PYQ 2022

Why do we require unification ?

Unification is required in first-order logic (FOL) and propositional logic (PL) to enable the resolution and inference of logical formulas. In FOL and PL, logical formulas are represented as expressions made up of symbols, such as variables, constants, predicates, and logical connectives.

The process of unification involves finding a substitution of values for the variables in a set of expressions such that they can be made identical. This substitution allows us to compare and combine expressions in a systematic way. Unification is a key component of resolution refutation, which is a technique for proving the validity of logical formulas by reducing them to a contradiction.

In FOL, unification is used to unify variable symbols and to instantiate quantified variables. Quantifiers are logical symbols that bind variables and indicate the scope of the variable. For example, the formula "$\forall x\ (P(x) \rightarrow Q(x))$" states that for all values of x, if P(x) is true, then Q(x) is also true. To evaluate such a formula, we need to instantiate the variable x with a specific value, which requires unification.

In PL, unification is used to unify propositional variables, such as P, Q, and R, and to instantiate truth values. For example, the formula "$P \rightarrow (Q \land \neg R)$" states that if P is true, then both Q and ¬R must be true. To evaluate such a formula, we need to instantiate the propositional variables with truth values, which requires unification.

In summary, unification is required in FOL and PL to enable the comparison and combination of logical expressions, and to instantiate variables and truth values. Unification plays a crucial role in enabling the resolution and inference of logical formulas, and is therefore an essential component of these logical systems.

What is skolemisation?

Skolemisation is a technique used in first-order logic (FOL) to eliminate existential quantifiers from a formula by introducing Skolem functions or Skolem constants.

An existential quantifier (∃) in FOL asserts the existence of a particular object or entity that satisfies the quantified formula. Skolemisation replaces this existential quantifier with a Skolem function or Skolem constant, which generates a new term that represents the previously existentially quantified variable.

For example, consider the following formula in FOL:

$\exists x\ P(x)$

This formula asserts the existence of an object x that satisfies the predicate P. To Skolemise this formula, we can introduce a Skolem function f that generates a new term y, which is a function of the remaining variables in the formula:

$P(f(y))$

The Skolem function f takes the place of the existentially quantified variable x, and the term y represents the object that satisfies the predicate P. The Skolemised formula states that there exists a term y that satisfies the predicate P, but it does not explicitly state the existence of an object x.

Skolemisation can also be used to eliminate multiple existential quantifiers from a formula. For example, consider the following formula:

$\exists x \; \exists y \; P(x, y)$

We can Skolemise this formula by introducing a Skolem function f that generates a new term z, which is a function of the remaining variables in the formula:

$P(f(x), f(y))$

The Skolem function f takes the place of both existentially quantified variables x and y, and the term z represents the object that satisfies the predicate P.

Skolemisation is a useful technique in FOL because it allows us to eliminate existential quantifiers and simplify the logical formula. Skolemised formulas are often easier to work with and can be used in resolution refutation and other logical inference techniques.

Why resolution refutation always terminates by finding a contradiction or by failing to find a contradiction ?

Resolution refutation is a proof technique used to show the validity or satisfiability of a formula in first-order logic (FOL). The technique is based on a process of resolving clauses, where two clauses are combined to produce a new clause that is more general than either of the original clauses.

Resolution refutation always terminates by finding a contradiction or by failing to find a contradiction because of the following reasons:

Soundness: Resolution refutation is a sound proof technique, meaning that if it derives the empty clause, then the original formula is unsatisfiable. The empty clause represents a contradiction, which means that the formula cannot be satisfied. If the empty clause is not derived, then the proof process continues until it reaches a point where no further resolutions are possible. At this point, if the original formula is satisfiable, then there must be a model that satisfies it.

Completeness: Resolution refutation is a complete proof technique for propositional logic, meaning that if a formula is unsatisfiable, then the proof process will eventually derive the empty clause. However, it is not complete for first-order logic, since there are formulas that are valid but not provable by resolution refutation. Nevertheless, the technique can still be used to show unsatisfiability in many cases.

