CS3243 Introduction to Artificial Intelligence
Help Sheet for Final Examinations (Second Half of Semester)

Constraint Satisfaction Problems (CSP)

Constraint Satisfaction Problems.

We are only interested in the solution. There are no fixed goal states that we want to reach. Instead, we define the goal state as one that is able to fully assign all our variables while satisfying all our constraints.

Component	Description
Set of Variables, X	$\{x_1, x_2, \cdots, x_n\}$
Set of Domains, D	$\{D_1, D_2, \cdots, D_n\}$ where $D_i = domain(x_i)$
Set of Constraints, C	Each constraint C_i is a pair $< scope, rel >$, $scope$ is a tuple of variables involved in the constraint, rel is a relation defining the values that the tuple of variables can take.

Table: Components of an CSP

Backtracking Search with Inference.

```
BacktrackSearch(prob, assign):
    if AllVarsAssigned(prob, assign):
        return assign
    var ← PickUnassignedVar(prob, assign)
    for val in OrderDomainValue(var, prob, assign):
        if (val is consistent with assign):
            assign.add(var=val)
            inference ← Infer(prob, var, assign)
            assign.add(inference)
            if (inference != failure):
                result ← BacktrackSearch(prob, assign)
                if (result != failure):
                    return result
        remove {var=val} and inference from assign
    return failure
```

Executing the algorithm with the context of the N-Queens Problem:

Pick x_1.
Set $x_1 = 1$.
$Inference$ returned as $failure$.
Remove $x_1 = 1$.
Set $x_1 = 2$.
$Inference$ returned as $\{x_2 = 4, x_3 = 1, x_4 = 3\}$.
Return $assignments$ because all variables are assigned.

Inference.

```
Infer(prob, var, assign)
  inference ← empty
  varQueue ← [var]
  while (varQueue is not empty):
    Y ← varQueue.pop()
    for each constraint C where Y ∈ Vars(C)
      for all X ∈ Vars(C)\Y:
        S ← ComputeDomain(X, assign, inference)
        for each v in S:
          if (no value exists in Var(C)\X
          s.t c[X↦v] is satisfied):
            inference.add(x ∉ {v})
        T ← ComputeDomain(X, assign, inference)
        if (T = {}):
          return failure
        if (S ≠ T):
          varQueue.add(X)
  return inference
```

The representation of an $inference$ on a variable X is defined as $\{X \notin S\}$, where S is a set of values in which the variable X cannot take.

`ComputeDomain(X, assign, inference)` returns the set of values of X that X can take under the current assignment and inference.

Alternative implementations:
Choice 1 (Forward Checking): Don't add anything to varQueue.
Choice 2: Instead of "`if (S ≠ T)`", we replace it with "`if (|T| = 1)`"

Making an Inference. In the context of the N-Queens Problem, suppose $assign = \{x_1 = 1\}, var = x_1$.

$NoAttack(x_1, x_2)$	$x_1 = 1 \Rightarrow x_2 \notin \{1,2\} \Rightarrow x_2 \in \{3,4\}$
$NoAttack(x_1, x_3)$	$x_1 = 1 \Rightarrow x_3 \notin \{1,3\} \Rightarrow x_3 \in \{2,4\}$
$NoAttack(x_1, x_4)$	$x_1 = 1 \Rightarrow x_4 \notin \{1,4\} \Rightarrow x_4 \in \{2,3\}$
$NoAttack(x_2, x_4)$	If $x_2 = 3$ then $x_4 = 2$. If $x_2 = 4$ then $x_4 = 3$.
$NoAttack(x_2, x_3)$	If $x_2 = 3$ then $x_3 \notin \{2,4\} \wedge x_3 \notin \{1,3\} \Rightarrow x_3 \in \{\} \Rightarrow x_2 \neq 3$ If $x_2 = 4$ then $x_3 \notin \{4\} \wedge x_3 \notin \{1,3\} \Rightarrow x_3 = 2$ Hence $x_2 = 4$, $x_3 = 2$.
$NoAttack(x_3, x_4)$	$x_3 = 2 \Rightarrow x_4 \notin \{1,2,3\} \wedge x_4 \notin \{1,4\} \Rightarrow x_4 \in \{\}$

Return failure.

Markov Decision Processes (MDP)

Markov Decision Process (MDP). Given an action a, the action's outcome is only dependent on the present state s.

