CS3243 Cheatsheet

Ng Xuan Jun

Miscellaneous:

Arithmetic Series: If $a_n = a_{n-1} + c$, where c is a constant

$$\sum_{i=1}^{n} a_i = a_1 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

Geometric Series: If $a_n = ca_{n-1}$, where $c \neq 1$ is a constant

$$\sum_{i=0}^{n} a_i = a_1 + \cdots + a_n = a_1\left(\frac{c^n - 1}{c - 1}\right)$$

If $0 < c < 1$, then the sum of the infinite geometric series is

$$\sum_{i=1}^{\infty} a_i = \frac{a_1}{1 - c}$$

Environment Properties:

Property	Description
Fully Observable vs Partially Observable	(*States*) Concerned with whether the agent has full knowledge of the search space. Requires handling of uncertainty
Deterministic vs Stochastic	(*Action and State*) Whether given an initial state and action, we can determine the intermediate state (i.e. if we start from the same place and take the same action, will we get to different results every time we run the program) Note that stochastic may still be fully observable (sense all) but there is randomness with the actions (e.g. Monopoly)
Episodic vs Sequential	(*Action*) Whether the previous moves affects the next moves or not Note that we can model an episodic environment into a sequential search space
Discrete vs Continuous	(*Action/State*) Refers to state information, time, percepts, actions Most things in life are continuous and we will try to make them discrete
Single vs Multi-Agent	(*Action*) Do other entities exist within the environment that are themselves agents whose actions directly influence the performance of this agent
Known vs Unknown	(*Problem Knowledge/Rules*) Refers to the knowledge of the agent/designer about the problem Includes performance measure
Static vs Dynamic	(*Environment/State*) Does the environment change while the agent is deciding on an action

Taxonomy of Agents:

Agent Type	Description
Reflex Agent	Uses rules in the form of if-else statements to make decisions
Model-based reflex agent	Internalised model for rules to make decisions.
Goal-based agent	*Given*: State and action representation + definition of goal *Determines*: (1) Sequence of action to reach goal OR (2) Final state that satisfies goal
Utility-based agent	*Given*: State and action representation + definition of utility *Determines*: (1) Sequence of action to maximise utility OR (2) Final state that maximises utility
Learning agents	Agents that learn how to optimise performance

Search Problems:

Path Planning: Path to a goal is necessary and path cost is important

Properties:

Note that if we need to formulate a search problem, we should try to ensure these properties are met

- Fully observable, Deterministic, Discrete, Episodic

This means that we have **complete information**, it is **fully deterministic** and we are able to **PLAN**, i.e. can look ahead at what to do and execute the plan once we have defined it

Uninformed Search:

Correctness Definition:

- ***Completeness***: An algorithm is **complete** if it will find a solution when one exists and correctly report failure when it does not
- ***Optimality***: An algorithm is **optimal** if it finds a solution with the lowest path cost among all solutions (i.e. path cost optimal)

Nodes: We can store the following information in our nodes to improve performance

- ***State***: Each node represents one state
- ***Parent node and action***: Useful for DFS because we want to perform backtracking and assuming a static sequence over the possible actions so that the space complexity can be shrunk to $O(m)$
- ***Depth***: Useful for DLS and IDS to know if the depth limit has reached
- ***Path cost***: To efficiently update the path cost when extending upon the current path

Time and Space Complexity (Graph): $O(|V| + |E|)$

Algorithm	BFS	DFS
Frontier	Queue (FIFO)	Stack (LIFO)
Goal Test	Early Goal Test	Late Goal Test
Complete?	Yes[1]	No[3]
Optimal?	Yes[4]	No
Time (Tree)	$O(b^d)$	$O(b^m)$
Space (Tree)	$O(b^d)$	$O(bm)$
Notes	Early Goal Test is carried out to save on the computational time so we don't have to branch out the final layer which will be b^{d+1} nodes If we want BFS to be optimal, we need the sequence of nodes explored to be monotonically increasing in terms of action cost. Explore **shallowest** nodes first	Not complete under (1) as even if the solution exists, it may infinitely traverse down a path without a solution (Needs both so b and m to be finite) DFS space complexity can be improved to O(m) with backtracking Take note that when we pop out it is in reverse. Explore **deepest** nodes first
	Not concerned with path cost	