Finite search space: Resolution refutation involves a finite search space, which means that the proof process will always terminate in a finite amount of time. This is because the number of possible resolutions is limited by the number of clauses in the formula and the number of literals in each clause. Therefore, the search space is bounded and can be explored completely.

In summary, resolution refutation always terminates by finding a contradiction or by failing to find a contradiction because it is a sound and complete proof technique for propositional logic and has a finite search space. For first-order logic, it is not complete, but it can still be used to show unsatisfiability in many cases.

What is a refutation tree ?

A refutation tree is a tree-like structure that represents the proof of unsatisfiability (i.e., contradiction) of a set of logical formulas using resolution refutation.

The tree has a root node that represents the negation of the conclusion of the argument or the conjunction of the negated formulas in the set. The children of the root node represent the clauses in the negated formula, and the children of each clause node represent the resolvents that are derived by resolving the clause with other clauses in the tree. The tree grows by adding new nodes that represent resolvents until a contradiction is derived, or no further resolvents can be generated.

A refutation tree is constructed by applying the resolution rule repeatedly to pairs of clauses in the set until a contradiction is derived. The tree represents a proof of unsatisfiability if and only if a contradiction is derived at some point in the tree.

Control strategies for performing resolution refutation in FOL.

Resolution refutation is a proof technique used to show the validity or satisfiability of a formula in first-order logic (FOL). The general idea is to transform the original formula into a set of clauses and then use a process of resolution to derive the empty clause, which indicates that the formula is unsatisfiable.

There are several control strategies that can be used to perform resolution refutation in FOL. Some of the commonly used ones are:

Unit resolution: This strategy selects a unit clause (a clause with only one literal) and applies it to all other clauses to derive a new set of clauses. This is repeated until the empty clause is derived.

Pure literal elimination: This strategy selects a literal that only appears with a particular polarity in the formula and removes all clauses containing that literal or its negation.

Binary resolution: This strategy selects two clauses containing complementary literals (i.e., one clause has a positive literal and the other has its negation) and resolves them to derive a new clause.

Input ordering: This strategy controls the order in which the input clauses are processed. This can be done based on various criteria such as the size of the clauses, the frequency of occurrence of literals, or the number of variables in the clauses.

Branching heuristics: This strategy is used when the refutation process reaches a point where there is more than one resolution step to be performed. It selects one of the possible resolutions based on a heuristic that aims to minimize the search space and increase the chances of finding a refutation quickly.

What is a fuzzy set ?

A fuzzy set is a type of set in which each member has a degree of membership, or a degree of belonging, to the set. Unlike classical sets, where each element is either a member or not a member of the set, fuzzy sets allow for degrees of membership that lie between 0 (not a member) and 1 (fully a member).

Fuzzy sets are used in fuzzy logic, which is a type of logic that allows for approximate reasoning and decision-making based on degrees of truth or probability. Fuzzy sets can be used to model situations where the boundaries between categories or classes are unclear or where there is uncertainty or imprecision in the data.

For example, consider the concept of "tallness". In a classical set, someone is either tall or not tall. But in a fuzzy set, someone might have a degree of membership in the "tall" set that ranges from 0 to 1, depending on how tall they are. Someone who is 7 feet tall might have a high degree of membership, say 0.9, while someone who is 5'6" might have a lower degree of membership, say 0.3.

Fuzzy sets have applications in many fields, including artificial intelligence, control systems, decision-making, and pattern recognition.

What is classification ?

Classification is a type of supervised learning in machine learning where the goal is to classify or categorize input data into a predefined set of categories or labels. It involves training a model on a labeled dataset to learn the relationship between the input features and their corresponding labels. Once the model is trained, it can be used to predict the label of new, unseen data.

In classification, the input data is represented by a set of features or attributes, and the output is a discrete label or category. The task is to learn a function that maps the input features to the output label. This function is typically represented by a classification algorithm or model, which can be trained using various machine learning techniques such as decision trees, logistic regression, support vector machines, and neural networks.