Policy. A policy $\pi: States \rightarrow Actions$ will give an action for each state.

Optimal Policy. An optimal policy π^* is one that maximizes the expected utility and minimizes the cost if followed.

Component	Description
State(s)	$s \in S$
Action(s)	$a \in A$
Transition Model	$T(s,a) = \begin{cases} s_1 \; with \; probability \; p_1 \\ \dots \\ s_k \; with \; probability \; p_k \end{cases}$
Reward Function	$R(s,a)$ or $R(s)$
Initial State	s_0
Goal State(s)	Flexible
Terminal State(s)	States in S where no action is taken after you reach those states.

Table: Components of an MDP

Utility of a Path. The utility function $U: Sequence \; of \; States \rightarrow \mathbb{R}$ denotes the "usefulness" of a path T. The discounted utility function is defined as

$$U_n([s_0, s_1, \dots, s_n]) = R(s_0) + \gamma R(s_1) + y^2 R(s_2) + y^n R(s_n)$$

$$U(T) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \cdots = \sum_{t=0}^{\infty} \gamma^t R(s_t)$$

s_t is a random variable, which denotes the state that s will be at time t.

If there is a maximum reward R_{max} such that for all s_i, $R(s_i) \le R_{max}$, then for any path,

$$U_n([s_0, s_1, \dots, s_n]) \le \frac{R_{max}}{1-\gamma}$$

Utility of a State. The utility for state s for policy π is defined as the expected value of the utility of its path,

$$U^\pi(s) = E[U(T)] = E[\sum_{t=0}^{\infty} \gamma^t R(s_t)]$$

Derivation of Optimal Policy. To find an optimal policy $\pi^*(s)$,

$$\pi^*(s) = (\underset{\pi}{argmax} \, U^\pi(s))(s)$$

$$\pi^*(s) = \underset{a \in A(s)}{argmax} \, P(s'|s,a) \cdot U^{\pi^*}(s')$$

Optimal policy should return optimal action a given s.

$$\pi^*(s) = \underset{a \in A(s)}{argmax} \, P(s'|s,a) \cdot U(s)$$

Optimal policy is independent of the start state, so $U(s) = U^{\pi^*}(s')$

Derivation of Bellman Equation. The Bellman Equation gives us the utility function (under optimal policy), $U^{\pi^*}(s)$.

$$\begin{aligned} U^{\pi^*}(s) & \\ &= E[\sum_{t=0}^{\infty} \gamma^t R(s_t)] && \text{(from definition of utility of a state for a given policy)} \\ &= E[R(s_0) + \sum_{t=1}^{\infty} \gamma^t R(s_t)] && \text{(expand equation for } t = 0\text{)} \\ &= R(s) + E[\sum_{t=1}^{\infty} \gamma^t R(s_t | s_0 = s)] && \text{(agent receives full reward for current state } s\text{)} \end{aligned}$$

$$\begin{aligned} E[\sum_{t=1}^{\infty} \gamma^t R(s_t | s_0 = s)] & \\ &= \sum P(s' \mid s, \pi^*(s)) \cdot \gamma \left(R(s') + E\left[\sum_{t=2}^{\infty} \gamma^{t-1} (R(s_t \mid s_1 = s'))\right]\right) && \text{(expand the expectation for all possible } s'\text{)} \\ &= \sum P(s' \mid s, \pi^*(s)) \cdot \gamma \left(R(s') + E\left[\sum_{t'=1}^{\infty} \gamma^{t'} (R(s_{t'} \mid s_0 = s'))\right]\right) && \text{(sub. } t' = t-1\text{)} \\ &= \sum P(s' \mid s, \pi^*(s)) \cdot \gamma \left(U^{\pi^*}(s)\right) && \text{(sub. latest definition of optimal utility)} \\ &= \gamma \sum P(s' \mid s, \pi^*(s)) \, U(s') && \text{(pull } \gamma \text{ out of the summation)} \\ &= \gamma \max_{a \in A(s)} (\sum_{s'} P(s' \mid s, a) \, U(s')) && \text{(optimal policy always produces best action)} \end{aligned}$$

We get the **Bellman Equation**, which is

$$U(s) = R(s) + \gamma \max_{a \in A(s)} (\sum_{s'} P(s' \mid s, a) \, U(s'))$$

Q-Learning

Value Iteration. Suppose that you have n states s, this algorithm gives the utility $U(s)$ for all those states within an acceptable error threshold.