Algorithm	UCS	DLS	IDS
Frontier	Priority Queue **Priority**: Path Cost	Stack	Stack
Early / Late Goal Test	Late Goal Test	[redacted]	
Complete?	Yes[1,2]	No	Yes[1]
Optimal?	Yes	No	Yes[4]
Time (Tree)	$O\left(b^{1+\left\lfloor\frac{C^*}{\epsilon}\right\rfloor}\right)$	$O(b^\ell)$	$O(b^d)$
Space (Tree)	$O\left(b^{1+\left\lfloor\frac{C^*}{\epsilon}\right\rfloor}\right)$	$O(b\ell)$	$O(bd)$
Notes	Late Goal Test is required for UCS to be optimal. Because we can only know that a state is optimal after popping it out since still needs to be pushed into the Priority Queue **Explore in order of cost**	Variation of DFS with max depth Space complexity can be improved to O(ℓ) with backtracking	Iterative version of DLS Space complexity can be improved to O(d) with backtracking

1. Complete if b finite and either has a solution **or** m finite
2. Complete if all action costs are $> \epsilon > 0$
3. DFS is incomplete unless the search space is finite – i.e., when b is finite and m is finite
4. Cost optimal if action costs are all identical

Tree Search Algorithm:

- Can revisit nodes

Graph Search Algorithm:

- Does not allow for revisiting of states (unless there is some specified condition like Version 2)

Typical practice:

- Maintain a reached (or visited) ***Hash Table***
- Add reached states
- Only add new nodes to frontier and reached if: State represented by node not previously reached

Algorithm:

Assume that Version 1 is used unless stated otherwise

Version 1: Ensures that nodes are never revisited (Omits all redundant paths and may omit optimal path)

Version 2: More relaxed constraint on paths, also considers paths with lower path cost (Includes **purple** part of the algorithm). Non redundant paths are never skipped

Version 3: Only adds a node to reached when it is popped. (Add in the **blue** part and omit the **green** part. For Version 1 and 2, include the **green** and omit the **blue**)

Informed Search:

- Make use of domain knowledge to estimate the cost to the goal
- Make use of a heuristic function, $h(n)$ to estimate the cost of going from the current node to the nearest goal state. However, it should be an efficient function to compute, time complexity should not be high

Heuristic Theorems:

Admissible: $h(n)$ is admissible if $\forall n, h(n) \leq h^*(n)$

- $h(n)$ never overestimates the true cost
- By the time we visit a path to a goal, P, all paths with actual costs less than P must be searched
- Paths not ending at a goal are never over-estimated. (At non-goal, n, $f(n) = g(n) + h(n) \leq g(n) + h^*(n)$)
- Paths ending at a goal are exact, $h(g) = 0$. Only when we pop off the goal node can we be sure that the optimal path to the goal node is found. If we pop off any other node, we cannot say that it is the optimal path to that node since there is no ordering

Consistent: $h(n)$ is consistent fi $\forall n$, and successor $n, n', h(n) \leq cost(n, a, n') + h(n')$

- Ensures that f costs are monotonically increasing along a path
- Note that for this, all paths are in increasing order and once we get to a state, we must have the optimal path to that node.
- Consistency $\Rightarrow$ Admissibility

Algorithm	Uniform-Cost Search (UCS)	Greedy Best-First Search	A* Search
Data Structure (Frontier)	Priority Queue **Priority**: Evaluation Function	Priority Queue **Priority**: Evaluation Function	Priority Queue **Priority**: Evaluation Function
Early / Late Goal Test	Late Goal Test	Late Goal Test	Late Goal Test
Evaluation Function	$f(n) = g(n)$	$f(n) = h(n)$	$f(n) = h(n) + g(n)$
Complete under Tree?	Yes[1,2]	No	Yes[1,2]
Optimal under Tree?	Yes	No	Yes[3]
Complete under Graph?	Yes[1,2]	Yes[1]	Yes[1,2]
Optimal under Graph?	Yes (Version 2, 3)	No	Yes[3] (Version 2) Yes[4] (Version 2 & 3)
Notes	It is the worst case if we are considering admissible heuristic. Because we are just taking $h(n) = 0$	It does not exploit the information of cost of path already taken. **Might get us to a solution faster** but may not be the optimal one. May explore lesser paths if we are not concerned with optimality	**Theorem**: If $h(n)$ is admissible, tree search is optimal **Theorem**: If $h(n)$ is consistent, graph search is optimal

