from the **critic**, **performance element** (what was agent) responsible for selecting external actions, **problem generator** responsible for suggesting actions (do suboptimal now to explore better actions in the long run).

2 Solving Problems by Searching

2.1 Problem Formulation

Initial State, **Actions** (set of actions possible given a particular state), **Transition Models** (description of each action), **Goal Test** (determines whether a state is a goal state), **Path Cost** (assigns a numeric cost to each path)

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

function GRAPH-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  initialize the explored set to be empty
  loop do
    if the frontier is empty then return failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state then return the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set
```

A **node** includes state, parent node, action, and path cost.

2.2 Evaluation criteria

- **Completeness**: always find solution if one exists
- **Optimality**: finding a least-cost solution
- **Time complexity**: no of nodes generated
- **Space complexity**: max. no of nodes in memory

2.3 Problem parameters

- b: max. no of successors of any node
- d: depth of shallowest goal node
- m: max. depth of search tree

Uninformed Search Strategies

Property	BFS	UCS	DFS	DLS	IDS
Complete	Yes*	Yes**	No***	No	Yes*
Optimal	No*	Yes	No	No	No*
Time	$O(b^d t)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^d)$	$O(b^{1+\lfloor \frac{C^*}{\epsilon} \rfloor})$	$O(bm)$	$O(bl)$	$O(bd)$

*: BFS, IDS – complete if b is finite, optimal if step costs are identical
**: UCS is complete if b is finite and step cost $\geq \epsilon$
***: DFS is complete only on infinite depth graphs
C^* is the optimal cost

2.4 Breadth-First Search (BFS)

Expand shallowest unexpanded node, frontier is FIFO

2.5 Uniform-Cost Search (UCS)

Expand least-path-cost unexpanded node, frontier is PQ by path cost. Equivalent to BFS if all step costs are equal

2.6 Depth-First Search (DFS)

- Expand deepest unexpanded node, frontier is LIFO.
- **Backtraking Search**, space can be $O(m)$ if successor is expanded one at a time (partially expanded node remembers which successor to generate next)

2.7 Depth-Limited Search (DLS)

Run DFS with depth limit l, to solve the infinite-path problem

2.8 Iterative Deepening Search (IDS)

- Perform DLS with increasing depth limit.
- Preferred if search space is large and depth of solution is not known

Informed (Heuristic Search Strategies)

Use an **evaluation function** $f(n)$ for each node n

2.9 Greedy best-first search

- $f(n) = h(n)$ = estimated cost of cheapest path from n to goal
- Expands nodes that appear to be closest to the goal
- **Complete** if b is finite, **Non-optimal**, **Time** $O(b^m)$, **Space** $O(b^m)$

2.10 A* Search

- $f(n) = g(n) + h(n)$ where $g(n)$ = path cost from start node to node n
- Avoids expanding paths that are already expensive
- **Admissible Heuristic** never overestimates the cost to reach the goal: $\forall n, h(n) \leq h^*(n)$ where $h^*(n)$ = true cost
- **Consistent Heuristic**: triangle inequality – $h(n) \leq c(n, n') + h(n')$
- Every consistent heuristic is also admissible.
- Theorem: If $h(n)$ is admissible, then A* tree-search is optimal
- Theorem: If $h(n)$ is consistent, then A* graph-search is optimal (from lemma consistent heuristic always follow optimal path)
- **Complete** if there is a finite no of nodes with $f(n) \leq f(G)$, **Optimal**, **Time** $O^{h^*(s_0) - h(s_0)}$ where $h^*(s_0)$ is the actual cost of getting from root to goal, **Space** $O(b^m)$
- **Dominant heuristic**: if $\forall n, h_2(n) \geq h_1(n)$ then h_2 **dominates** h_1
- More dominant heuristics incur lower search cost

3 Beyond Classical Search

- Path to goal is irrelevant, the goal state itself is the solution.
- Advantages: (1) use very little/constant memory, (2) can find reasonable solns in large/infinite continuous state spaces
- Useful for **pure optimization problems**: objective is to find the best state according to an **objective function**

3.1 Hill-climbing Search (aka Greedy Local Search)