We are exploiting the property of the **convergence** of the Bellman Equation, where

$$\lim_{i \to \infty} U_i(s) = U(s)$$

Initialization:
Initialize utilities to some value (eg. 0).

Repeat:
$U_i(s) \leftarrow$ utility value at iteration i
$U_{i+1}(s) \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a) \, U_i(s')$

Termination:
If $\forall s, |U_{i+1}(s) - U_i(s)| < threshold$, then we can stop repeating the above step.

Proof of Convergence.

Lemma: For $U_{i+1} = BU_i$, B is a contraction.
B is a contraction $\Leftrightarrow \|BU_x - BU_y\|_\infty \le \gamma \|U_x - U_y\|_\infty$

A function f is a **contraction** if $|f(a) - f(b)| \le k|a-b|$.
The **max norm** of a vector, $\|U\|_\infty = \max_i |U_i|$

Proof of Lemma:

$$\begin{aligned} |BU_x(s) - BU_y(s)| & \\ &= \left| R(s) + \gamma \max_a \sum_{s'} P(s' \mid s,a) \, U_x(s') - (R(s) + \gamma \max_a \sum_{s'} P(s' \mid s,a) \, U_y(s')) \right| && \text{(apply the Bellman Update)} \\ &= \left| \gamma \max_a \sum_{s'} P(s' \mid s,a) \, U_x(s') - \gamma \max_a \sum_{s'} P(s' \mid s,a) \, U_y(s') \right| && \text{(cancel out reward for current state)} \\ &\le \gamma \max_a \left| \sum_{s'} P(s' \mid s,a) \, U_x(s') - \sum_{s'} P(s' \mid s,a) \, U_y(s') \right| && \text{(move } \gamma \text{ and } max \text{ out)} \\ &= \gamma \max_a \left| \sum_{s'} P(s' \mid s,a) \, [U_x(s') - U_y(s')] \right| && \text{(combine the summations)} \\ &\le \gamma \max_a \left| \sum_{s'} P(s' \mid s,a) \, \|U_x - U_y\|_\infty \right| && \text{(incorporate definition of max norm)} \\ &= \gamma \max_a \left(\|U_x - U_y\|_\infty \left| \sum_{s'} P(s' \mid s,a) \right| \right) && \text{(move max norm out of the summation)} \\ &= \gamma \max_a \left(\|U_x - U_y\|_\infty \right) && \text{(sum of all probabilities is one)} \\ &= \gamma \|U_x - U_y\|_\infty && \text{(max norm is independent of } a\text{)} \end{aligned}$$

Main Proof: $\lim_{i \to \infty} U_i = U^* \Leftrightarrow \lim_{i \to \infty} \|U_i - U^*\|_\infty = 0$

$$\begin{aligned} \|U_i - U^*\|_\infty & \\ &= \|BU_{i-1} - BU^*\|_\infty && \text{(Incorporate contraction B. For optimal policy } U^*, BU^* = U^*\text{)} \\ &\le \gamma \|U_{i-1} - U^*\|_\infty && \text{(Incorporate Lemma)} \\ &\le \gamma^i \|U_0 - U^*\|_\infty && \text{(Recursively incorporate contraction B)} \end{aligned}$$

$$\lim_{i \to \infty} \|U_i - U^*\|_\infty \le \lim_{i \to \infty} (\gamma^i \|U_0 - U^*\|_\infty) = 0 \qquad (0 < \gamma < 1 \text{ so } \gamma^\infty = 0)$$

Hence proven that $\lim_{i \to \infty} U_i(s) = U(s)$. ■

Policy Iteration.

The algorithm takes in an MDP problem and returns an optimal policy π^*.

Initialization:
Initialize policy as an arbitrary mapping $\pi_0(s): States \rightarrow Actions$.

Repeat:
Now that we know policy π_i, for each state s, form the equation

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} P(s' \mid s, \pi_i(s)) \, U^{\pi_i}(s')$$

We have n states, hence we generate n linear equations. Solve them using Gaussian Elimination, which runs in $O(n^3)$ time. Following that, update the policy as

$$\pi_{i+1}(s) = \underset{a \in A(s)}{argmax} \sum_{s'} P(s' \mid s, a) \, U^{\pi_i}(s)$$

Convergence:
If $\forall s \lim_{i \to \infty} \pi_i(s) = \pi^*(s)$, we can terminate the loop.