Classification models are widely used in many applications, such as image recognition, natural language processing, fraud detection, and medical diagnosis. For example, in image recognition, a classification model can be trained to identify objects in an image and assign them to different categories, such as "cat", "dog", "car", or "tree". In natural language processing, a classification model can be used to classify text documents into different categories, such as "spam" or "ham" in email filtering.

The performance of a classification model is typically evaluated using metrics such as accuracy, precision, recall, and F1-score. These metrics measure how well the model can correctly classify the input data into their respective categories.

How can this be achieved by genetic algorithms ?

Genetic algorithms can be used to optimize the performance of classification models by finding the best combination of model hyperparameters and feature selection. Hyperparameters are settings that control the behavior of the model, such as the learning rate, number of hidden layers, and number of neurons in each layer. Feature selection is the process of selecting the most important features or attributes from the input data.

Genetic algorithms work by creating a population of potential solutions, each of which represents a possible combination of hyperparameters and feature selection. The algorithm then evaluates each potential solution by applying it to a training set of data and calculating the accuracy of the resulting classification model. The solutions are then ranked based on their accuracy, and the best solutions are selected to form a new population.

The new population is created through a process of crossover and mutation. Crossover involves combining two or more potential solutions to create new solutions, while mutation involves randomly changing the hyperparameters or feature selection of a solution to create a new solution. The new population is then evaluated again, and the process of selection, crossover, and mutation is repeated until a satisfactory solution is found.

By using a genetic algorithm to search the space of possible hyperparameters and feature selection, it is possible to find a combination that leads to a highly accurate classification model. However, this approach may require a large amount of data to train the classification model and may be computationally expensive. Therefore, it is important to carefully choose the hyperparameters and feature selection space to ensure efficient convergence of the genetic algorithm.

Pros and cons of mutation operator in GA.

Advantages:

Exploration of new solutions: The mutation operator introduces random changes to the genetic material, allowing the algorithm to explore new regions of the search space. This can be beneficial when the population is stuck in local optima and needs a random perturbation to escape.

Maintaining diversity: Mutation helps maintain diversity within the population by introducing new genetic material. This is particularly important in preventing premature convergence, where the algorithm gets stuck in a suboptimal solution.

Overcoming genetic drift: Genetic drift refers to the loss of genetic information over generations due to random sampling. Mutation helps counteract this effect by continuously introducing new genetic material, ensuring that valuable traits are not lost over time.

Disadvantages:

Premature convergence: Although mutation can help avoid premature convergence in some cases, it can also hinder convergence towards the global optimum. Excessive mutation rates may disrupt the convergence process and prevent the algorithm from reaching the optimal solution.

Search inefficiency: Random changes introduced by mutation can lead to inefficient search patterns. If the mutation rate is too high, it can result in a random walk through the search space, making it difficult to converge towards the optimal solution.

Loss of good solutions: Mutation is a random operator, and it can modify or eliminate already good solutions in the population. This can result in the loss of valuable genetic material and slow down the overall progress of the algorithm.

Tuning the mutation rate: Finding an optimal mutation rate is crucial in genetic algorithms. Setting the mutation rate too low may lead to slow exploration and limited diversity, while setting it too high can disrupt convergence and hinder the algorithm's performance.

Parameters to compare performances of search algorithms in AI.

There are several parameters that can be used to compare the performance of search algorithms in AI. Some of the most common ones include:

Completeness: A search algorithm is said to be complete if it is guaranteed to find a solution if one exists. Completeness is an important property, especially in problems where finding a solution is critical, such as in mission-critical systems.

Optimality: A search algorithm is said to be optimal if it finds the optimal solution, i.e., the solution with the lowest cost. Optimality is important in problems where minimizing the cost is critical, such as in route planning or scheduling.

Time complexity: This parameter measures the amount of time required by the algorithm to find a solution. A search algorithm that has low time complexity is desirable, especially in problems that have real-time constraints.