- Continually moves in the direction of icnreasing value, terminate when reaching a "peak"
- Possible to get stuck in local maxima, only use if OK with approximate solutions.

4 Adversarial Search

- 2 players, zero-sum game
- **Game formulation**:
  - S_0: The **initial state**
  - PLAYER(s): which player has the move in a state
  - ACTIONS(s): returns the set of legal moves in a state.
  - RESULT(s, a): The **transition model**, defines the result of a move
  - TERMINAL-TEST(s): A **terminal test**, true when game is over
  - UTILITY(s, p): A **utility function**), defines final numeric value for a game that ends in terminal state s for a player p

4.1 Optimal Decisions in Games

- **Winning** strategy for one player if for any strategy played by the other player, the game ended with the former as the winner. Similar for **non-losing** strategy.
- **Nash Equilibrium** – when players know the strategies of all opponents, no one wants to change their strategy.
- **Subgame Perfect Nash Eq** – every subgame is a **Nash Eq**

4.2 Minimax

- Optimal strategy can be determined from **minimax value** of each node: utility (for MAX) of being in the state, assuming both players play optimally from there to end of the game.
- Minimax returns a subperfect Nash equilibrium
- **Complete** (with finite game tree), **Optimal**, **Time** $O(b^m)$, **Space** $O(bm)$

4.3 $\alpha - \beta$ Pruning

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

- MAX node n: $\alpha(n)$ = highest observed value found on path from n, initially $-\infty$
- MIN node n: β = lowest observed value found on path from n, initially $+\infty$
- If a MIN node has value $v \leq \alpha(n)$, can prune
- If a MAX node has value $v \geq \beta(n)$, can prune

4.4 Imperfect Real-time Decisions

- Although very large search space in typical games is pruned by $\alpha - \beta$ pruning, minimax still has to search all the way to the terminal states.
- Replace utility function with heuristic **evaluation function** that estimates the position's utility, and replace the terminal test with a **cutoff test** that decides when to apply EVAL.

4.5 Evaluation Functions

- A mapping from game states to real values.
- Should be cheap to compute; for non-terminal states, must be strongly correlated with actual chances of winning
- Modern eval function: weighted sum of position features
- Need not return actual expected values, just maintain relative order of states, typically from statistically probabilities

4.6 Cutting off Search

Stop after a certain depth, can be combined with iterative deepening

5 Constraint Satisfaction Problems

Consists of 3 components:

- X is a set of variables, $\{X_1, ..., X_n\}$
- D is a set of domains, $\{D_1, ..., D_n\}$, one for each variable
- C is a set of constraints that specify allowable combinations of values

5.1 Terminologies

- **Consistent** assignment = does not vilate any constraints
- **Complete** assignment = every variable is assigned
- Goal: find a consistent and complete assignment
- **Binary constraint** relates 2 variables
- **Global constraint** involve an arbitrary number of variables
- Every finite-domain constraint can be reduced to a set of binary constraints if enough auxiliary variables are introduced.
- **Constraint graph**: nodes are variables, links are constraints

5.2 Variants

- Domain can be **discrete** (both **finite** and **infinite**) or **continuous**
- For discrete, infinite domains, a **constraint language** must be used without enumeration

5.3 Constraint propagation: Inference in CSP

Try to infer illegal values for variables by performing constraint propagation
For unary constraints, node consistency; For binary constraints, arc consistency
Arc Consistency = a variable X_i in CSP is arc-consistent with another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j). A network is arc-consistent if every variable is arc-consistent with every other variable.
Time $O(n^2 d^3)$ where n is number of vars, d is max domain size
K**-consistency** = if, for any set of $k-1$ vars and for any consistent assignment to those variables, a consistent value can always be assigned to any k-th var (arc-consistency is 2-consistency)