Q-Learning. Uses the explore (which contributes to learning of the model and help long-term well-being) and exploit (to maximize current rewards) strategy.

Derivation of Q-Function.

$$\begin{aligned} U(s) & \\ &= R(s) + \gamma \max_{a \in A(s)} (\sum_{s'} P(s' \mid s,a) \, U(s')) && \text{(from the Bellman Equation)} \\ &= \max_{a \in A(s)} (R(s) + \gamma \sum_{s'} P(s' \mid s,a) \, U(s')) && \text{(place } max \text{ outside)} \\ &= \max_{a \in A(s)} (Q(s,a)) && \text{(define } Q(s,a)\text{)} \\ Q(s,a) & \\ &= R(s) + \gamma \sum_{s'} P(s' \mid s,a) \, U(s') \\ &= R(s) + \gamma \sum_{s'} P(s' \mid s,a) \max_{a' \in A(s')} (Q(s',a')) && \text{(express utility of subsequent state } s' \text{ in terms of a Q-Function)} \end{aligned}$$

Interpretation. Q-Function is the reward right now, plus expected utility in future after taking action a assuming future actions are optimal.

As we don't know the transition model $T(s,a)$, then $P(s' \mid s,a)$ is unknown.

$$Q(s,a) = R(s) + \gamma \max_{a' \in A(s')} (Q(s',a'))$$

Assume $P(s' \mid s,a) = 1$. But this tends to overestimate the actual expected utility in the future.

Bellman Update. Hence, every time the Agent updates his estimate $\hat{Q}$ of Q, we weight the new estimate by α and the previous estimate by $(1-\alpha)$.

$$\hat{Q}(s,a) \leftarrow \alpha \left(R(s) + \gamma \max_{a'} \left(\hat{Q}(s',a') \right) \right) + (1-\alpha) \hat{Q}(s,a)$$

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

Learning rate. α should be initially high (close to 1) but tends to 0 with time. It can be:
$\alpha(t) = \frac{1}{t}$ or any function that decreases in t.
Any function that decreases in $N(s,a)$ which is number of times agent has taken action a at state s.

Q-Learning Algorithm. Chooses the best action based on what it has learnt so far at the current state.

```
Q-Learning-Agent(s',r'):
    if isTerminal(s):
        Q[s,None] ← r'
    if s is not null:
        $N[s',a'] += 1$
        $\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha \left( R(s) + \gamma \max_{a'} \left( \hat{Q}(s',a') \right) - \hat{Q}(s,a) \right)$
```

`s',r'`: current state and rewards

`s,a,r`: previous state, action and reward, initially null

Joshua Chew Jian Xiang

```
    s ← s'
    r ← r'
    a ← ChooseAction(s')
    return a

ChooseAction(s):
    â ← $\underset{a\in A(s)}{argmax}\ \hat{Q}(s,a)$
    With probability β we choose â.
    With probability (1 − β) we choose an action
randomly or based on $\hat{Q}(s,a)$.
```

$\hat{Q}$: A dictionary such that $\hat{Q}[s,a]$ gives Q-value for state s and action a.

N: A dictionary such that N[s,a] gives number of times agent has taken action a at state s.

Approximate Q-Learning Algorithm. Let function $f_i(s,a)$ denote a feature i of state s and action a that returns some numerical value, eg. Number of pawns on a chessboard. We now approximate $Q(s,a)$ as a weighted linear combination of features,

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a)w_i$$

Now, instead of updating $\hat{Q}(s,a)$ at every iteration, you update all the weights instead, where for $i = 1$ to n,

$$\widehat{w}_i \leftarrow \widehat{w}_i + \alpha[R(s) + \gamma \max_{a'} Q(s',a') - \hat{Q}(s,a)](\frac{\partial \hat{Q}(s,a)}{\partial \widehat{w}_i})$$

Note that $\frac{\partial \hat{Q}(s,a)}{\partial \widehat{w}_i} = f_i(s,a)$.

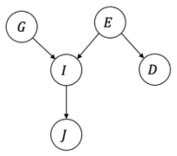
Bayesian Networks

Probability Laws. In case you forgot.