1. Complete if b finite and either has a solution or m finite
2. Complete if all action costs are $> \epsilon > 0$
3. If the heuristic $h(n)$ is admissible
4. If the heuristic $h(n)$ is consistent

Admissibility under Max/Min

Cases	**Admissible**/**Inadmissible**/**Indeterminate**
max(Admissible, Admissible)	**Admissible**
max(Admissible, Inadmissible)	**Inadmissible**
max(Inadmissible, Inadmissible)	**Inadmissible**
min(Admissible, Admissible)	**Admissible**
min(Admissible, Inadmissible)	**Admissible**
min(Inadmissible, Inadmissible)	**Indeterminate**

Efficiency of Heuristic:

Dominance: If $h_1(n) \geq h_2(n)$ for all n, then h_1 dominates h_2

- If h_1 is also admissible, (also means that h_2 is admissible). Then it means that h_1 is closer to h^* and therefore it should be more efficient.
- h_1 will be more efficient because the set of paths that it needs to check is lesser

Effective Branching Factor:

- Makes use of empirical results of: N nodes explored and solution path is at depth d
- We try to approximate the amount of times we need to branch out given the number of nodes explored and the depth of the solution.
- The branching factor allows us to know an approximate number of nodes that is expanded by the algorithm as if we branch out lesser, it means that we have more efficient algorithm. Because if they branch out lesser, it means that even if we go deeper, the number of states to search is lesser.

Solve for b^* using the solve:

$$N + 1 = (b^*)^0 + (b^*)^1 + \cdots + (b^*)^d$$

$$N + 1 = \frac{(b^*)^{d+1} - 1}{(b^*) - 1}$$

Proof that Tree Search Implementation for Greedy Best-First Search is incomplete

- Just need to make sure that we have 2 nodes with the same $h(n)$ and we can travel bidirectional

Proof that Graph Search Implementation for Greedy Best-First Search is complete

- Assuming that we have a finite search space, a graph search will not revisit states and therefore, would visit all states within the state space. Therefore, it will either report a solution or failure

Example that Tree/Graph Search Greedy Best-First Search is not optimal:

- Idea is just to give estimation for states with higher cost to be lower than that of those with lower cost and we will be able to get a non-optimal solution

Proof that tree search for A* Search is optimal when Admissible Heuristic is utilised:

- Let s be the initial state, n be an intermediate state along the optimal path, t be a suboptimal goal state and t^* be the goal along the optimal path
- Optimal solution means that n must be expanded before t since we need to travel to n first before t
- ***Proof by Contradiction***:
  - Assume that a suboptimal solution is found (i.e. t is expanded before n which means $f(t) \leq f(n)$)
  - Assuming that both t and n are in the frontier, we need $f(t) < f(n)$ for it to be popped off first
  - However, since t is not on the optimal path but t^* is then we must have: $f(t) > f(t^*)$
  - $f(t) > g(t^*)$ since $h(t^*) = 0$ for goal node $\Rightarrow f(t) > g(n) + p(n, t^*)$ where $p(n, t^*)$ is cost of going from n to $t^* \Rightarrow f(t) > g(n) + h(n)$ by Admissibility $\Rightarrow f(t) > f(n)$ which is a contradiction
  - Note that we do not consider $f(t) = f(n)$ since this would mean that $f(t)$ is equally optimal

Proof that if h is consistent, it is also admissible given $h(t) = 0$ when t is a goal node

- ***Main Idea***: Proof by Induction on $k(n)$ which is the number of actions required to reach the goal from a node n to the goal node t
- ***Proof by Induction***:
  - **Base Case**: $(k = 1)$ The node is one step from t.
    Since the heuristic is consistent, $h(n) \leq c(n, a, t) + h(t)$ and since $h(t) = 0$, we have $h(n) \leq c(n, a, t) = h^*(n)$ (Therefore, h is admissible)
  - **Induction Hypothesis**: Suppose the assumption holds for every node that is $k - 1$ actions away from t, where the least-actions optimal path from n to t has $k > 1$ steps
  - **Induction Step**: If we look at node n which is k actions away from t and considering its optimal path
    Optimal path from n to t is: $n \rightarrow n_1 \rightarrow \cdots \rightarrow n_{k-1} \rightarrow t$
    Since h is consistent: $h(n) \leq c(n, a, n_1) + h(n_1)$
    Since n_1 is on the least-cost path from n to t, we must have the path $n_1 \rightarrow \cdots \rightarrow n_{k-1} \rightarrow t$ is a minimal cost path from n_1 to t as well. By the induction hypothesis, $h(n_1) \leq h^*(n_1)$
    $$\Rightarrow h(n) \leq c(n, a, n_1) + h^*(n_1)$$
    Note that $h^*(n_1)$ is the cost of the optimal path from n_1 to t; we have that the cost of the optimal path from n to t, $h^*(n) = c(n, a, n_1) + h^*(n_1)$. Therefore, the heuristic is admissible