```
function AC-3(csp) returns false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  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 REVISE(csp, X_i, X_j) then
      if size of D_i = 0 then return false
      for each X_k in X_i.NEIGHBORS - {X_j} do
        add (X_k, X_i) to queue
  return true

function REVISE(csp, X_i, X_j) returns true iff we revise the domain of X_i
  revised ← false
  for each x in D_i do
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j then
      delete x from D_i
      revised ← true
  return revised
```

5.4 Backtracking Search for CSPs

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

- Better an just doing search, because CSPs are **commutative**
- DFS that chooses values for one variable at a time, and backtracks when a var has no legal values left to assign.
- For SELECT-UNASSIGNED-VARIABLE: use **Most Constrained Variable** choose the var with fewest legal values (Minimum Remaining Values (MRV) heuristic)
- Once a variable is selected, to decide the order to examine its values, use **Least Constraining Value** heuristic: prefer value that rules out the fewest choices for the neighbouring variables in the constraint graph

5.5 Local Search for CSPs

- Similar to hill-climbing, but instead with complete states, allow states that violate constraints, then reassign variable values
- In choosing a new value for a variable, herustic: select the value that results in the minimum number of conflicts with other variables

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

5.6 The structure of problems

- Theorem: if CSP constraint graph (with binary constraints) is a **tree**, then we can compute a satisfying assignment (or determine one does not exist) in $O(nd^2)$ time (no need to backtrack)
- Proof: Pick any variable to be the root of tree, and choose an ordering of vars such that each var appears after its parent in the tree (Toposort: $O(n)$, each of which must compare up to d possible domain values for the two variables)

6 Logical Agents

6.1 Knowledge-based Agents

- Inference Engine (domain-independent algorithms)
- **Knowledge base** = set of sentences in a formal language (domain-specific content)
- Declarative approach to building an agent: TELL it what it needs to know, then it can ASK itself what todo according to the KB

6.2 Logic

- **Logic** = formal language for KR, infer conclusions
- **Syntax** = defines the sentences in the language
- Semantics = define the "meaning" of sentences (truth of a sentence in a world)

6.3 Entailment

- **Modeling**: m models α if α is true under m.
- Let $M(\alpha)$ be the set of all models for α
- **Entailment** means that one thing follows logically from another sentence: $\alpha \models \beta \iff M(\alpha) \subseteq M(\beta)$
- Properties of inference algorithms:
  - **Sound** or **Truth-preserving** – derives only entailed sentences
  - **Completeness** – it can derive any sentence that is entailed

7 Propositional Logic

7.1 Syntax

- Atomic sentences consisting of a single proposition symbol
- Complex sentences constructed from simpler sentences using parantheses and **logical connectives**: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\implies$ (implies), $\iff$ (iff/biconditional)

7.2 Semantics

A **truth assignment** to every proposition symbol.

7.3 Validity and Satisfiability

- A sentence is **valid** if it is true in **all** models (e.g. $A \Rightarrow A$, $A \vee \neg A$)
- A sentence is **satisfiable** if it is true in **some** model (e.g. $A \vee B$)
- A sentence is **unsatisfiable** if it is true in **no** model (e.g. $A \wedge \neg A$)

7.4 Inference

- By **Truth-Table Enumeration**, DFS is sound and complete, time $O(2^n)$, space $O(n)$
- **Deduction Thm**: $KB \models \alpha \iff (KB \implies \alpha)$ is valid $\iff (KB \wedge \neg\alpha)$ is unsatisfiable

7.5 Inference Rules

- **Modus Ponens**: $a \wedge (a \implies b) \models b$
- **And-Elimination**: $a \wedge b \models a$
- Logical equivalences:

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

7.6 Resolution for Conjunctive Normal Form (CNF)

- Conjunction of disjunction of literals
- if a literal x appears in a clause and its negation $\neg x$ appears in another clause, it can be deleted
- Resolution is **sound** and **complete** for propositional logic
- **Resolution thm**: if a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause

7.7 Forward and Backward Chaining

- Horn clause = disjunction of literals of which at most 1 is positive
- Both forward and backward chaining with Horn clauses run in **linear** time
- Inference using **Modus Ponens**: **sound** for Horn KB

7.7.1 Forward Chaining

- Data-driven reasoning, e.g. object recognition, routine decisions.
- May do a lot of work 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
```

7.7.2 Backward Chaining

- To prove q by BC, check if q is known already, or prove by BC the premise of some rule concluding q
- **Avoid loops**: check if new subgoal is already on the goal stack
- **Avoid repeated work**: check if new subgoal already failed or proven true
- Goal-driven reasoning, complexity can be sublinear in size of KB
- Improvements over truth table enumeration:
  - **Early termination**: a clause is true iff any literal in it is true; the formula is false if any clause is false.
  - **Pure symbol heuristic** (Least constraining value): **pure symbol** always appear with the same "sign" in all clauses, make a pure symbol's literal true, ignore clauses already true in the model constructed so far
  - **Unit clause heuristic**: Unit clause = only 1 literal in the clause, the only literal in a unit clause must be true.