Law	Formula	
Bayes' Rule	$P(A\|B) = \frac{P(A\cap B)}{P(B)} = \frac{P(A)\cdot P(B\|A)}{P(B)}$	
Law of Total Probability	If $B_1, B_2, B_3 \cdots$ serve as a partition of the sample space S, then $P(A) = \sum_i P(A\cap B_i) = \sum_i P(A\|B_i)\cdot P(B_i)$	
Basics of Probability	$0 \le P(A) \le 1$	$P(A\cup B) = P(A) + P(B) - P(A\cap B)$
	$P(True) = P(U) = 1$ where U is the universal set.	$P(False) = P(\phi) = 0$ where ϕ is the empty set.
Independence	A and B are independent if $P(A \mid B) = P(A)$.	
Conditional Independence	Given B, A is conditionally independent of C if $P(A \mid B,C) = P(A \mid B)$. This means that $A ⫫ C\|B$.	

Bayesian Networks. Each variable represented as a node. The <u>causal relation</u> is represented by a directed edge.

Property. For each node in the Bayesian Network, given the value of the node's parents, the node is conditionally independent of its non-descendants.



$P(I\|G,E,D) = P(I\|G,E)$	I, given G and E, is conditionally independent of D.
$P(D\|E,G) = P(D\|E)$	D, given E, is conditionally independent of G.

It is not possible to determine conditional independence if some of the nodes descendants are given, or if some of the node's parents are not given. Eg. We cannot say that $P(I|J,D,E) = P(I|D,E)$.

Chain Rule. $P(X_1,X_2,\cdots,X_n) = P(X_1|X_2,\cdots,X_n) \times P(X_2|X_3,\cdots,X_n) \times \cdots \times P(X_{n-1}|X_n) \times P(X_n)$

Computing Probabilities.

$$\begin{aligned} &P(G,I,J,E,D) \\ &= P(J,I,G,D,E) && \text{(get reverse topological ordering)} \\ &= P(J|I,G,D,E)\cdot P(I|G,D,E)\cdot P(G|D,E)\cdot P(D|E)\cdot P(E) && \text{(apply chain rule)} \\ &= P(J|I)\cdot P(I|G,E)\cdot P(G|D,E)\cdot P(D|E)\cdot P(E) && \text{(extract conditional independence)} \end{aligned}$$

Reverse Topological Sort. Recursively choose nodes without children and remove them.

```
ReverseTopoSort(graph):
    list ← Empty()
    while HasNode(graph):
        node ← PickNodeWithoutChildren(graph)
        list.append(node)
        graph.remove(node)
    return list
```

Deciding on a Model. Given two different Bayesian Network models M_1 and M_2, to decide which gives a better description of the world,

First collect a number of observations or evidence.

$$e_1: G = T, E = T, D = F, I = T, J = F$$
$$e_2: G = T, E = F, D = T, I = T, J = F$$

We want to compare $P(M_1|\{e_1,e_2\})$ and $P(M_2|\{e_1,e_2\})$.

$P(M_1\|\{e_1,e_2\}) = \frac{P(\{e_1,e_2\}\|M_1)P(M_1)}{P(\{e_1,e_2\})}$	(by Bayes' Rule)
$P(\{e_1,e_2\}\|M_1) = P(e_1\|M_1)\cdot P(e_2\|M_1)$	(by data independence)
$P(e_j\|M_i) = P(G,E,D,I,J\|M)$	(can calculate using topo sort)

We can ignore the computation of $P(\{e_1,e_2\})$ since both $P(M_1|\{e_1,e_2\})$ and $P(M_2|\{e_1,e_2\})$ share the same denominator.

Ultimately, we want to obtain $\frac{P(M_1)}{P(M_2)}$.

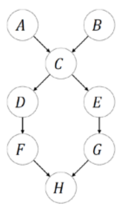
Coming up with Models. To obtain an optimal Bayesian Network model based on a set of evidence, we use the following algorithm similar to a local search.

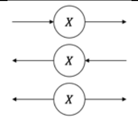
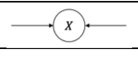
1. Initialize some random model M.
2. At every step, look for a neighbor $N(M)$ of the current model.
3. Compute $\frac{P(evidence|N(M))}{P(evidence|M)}$.
4. Transition to a new model, or remain on the current model, based on the probability ratio that is computed.

Inferring Conditional Independence.

Bayes' Ball Algorithm. Given a Bayesian Network, we want to find if node F and G are independent, given knowledge of the value (ie. *observing*) C.

Check if there is a path between the two nodes F and G in question, following these rules:



	**X observed:** Path is broken. **X not observed:** Path is free.
	**X observed:** Path is free. **X not observed:** Path is broken.