Local Search:

- When we are only concerned with what the goal state is. Not concerned with how to get there

Advantages:

- Can just store the current and immediate successor states:

Space Complexity: $O(b)$ – Can be reduced to O(1) if successors may be processed one at a time

- Applicable to very large or infinite search spaces

Problem Formulation for Local Search:

States: Each state will be a **complete** assignment

Initial State: Probably just some randomly initialised **complete** assignment.

Next State (Actions): Perturbs the current state by 1 move but still a **complete** assignment. We need to state how we will find the next state.

Stopping criteria: Stop when the optima is found or after a specified number of iterations n, whereby the value of the current state is better than all its neighbours or next states. We can use hill-climbing with random restarts as well.

Stopping Condition: If val(next_state) < val(current_state), then set next_state as current_state and repeat the process . Else if val (next_state) ≥ val(current_state) then terminate the process, and return current state as the solution.

If sideway: Continue need val(next_state) ≤ val(current_state), terminate need val(next_state) > val(current_state).

Completeness: Incomplete because cannot report no goal / terminates too early even when the goal exists

Optimality: Don't need to talk about optimality since the path is inconsequential and we just want a solution.

Probabilistic Complete: The probability of not finding the answer tends to zero as more work is done. For probabilistic algorithm, so long as the probability is positive, we can say that we will eventually find a solution.

Hill Climbing (Steepest Ascent) Algorithm

Explanation:

- Start with a random initial state
- Only store the current state
- In each iteration, find a successor that improves on the current state
  - Requires actions and transition to determine successor
  - Requires value; a way to value each state e.g. $f(n) = -h(n)$
- If none exists, return the current state as the best option
  - Note that the algorithm can fail; return a non-goal state

Issues:

- May get stuck at Local Maxima, Shoulder or Plateau, Ridge (Sequence of local maxima)

Variants:

Variant	Details
Stochastic hill climbing	Changes the highest_valued_successor(…) Chooses randomly among states with values better than the current one (*Not choosing the highest one but any that is better*) May take longer to find a solution but sometimes leads to better solutions
First-choice hill climbing	Changes the highest_valued_successor(…) Handles high b by randomly generating successors until one with better value than current is found (rather than generating all possible successors) Space complexity will be O(1) since don't have to keep track of all of them
Sideways move	Replaces ≤ with <; allows for continuation when value(neighbour) == value(current) Can help to traverse shoulders/plateaus
Random-Restart hill climbing	Adds an outer loop which randomly picks a new starting state Keeps attempting random restarts until a solution is found Note that this is the default one that we should be using

Local Beam Search:

Stores k states instead of 1 now

- Hill Climbing stores only the current state but beam stores k states

Algorithm:

Begin with k random initial states Each iteration generate all successors of the current k states Repeat with the best k among ALL generated successors unless goal is found

Advantages:

- Better than k parallel random restarts because we just take the best k among all successors and not just the best from each set of k sets
- No longer a constraint to have 1 best successor from each set of random initial state

Stochastic Beam Search

- Original variant may get stuck in a local cluster
- Adopt stochastic strategy similar to stochastic hill climbing to increase state diversity

Constraint Satisfaction Problems

Problem Formulation:

State Representation:

Variables: $X = \{x_1, \cdots, x_n\}$

Domains: $D = \{d_1, \cdots, d_n\}$ such that each x_i has a domain d_i

Initial State: All variables unassigned

Intermediate State: Partial Assignment

Goal Test:

Constraints: $C = \{c_1, \cdots, c_m\}$

- Each c_i corresponds to a requirement on some subset of X_i

Actions, Transition: Assignment of variables (within domain) to variables

Costs: Cost are not utilised

Objective: Is a complete and consistent assignment if we find a legal assignment

- Find a legal assignment $(y_1, \cdots, y_n): y_i \in d_i \ \ \forall i \in [n]$
- ***Complete***: All variables are assigned values
- ***Consistent***: All constraints C satisfied