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })
```

```
function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false}))
```

7.8 Local Search Algo: WalkSAT

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
    if model satisfies clauses then return model
    clause ← a randomly selected clause from clauses that is false in model
    with probability p flip the value in model of a randomly selected symbol from clause
    else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

8 First-Order Logic (FOL)

8.1 Syntax

- **Constants**, e.g. John, 2, NUS
- **Predicates**, e.g. $Brother(x, y)$, $x > y$
- **Functions**, e.g. $\sqrt{x}$, $LeftLeg(x)$
- **Variables**, e.g. x, y, a, b
- **Operator Precedence**: $\neg, =, \wedge, \vee, \implies, \iff$
- **Quantifiers**: $\forall, \exists$
- **Atomic Sentences**: constant or variable or function or predicate
- **Complex Sentences**: constructed from atomic sentences via connectives

8.2 Semantics

Sentences are true in a model, comprising a set of objects (domain elements) & an interpretation

8.3 Quantifiers

- Universal ($\forall$): uses $\implies$, equivalent to conjunction of instantiations
- Negation of $\forall x : P(x)$ is $\exists x : \neg P(x)$
- Existential ($\exists$): uses $\wedge$, equivalent to disjunction of instantiations
- Negation of $\exists x : P(x)$ is $\forall x : \neg P(x)$

8.4 Knowledge Engineering in FOL

(1) Identify the task, (2) Assemble the relevant knowledge, (3) Decide on a vocabulary of predicates, functions, and constants, (4) Encode general knowledge about the domain, (5) Encode a description of the specific problem instance, (6) Pose queries to the inference procedure and get answers, (7) Debug the knowledge base,

9 Inference in FOL

9.1 Reduction to Propositional Inference

- Every FOL KB can be propositionalised, preserves entailment: α is entailed by new KB iff entailed by original KB
- **Herbrand Thm**: If α if entailed by FOL KB, then it is entailed by a **finite subset** of the propositionalised KB.
- For $n = 0$ to ∞, create propositionalized KB_n by instantiating with depth-n terms, see if α is entailed by this KB_n, semi-decidable (return TRUE if entailed, but cannot return FALSE if not entailed)
- Exponential blowup: k-ary predicate has n^k instantiation with n constants, some of the things generated irrelevant
- Rule of **Universal Instantiation**: we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable.
- Rule of **Existential Instantiation**: variable is replaced by a single new constant symbol that has not appeared elsewhere in the KB.

9.2 Unification

- Find a substitution θ such that different logical expressions look identical
- **Standardising apart** one of the two sentences being unified (rename to avoid name clash)
- There is a single unique **most general unifier** (MGU) up to renaming and substitution of variables
- **Occur check** to check whether the variable itself occurs inside the term, in which the match fails, e.g. $S(x)$ with $S(S(x))$ – this makes the entire algo quadratic in the size of the expressions being unified

```
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} ∈ θ then return UNIFY(val, x, θ)
  else if {x/val} ∈ θ then return UNIFY(var, val, θ)
  else if OCCUR-CHECK?(var, x) then return failure
  else return add {var/x} to θ
```

9.3 Generalized Modus Ponens (GMP)

There is some substitution θ such that the premise and the LHS of implication are the same, can infer RHS of implication

9.4 Forward Chaining

At every round, add all newly inferred atomic sentences to KB. Repeat until: one of these sentences is α or no new sentences can be inferred

```
function FOL-FC-ASK(KB, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, the query, an atomic sentence
  local variables: new, the new sentences inferred on each iteration