F and G are dependent if there is a dependency path between them in the graph.

Knowledge Representation

Definition of Knowledge. An agent A knows that a statement S is true if and only if (1) S is true, (2) A believes that S is true, (3) A is justified in believing that S is true.

Propositions. Propositions are variables that take on values `TRUE` or `FALSE`.

Set	Definition
$PROP$	The set of all propositions.
$FORM$	The set of all formulas that can be expressed by propositional logic.

Table: Sets relevant to Knowledge Representation

Recursive Definition of $\boldsymbol{FORM}$. $FORM$ can be recursively defined as:

$$FORM_0 = PROP$$
$$FORM_{i+1} = FORM_i \cup \{(\alpha \circ \beta) | \alpha,\beta \in FORM_i\} \cup \{(\neg\alpha) | \alpha \in FORM_i\}$$
$$FORM = \bigcup_{i=0}^{\infty} FORM_i$$

where $(\alpha \circ \beta) \equiv \{\alpha \vee \beta\} \cup \{\alpha \wedge \beta\}$

Knowledge Base ($\boldsymbol{KB}$). An agent's Knowledge Base (KB) is defined as everything that the agent knows, where $\forall\alpha \in KB$ is also in $FORM$.

Truth Assignment ($\boldsymbol{\tau}$). A truth assignment τ is defined as a mapping of a proposition to values `TRUE (0)` or `FALSE (1)`. $\tau: PROP \longmapsto \{0,1\}$

Logical Statement ($\boldsymbol{\varphi}$). In a logical statement φ with $|PROP|$ number of propositions, there are $2^{|PROP|}$ possible combinations of truth assignments.

Modelling ($\vDash$). A truth assignment τ models a statement φ (ie. $\tau \vDash \varphi$), if and only if φ holds true for all mapping in τ, ie. $\varphi(\tau) = 0$.

Given the following statement $\varphi = ((p \vee q) \wedge (\neg r))$ and truth assignments τ_1 and τ_2,

$$\tau_1 = \begin{cases} p \longmapsto 1 \\ q \longmapsto 1, \\ r \longmapsto 1 \end{cases} \qquad \tau_2 = \begin{cases} p \longmapsto 1 \\ q \longmapsto 0 \\ r \longmapsto 0 \end{cases}$$

$$\varphi(\tau_1) = \big((1 \vee 1) \wedge (\neg 1)\big) = 0, \qquad \varphi(\tau_2) = \big((1 \vee 0) \wedge (\neg 0)\big) = 1$$

Hence, we can conclude that $\tau_1 \vDash \varphi$ and $\tau_2 \nvDash \varphi$.

Satisfiability of Statement. A statement φ is

Nature of φ		Condition	
Satisfiable (SAT)	$\Leftrightarrow$	There exists a truth assignment τ such that τ models φ.	$\exists\tau: \tau \vDash \varphi$
Unsatisfiable ($UNSAT$)	$\Leftrightarrow$	For all possible truth assignments τ, τ does not model φ.	$\forall\tau: \tau \nvDash \varphi$
Valid ($VALID$)	$\Leftrightarrow$	For all possible truth assignments τ, τ models φ.	$\forall\tau: \tau \vDash \varphi$

Table: Conditions for a statement φ to be SAT, $UNSAT$, or $VALID$

Nature of φ	Example
Satisfiable (SAT)	$(p \vee q)$ is SAT but not $VALID$ statement.
Unsatisfiable ($UNSAT$)	$(p \wedge \neg p)$ is a contradiction.
Valid ($VALID$)	$(p \vee \neg p)$ is a tautology.

Table: Examples of SAT, $UNSAT$, and $VALID$ statements

Note the following relationships between the different natures of statements:
- Statement φ is $VALID$ → Statement φ is SAT.
- Statement φ is $VALID$ if and only if $\neg\varphi$ is $UNSAT$. (**Theorem 12.10**)

Relationships between Logical Statements. For any logical statements φ and ψ,

Relationship		Condition
Models / Entails ($\varphi \vDash \psi$)	$\Leftrightarrow$	$\varphi \rightarrow \psi$ is $VALID$.
Semantically Equivalent ($\varphi \equiv \psi$)	$\Leftrightarrow$	$\varphi \leftrightarrow \psi$ is $VALID$.
$\varphi \leftrightarrow \psi$ is $VALID$	$\Leftrightarrow$	$\forall\tau: \varphi(\tau) = \psi(\tau)$

Table: Conditions for relationships between two logical statements

Conjunctive Normal Form ($\boldsymbol{CNF}$). A statement is in Conjunctive Normal Form if that statement comprises of clauses of the form $C_1 \wedge C_2 \wedge \ldots \wedge C_n$.