Backtracking Algorithm for CSPs

1. Determine the variable to assign to
2. Determine the value to assign
3. Trying to determine if the chosen assignment will lead to a terminal state
4. Continues recursively as long as the assignment is viable

SELECT-UNASSINGED-VARIABLE

Fail-First: Because every variable must be assigned to arrive at a solution and we need to look at all variables. Therefore, we should just try to fail as much as possible so that we don't have to look at so many solutions

Minimum-Remaining-Values (MRV) Heuristic

Choose the variable with the fewest legal values (most constrained variable / smallest consistent domain size among unassigned variables)

- Usually performs better than static or random ordering

General Idea: Place larger subtrees closer to the root (so that any invalid states prunes a larger subtree) and we can eliminate larger subtrees earlier

Degree Heuristic:

If the MRV requires tie breaking, we can make use of this. (MRV -> Degree -> Random)

Pick the variable with most constraints relative to unassigned variables

General Idea: By selecting a variable that restricts the most number of other variables, we can reduce the branching by b since those variables will need to be constrained by b values. We try to branch less so that when we backtrack, we prune off a bigger tree faster

ORDER-DOMAIN-VALUES

Fail Last: Only one solution required and we may not have to look at some values. Therefore, we just want to choose the value that can maximise our chances of succeeding

Least-Constraining-Value Heuristic

Choose the value that rules out the fewest choices

- Given assignment of value v to variable x' (which is the variable chosen)
- Determine unassigned variables $U = \{x_a, x_b, \cdots\}$ that share a constraint with x'
- Pick v that maximises sum of consistent domain sizes of variables in U

General Idea: Avoid failure (avoid empty domains). Try to Leave maximum flexibility for subsequent assignments

INFERENCE

Forward Checking:

- Track remaining legal values for unassigned variables
- Terminate search when any variable has no legal values

Issue: Forward checking propagates information from assigned to unassigned variables but does not provide early detection for all failures

Constraint Propagation:

Inference step to ensure local consistency of all variables

- Traverse constraint graph to ensure variable at each node is consistent (Eliminate all values in variable's domain that are not consistent with linked constraints

Node-consistent: Domain of the single variable is consistent with unary constraints

- Can be done as a pre-processing step (just once at the start of the program) and for each variable we just eliminate domain values that are inconsistent with the unary constraints

Arc-consistent: Domain of the single variable is consistent with binary constraints

- For each variable: Eliminate domain values inconsistent with binary constraints. Variable domain value must have partnering domain value in other variable that will satisfy the constraint

Definition: X_i is arc-consistent with X_j (i.e. arc (X_i, X_j) is consistent) iff for every value $x \in D_i$ there exists some value $y \in D_j$ that satisfies the binary constraint on the arc (X_i, X_j)

AC-3 Algorithm

1. Initialise a queue containing all arcs (both directions for each binary constraint)
2. Each time a variable $X_i's$ domain is updated, add all arcs corresponding to binary constraints with other variable (not X_i) as target (except the one that just caused the revision)
3. Eliminating domain values of the target variable X_i relative to the other variable X_j in the binary constraint

Time Complexity: $O(n^2 d^3)$

- CSPs have at most $2 \times nC_2$ or $O(n^2)$ directed arcs (given n variables)
- Each arc (X_i, X_j) can be inserted at most d times because X_i has at most d values to delete (given domain size d) – Checking consistency of arc (REVISE function) takes $O(d^2)$

Adversarial Search:

Formulation

- ***State***: As per the general formulation
- ***TO-MOVE(s)***: Returns p, which is the player to move in state s
- ***ACTIONS(s)***: Legal moves in state s
- ***RESULT(s, a)***: The transition model; returns resultant state when taking action a at state s
- ***IS-TERMINAL (s)***: Returns TRUE when game is over and FALSE otherwise. States where game has ended are called terminal states
- ***UTILITY(s, p)***: Defines the final numeric value to player p when the game ends in terminal state s

Assumptions on environment:

- 2 Player, Deterministic, Turn-Taking
- Zero Sum Game: Loser for every winner

Properties of Minimax:

Completeness: Yes (If there is a finite game tree)

Optimal: Yes

- Note that this is assuming optimal gameplay from both players. We do not consider suboptimal plays from the opponent because assuming optimality will give us a lower bound.

Time Complexity: $O(b^m)$

Space Complexity: $O(bm)$

$\alpha - \beta$ Pruning: Helps to remove large parts of the search tree that are redundant.