  repeat until new is empty
    new ← { }
    for each rule in KB do
      (p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES(rule)
      for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
                for some p'1, ..., p'n in KB
        q' ← SUBST(θ, q)
        if q' does not unify with some sentence already in KB or new then
          add q' to new
          φ ← UNIFY(q', α)
          if φ is not fail then return φ
    add new to KB
  return false
```

9.4.1 Properties

- **Sound** and **complete** for FOL definite clauses (disjunction of literals, exactly 1 is positive)
- Terminates in finite no of iterations if KB contains no functions
- May not terminate in general (with functions) if α is not entailed

9.4.2 Inefficiencies of Foward Chaining

- Matching rule premises to known fact (pattern matching) is costly. Solution:
  - **Conjunct ordering problem**: apply minimum-remaining-values heuristic of CSP,
  - **Predicate indexing**: constant time to retrieve known facts
- Redundant rule matchings. Solution:
  - **Incremental foward chaining**: match rule at time t only if a conjunct in its premise unifies with new fact inferred at $t-1$
  - **Rete algorithm**: don't discard partially matched rules, keep track of conjuncts matched against new facts, avoid duplicate work
- Generating irrelevant facts. Solution: use backward chaining

9.5 Backward Chaining

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })
```

```
generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'
```

```
generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first,rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

9.5.1 Properties

- DFS: space is linear in size of proof
- Incomplete due to infinite loops: fix by checking current goal against every goal on stack
- Inefficient due to repeated subgoals (both success and failure): fix by caching solutions to previous subgoals

9.6 Resolution: convert to CNF

1. Standardize variables,
2. Skolemize existential quantifiers (replace with functions depending on external universal quantifier),
3. Drop universal quantifiers,
4. Distribute $\vee$ over $\wedge$

9.6.1 Resolution Inference Rule

- FOL literals are complements if one unifes with negation of the other.
- First-order factoring: removes redundant literals by reducing 2 literals to one if they are unifiable (similar to propositional reducing 2 literals to one if they are identical).
- To prove $KB \models \alpha$, show that $KB \wedge \neg\alpha$ results in contradiction

9.6.2 Properties

Complete

10 Uncertainty

10.1 Sources of uncertainty

(1) Partial observability, (2) Noisy sensors, (3) Uncertainty in action outcomes, (4) Complexity in modelling and predicting traffic

10.2 Events

Atomic events = an assignment of a value to each random var; a singleton event

10.3 Axioms of probability

- Let X be an r.v. with finite domain D_X
- A prob distribution over D_X assigns a value $p_X(x) \in [0,1]$ to every $x \in D_X$ s.t. $\sum_{x \in D_X} p_X(x) = 1$
- $\Pr(A) + \Pr(B) = \Pr(A \cap B) + \Pr(A \cup B)$
- $\Pr(A \mid B) = \frac{\Pr(A \wedge B)}{\Pr(B)}$ assuming $\Pr(B) > 0$
- **Independent** if $\Pr(A \wedge B) = \Pr(A)$, equiv to $\Pr(A \mid B) = \Pr(A)$
- **Summing out**: $\Pr(X) = \sum_z \Pr(X, z)$ where z is all possible value of other vars
- **Normalization**: in conditional, the prob of the given event is a constant α
- **Bayes rule**: $\Pr(A \mid B) = \frac{\Pr(B|A)\Pr(A)}{\Pr(B)}$
- **Chain rule**: derived by successive application of Bayes rule: $\Pr(X_1 \wedge X_2 \wedge \ldots \wedge X_k) = \prod_{j=1,\ldots,k} \Pr(X_j \mid X_1 \wedge \ldots \wedge X_{j-1})$

10.4 Conditional Independence

- Events are independent of each other, only related by the given event
- $\Pr(B \wedge T \mid S) = \Pr(B \mid S)\Pr(T \mid S)$
- Full joint distribution by chain rule: where effects are T_i, cause is S: $\Pr(T_1 \wedge T_2 \wedge \ldots \wedge T_n \wedge S) = \Pr(T_1 \mid S)\Pr(T_2 \mid S)\ldots\Pr(T_n \mid S)\Pr(S)$
- Joint distribution of n boolean r.v. $= 2^n - 1$ entries
- Conditional independence is linear

11 Bayesian Networks

- Nodes are random variables, edge from X to Y : X directly influences Y
- A conditional distr for each node given its parents: $\Pr(X \mid Parents(X))$, represented as **Conditional Probability Table** (CPT): the distr of X for each combination of parent values
- Given $X_1, \ldots, X_n$: $\Pr(X_1 \wedge \ldots \wedge X_n) = \prod_i \Pr(X_i \mid Parents(X_i))$

11.1 Markov Blanket

A node is conditionally independent of everything else given the values of its **parents, children, its children's parents**

11.2 d-separation

1. **Draw the ancestral graph**: consisting of only all vars mentioned in prob expression and all their ancestors
2. **"Moralize" the ancestral graph by "marrying" the parents**: for *each pair* of variables with common child, draw an undirected edge betweeen them
3. **"Disorient" the graph by replacing directed edges with undirected edges**
4. **Delete the given and their edges**
5. **Read the answer off the graph**:
   - If vars **disconnected** in graph: guaranteed to be independent
   - If vars **connected** in graph: not guaranteed to be indep (dependent as far as Bayes net is concerned – can still be numerically indep)
   - If one/both of the vars missing (because they were givens), independent

```
function ENUMERATION-ASK(X, e, bn) returns a distribution over X
  inputs: X, the query variable
          e, observed values for variables E
          bn, a Bayes net with variables {X} ∪ E ∪ Y   /* Y = hidden variables */

  Q(X) ← a distribution over X, initially empty
  for each value x_i of X do
    Q(x_i) ← ENUMERATE-ALL(bn.VARS, e_{x_i})
        where e_{x_i} is e extended with X = x_i
  return NORMALIZE(Q(X))
```

```
function ENUMERATE-ALL(vars, e) returns a real number
  if EMPTY?(vars) then return 1.0
  Y ← FIRST(vars)
  if Y has value y in e
    then return P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e)
    else return Σ_y P(y | parents(Y)) × ENUMERATE-ALL(REST(vars), e_y)
        where e_y is e extended with Y = y
```