Statement Type	Form
Statement in CNF	$C_1 \wedge C_2 \wedge \ldots \wedge C_n$
Clause (C_i)	$C_i = l_{i,1} \vee l_{i,2} \vee \ldots \vee l_{i,k}$
Literal ($l_{i,j}$)	$l_{i,j} \in \{p \| p \in PROP\} \cup \{\neg p \| p \in PROP\}$

Table: Forms of the components of a statement in CNF

Theorem 12.9. Every formula φ can be converted into CNF, say φ_{CNF}, by distribution of all clauses.

Checking Satisfiability. Suppose that we have a Knowledge Base KB in CNF. An agent would like to find out if KB entails a statement α, ie. $KB \vDash \alpha$.

$$\begin{aligned} &(KB \vDash \alpha) \\ &\Leftrightarrow (KB \rightarrow \alpha\ is\ VALID) && \text{(from relationships between logical statements)} \\ &\equiv (\neg KB \vee \alpha\ is\ VALID) && \text{(from definition of implication)} \\ &\equiv (\neg(\neg KB \vee \alpha)\ is\ UNSAT) && \text{(from \textbf{Theorem 12.10})} \\ &\equiv (KB \wedge \neg\alpha\ is\ UNSAT) && \text{(this statement is also in } CNF\text{)} \end{aligned}$$

We can check if a statement φ in CNF is $UNSAT$ if we find a contradiction in φ, ie. there exists a clause C_i and another clause $C_j = \neg C_i$. Hence, you should proceed to show that $KB \wedge \neg\alpha\ is\ UNSAT$ through resolution.

Resolution. To resolve two clauses, $(\alpha \vee p)$ and $(\neg p \vee \beta)$,

$$(\alpha \vee p) \wedge (\neg p \vee \beta) \Rightarrow (\alpha \vee \beta)$$

Theorem 12.11. $C_1 \wedge C_2 \rightarrow C_3\ is\ VALID \Rightarrow C_1 \wedge C_2 \leftrightarrow C_1 \wedge C_2 \wedge C_3$.
As a result, $(\alpha \vee p) \wedge (\neg p \vee \beta) \leftrightarrow (\alpha \vee p) \wedge (\neg p \vee \beta) \wedge (\alpha \vee \beta)$.

Resolution Refutation. A resolution refutation of a statement φ given in CNF is a list of clauses $C_1, C_2, \ldots, C_n$ such that either $C_i \in \varphi$ or C_i is derived from C_a, C_b using resolution where $a, b < i$, and the last clause $C_n = \square\ UNSAT$.

Theorem 12.13. A statement φ is $UNSAT$ if and only if there exists a resolution refutation of φ.

$C_1: g \rightarrow h \equiv (\neg g \vee h)$ "All Greeks are human" $C_2: h \rightarrow m \equiv (\neg h \vee m)$ "All humans are mortals" Agent's Knowledge Base can be represented as $KB \equiv (\neg g \vee h) \wedge (\neg h \vee m)$. We want to deduce that $g \rightarrow m \equiv (\neg g \vee m)$ "All Greeks are mortals". We do so by proving that $KB \wedge \neg(\neg g \vee m)$ is $UNSAT$. $KB \wedge \neg(\neg g \vee m) \equiv (\neg g \vee h) \wedge (\neg h \vee m) \wedge (g) \wedge (\neg m)$	$C_1: (\neg g \vee h)$ $C_2: (\neg h \vee m)$ $C_3: (g)$ $C_4: (\neg m)$
Resolve C_1 and C_3:	$\frac{(\neg g \vee h) \wedge (g)}{C_5: (h)}$
Resolve C_2 and C_4:	$\frac{(\neg h \vee m) \wedge (\neg m)}{C_6: (\neg h)}$
Resolve C_5 and C_6:	$\frac{(h) \wedge (\neg h)}{C_7: \square\ UNSAT}$
$KB \vDash (g \rightarrow m)\ is\ VALID$.	

Given a statement φ, checking if φ is SAT or not is NP-complete.
Applying resolution to derive a contradiction can take exponential time.