Idea:

α – Bounds MAX's value (the highest utility that we have seen thus far)

- Initialised as $-\infty$. Will be updated when we see a higher value than the current value.

β – Bounds MIN's value (the lowest utility that we have seen thus far)

- Initialised as $+\infty$. Will be updated when we see a lower value than the current value.

Main Condition: Prune if $\alpha \geq \beta$

Pruning Rules:

NOTE that pruning does not affect the final outcome.

1. Given a MIN node n, stop searching below n if: Some MAX ancestor i (of n) has an α value that more than β value of n.
2. **Only continue if $\beta(n) > \alpha(i)$**
3. **Prune if $\beta(n) \leq \alpha(i)$**

We know that once the current β value is smaller than α, then the MAX player will not choose this anymore since we will continue checking if the MIN player's values are going to be smaller and therefore, there is no way the MAX player will choose this path.

1. Given a MAX node n, stop searching below n if: Some MIN ancestor i (of n) has an β value that less than α value of n.
2. **Only continue if $\alpha(n) < \beta(i)$**
3. **Prune if** $\alpha(n) \geq \beta(i)$

We know that once the current α value is more than β, then the MIN player will not choose this anymore since we will continue checking if the MAX player's values are going to be bigger and therefore, there is no way the MIN player will choose this path.

Note: If the question is only bounded by $\leq A \leq$ then we will still follow this rule and it will result in us only account for integer values.

Time Complexity Improvements for $\alpha - \beta$ pruning:

- Without Move Ordering: $O(b^m)$
- With Perfect Move Ordering: $O\left(b^{\frac{m}{2}}\right)$. Good pruning can let us search twice as deep
- Random Ordering: $O\left(b^{\frac{3m}{4}}\right)$ for $b < 1000$

Heuristic Minimax: Helps to resolve the issue when the depth of the tree is too deep and evaluation of needs to occur at terminal states. (Backward induction will only works backwards from terminal states)

- Cutoff test – Depth Limited (DLS) / IDS

Run MINIMAX until depth d, then use the evaluation function to choose nodes.

- Evaluation Function – Estimates expected utility of state

Note that the deeper we go, the better our estimation, but the longer the algorithm needs to run

Logical Agents:

Entailment: Means that one thing follows from the other

$\alpha \vDash \beta$ or equivalently $M(\alpha) \subseteq M(\beta)$

- If it is a subset then it must mean that it infers that the bigger set must be true since all the values in the subset "satisfies" the bigger set.
- $KB \vDash \alpha$ means that whenever KB is true, α is also true
- Entailment works because when the KB is a subset of the entailed statement, we know that when the KB is true, α must also be true since all conditions that makes KB true must also make α a true statement.

$KB \vdash_A \alpha$: Sentence α is derived (i.e. inferred) from KB by inference algo, A

Necessary Condition: $A \Rightarrow B$. B happens if A happens

Sufficient Condition: $B \Rightarrow A$. B happens only if A happens

Soundness:

A is sounds if $KB \vdash_A \alpha$ implies $KB \vDash \alpha$

- For all sentences inferred from the KB by A, S
- The KB will entail each α in S

Completeness:

A is complete if $KB \vDash \alpha$ implies $KB \vdash_A \ \alpha$

- If KB entails a sentence (any sentence describing a superset of the KB)
- A can infer that sentence

Validity: Sentence is true for ALL possible truth value assignments

Satisfiable: Sentence is true for SOME truth value assignments

Unsatisfiable: Sentence is true for NO truth value assignments

Unsatisfiable Lemma: If there exists a cycle that connects a node x with a node $\neg x$ then the CNF formula is unsatisfiable.

Inference Rules:

And Elimination (AE): $a \wedge b \vDash a; a \wedge b \vDash b$

Modus Ponens (MP): $a \wedge (a \Rightarrow b) \vDash b$

Modus Tollens (MT): $\neg b \wedge (a \Rightarrow b) \vDash \neg a$

Logical Equivalences: $(a \vee b) \vDash \neg(\neg a \wedge \neg b)$

Contraposition: $(a \rightarrow b) \vDash \neg b \rightarrow \neg a$

Syllogism: $(a \rightarrow b) \wedge (b \rightarrow c) \vDash a \rightarrow c$

Propositional Rules:

Exclusive OR: $(a \vee b) \wedge \neg(a \wedge b)$

Distributive Law: $(a \wedge b) \vee (c \wedge d) \equiv (a \wedge c) \vee (a \wedge d) \vee (b \wedge c) \vee (b \wedge d)$

Splitting implication: $\big((a \vee b) \rightarrow c\big) \equiv (a \rightarrow c) \wedge (b \rightarrow c)$

Biconditional (XNOR): $(a \leftrightarrow b) \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

Cardinality Rules: Suppose $n =$ Number of variables we have, $k =$ Exact number we need

At least k:

DNF: We will wrap the k-way AND around a big OR

$$(A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$$

CNF: Check downwards to see which corresponds to k. Note that for all the $i - way$ combination we will then wrap in a giant $\wedge$

- Pairwise OR: $(n-1) = k$
- Threeway OR: $(n-2) = k$

Example: $(A \vee B) \wedge (A \vee C) \wedge (B \vee C) \ \ (n = 3, k = 2)$

At most k:

KB Way: Just write the implication on the left with k of the values true, means that the other remaining values are False.

$$\big((A \wedge B) \rightarrow \neg C\big) \wedge \big((A \wedge C) \rightarrow \neg B\big) \wedge \big((B \wedge C) \rightarrow \neg A\big)$$

CNF: Pairwise negation of $k+1$ ways

Example: (n=3, k=2) $(\neg A \vee \neg B \vee \neg C)$

- Since we have $k = 2$, we look at 3-way OR

Exactly k:

DNF: Choose all the combination of n variables that makes only k True and wrap in OR

$$(A \wedge B \wedge \neg C) \vee (A \wedge \neg B \wedge C) \vee (\neg A \wedge B \wedge C)$$

CNF: Do conjunction of at least k and at most k

Example:

$$(A \vee B) \wedge (A \vee C) \wedge (B \vee C) \wedge (\neg A \vee \neg B \vee \neg C)$$

Propositional Logic Laws:

De Morgan's	$\neg(p \vee q) \equiv \neg p \wedge \neg q$	$\neg(p \wedge q) \equiv \neg p \vee \neg q$
Idempotent	$p \vee p \equiv p$	$p \wedge p \equiv p$
Associative	$(p \vee q) \vee r \equiv p \vee (q \vee r)$	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
Commutative	$p \vee q \equiv q \vee p$	$p \wedge q \equiv q \wedge p$
Distributive	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
Identity	$p \vee False \equiv p$	$p \wedge True \equiv p$
Domination	$p \wedge False \equiv False$	$p \vee True \equiv True$
Double negation	$\neg\neg p \equiv p$	
Complement	$p \wedge \neg p \equiv False \wedge \neg True \equiv False$	$p \vee \neg p \equiv True \vee \neg False \equiv True$
Absorption	$p \vee (p \wedge q) \equiv p$	$p \wedge (p \vee q) \equiv p$
Conditional Identities	$p \Rightarrow q \equiv \neg p \vee q$	$p \Leftrightarrow p \equiv (p \Rightarrow q) \wedge (q \Rightarrow p)$

Resolution:

CNF: Conjunction of Disjunctive statements

$$R_1 \wedge R_2 \wedge \cdots \wedge R_n$$

If a literal, x, appears in R_i and its negation, $\neg x$ appears in R_j, where $R_i, R_j \in KB$. Then x and $\neg x$ can be removed from both sides

$$(x_1 \vee \cdots \vee x_m \vee x) \wedge (y_1 \vee \cdots \vee y_k \vee \neg x) \rightarrow (x_1 \vee \cdots x_m \vee y_1 \vee \cdots \vee y_k)$$

Note that we can only remove one variable at a time.

Under Propositional Logic: Resolution is **Sound** and **Complete**

Conversion to CNF

1. $\alpha \Leftrightarrow \beta$ change to $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
2. $\alpha \Rightarrow \beta$ change to $\neg\alpha \vee \beta$
3. Expand $\neg$ using De Morgan's and Double Negation

$\neg(\alpha \vee \beta)$ change to $\neg\alpha \wedge \neg\beta$

$\neg(\alpha \wedge \beta)$ change to $\neg\alpha \vee \neg\beta$

$\neg(\neg\alpha)$ chage to α

4. $\big(\alpha \vee (\beta \wedge \gamma)\big)$ change to $(\alpha \vee \beta) \wedge (\alpha \vee \gamma)$

Algorithm:

1. Make a clause list (Convert the KB to CNF using conversion rules)

Copy of KB specified in CNF including the negation of the query, $\neg\alpha$

2. Repeatedly resolve 2 clauses from clause list
3. Keep doing this till empty clause is found (can infer α) or no more resolutions possible (cannot infer α)

Conversion into 3CNF from N-CNF: We can add in additional variables to our domain and to the clauses such that we can split it into a 3-CNF formula. This is done with the idea of the resolution formula whereby if we have a literal that is the negation of each other, we can remove those and combine the conjunction.

Soundness Proof:

Suppose that we obtain α from KB by running a sequence of resolution operations. Our proof is by induction on the number of resolutions we executed before obtaining α.

Let $\vec{x} = (x_1, \cdots, x_n)$. Suppose that KB is in CNF form.

For the first resolution step we have that there exists two OR clauses in KB of the form $P(\vec{x}) \vee x$ and $Q(\vec{x}) \vee \neg x$. Applying resolution, we get $P(\vec{x}) \vee Q(\vec{x})$.

Note that if $\vec{t} = \{True, False\}^n$ is a satisfying truth assignment for KB then it must be that both $P(\vec{x}) \vee x$ and $Q(\vec{x}) \vee \neg x$ are true. In particular if $x = True$ under $\vec{t}$, then $Q(\vec{t}) = True$; if $x = False$ then $P(\vec{t}) = True$; in either case, the expression $P(\vec{t}) \vee Q(\vec{t}) = True$.

Since this is true for any resolvent reachable after 1 resolution step (this is an arbitrary one that we picked), we have shown the case for resolvents that are reached after one step.

For the inductive step, suppose that any resolvent q that is achievable after r resolution steps satisfies $M(KB) \subseteq M(q)$; we show the claim holds for $r+1$. However, this is simply a repetition of the proof for the one-step case, with KB being KB^r: the set of all resolvents reachable from KB after r resolution steps.

Since each resolvent is implied by the inference engine and it is always True, it must mean that it is sound as well. Note that if $\emptyset$ is found, it just means that $(KB \wedge \neg\alpha)$ is unsatisfiable and it is not implied as well. Therefore, the resolution algorithm is sound.

Probability

Conditional Probability: $P(A, B, C) = P(A|B, C) \times P(B|C) \times P(C)$

Conditioned Conditional Probability: $P(A, B|C) = P(A|B, C) \times P(B|C)$

- Note that this is just basically closing an eye on C and adding it back as a conditional afterwards

Law of Total Probability: $P(X) = \sum^n P(X|A_i)P(A_i)$

Conditional LOTP: $P(A|C) = \sum^n P(A|B_i, C) \times P(B_i|C)$

Bayes Theorem: $P(A|B, C) = \frac{P(A, B|C)}{P(B|C)} = \frac{P(B|A, C) \cdot P(A|C)}{P(B|C)}$

- Can think of this as conditioning on P(B|C) where |C is just an additional condition that B is conditioned on.

Chain Rule: $P(A, B, C, D) = P(D|C, B, A) \cdot P(C|B, A) \cdot P(B|A) \cdot P(A)$

Note that Conditional Independence is not the same as Independence

Conditional Independence: Events A, B are conditionally independent given C if and only if:

$$P(A|B, C) = P(A|C), \qquad P(C) > 0$$
$$P(A, B|C) = P(A|C)P(B|C)$$

$\Rightarrow$ Knowing C makes B useless

Conditional Dependence: $P(A|B, C) \neq P(A|C)$

Common Tricks:

1. Sum over nuisance variables (***Law of Total Probability***)
2. Look at subset of the Bayes Net (***Markov Blanket***)
3. If we need to jump across nodes, can use ***Marginalisation***

Things to consider for Conditional Independence & Markov Blanket:

Parents(x), Children(x), Parents(Children(x))

Lemma: Given two random Boolean variables, A and B, if $P(A|B) = 0$ and $P(A|\neg B) = 1$, then $P(A) = 1 - P(B)$. $A \equiv \neg B$

Bayesian Network:

Characteristics:

- Directed Acylic Graph (DAG) cause of chain rule and we assume conditional independence
- Edge links dependent variables
- **Max No. of Edges**: $n(n-1)/2$

Implications:

If $P(A|B, C) = P(B|A, C)$, then $P(A|C) = P(B|C)$

Only if we assume non-zero probabilities. If $P(A, B, C)$ then this could be False.