Space complexity: This parameter measures the amount of memory required by the algorithm to find a solution. A search algorithm that has low space complexity is desirable, especially in problems where memory is limited.

UCS

Advantages:

Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Time - Let C* is Cost of the optimal solution, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search isO(b1 + [C*/ε]).

General graph search algo can be used for wide variety of search processes. Explain.

General graph search algorithms are a family of algorithms that can be used to solve a wide variety of search problems, including pathfinding, puzzle-solving, and planning problems. These algorithms work by traversing a graph, searching for a goal node or state, and returning a solution path or sequence of actions that lead to the goal.

The main advantage of general graph search algorithms is their flexibility and versatility. They can be adapted to various problem domains by changing the representation of the graph, the cost function, and the heuristic function used to guide the search.

For example, a general graph search algorithm such as Breadth-First Search (BFS) can be used to find the shortest path between two nodes in a graph, to solve a sliding puzzle, or to generate a sequence of moves that lead to a goal state in a planning problem. Similarly, Depth-First Search (DFS) can be used to find any path between two nodes in a graph, to solve a maze, or to explore a tree of possible solutions in a puzzle-solving problem.

Furthermore, general graph search algorithms can be combined with other techniques, such as pruning, optimization, or domain-specific knowledge, to improve their performance or to tackle specific problem constraints. For instance, A* search algorithm combines heuristic information with the cost function to guide the search towards the goal node efficiently.

In conclusion, general graph search algorithms provide a powerful framework for solving a wide range of search problems in AI. By adapting these algorithms to the specific requirements of the problem domain, AI researchers can develop efficient and effective solutions to complex problems in many fields, including robotics, game AI, and natural language processing.

Bidirectional search procedures are applicable to all types of problems.

Bidirectional search algorithms are a type of search algorithm that searches simultaneously from both the initial state and the goal state, and they meet somewhere in the middle. Bidirectional search can be used in a variety of search problems, including pathfinding, puzzle-solving, and game-playing problems.

The advantage of bidirectional search is that it reduces the search space and speeds up the search process. In traditional search algorithms, the search space can be very large, and the search process can be time-consuming, especially in problems with deep search trees or large graphs. Bidirectional search reduces the search space by searching from both ends, and the search meets in the middle, reducing the total search space and the time required to find a solution.

Bidirectional search is applicable to a wide range of search problems, but it is especially useful in problems where the search space is large and the cost of expanding nodes is high. For example, in pathfinding problems, bidirectional search can be used to find the shortest path between two nodes in a graph or a grid, by searching from both the start node and the goal node simultaneously.

Similarly, in puzzle-solving problems, bidirectional search can be used to find the solution to a puzzle by searching from both the initial state and the goal state simultaneously, reducing the search space and the time required to find a solution.

And-Or tree in two player game playing.

An AND-OR graph is a representation of a two-player game that captures the various possible moves and their consequences at each step of the game. In an AND-OR graph, each node represents a state of the game, and each edge represents a possible move that one of the players can make.

In a two-player game, each player takes turns making a move, and the outcome of the game depends on the sequence of moves made by both players. To capture this sequence of moves, an AND-OR graph uses two types of nodes: AND-nodes and OR-nodes.

An AND-node represents a state of the game where it is the turn of player A to move. From an AND-node, there may be multiple edges, each representing a possible move that player A can make. Each edge leads to an OR-node, which represents a state of the game where it is the turn of player B to move. From each OR-node, there may be multiple edges, each representing a possible move that player B can make.

This alternating pattern of AND-nodes and OR-nodes continues until the end of the game is reached, at which point the outcome of the game is determined based on the final state. The outcome of the game may be a win for player A, a win for player B, or a draw.

AND-OR graphs are used in game-playing algorithms such as the minimax algorithm and its variants, which use the graph to explore the possible moves and outcomes of the game in order to determine the best move for a given player. By searching through the AND-OR graph, these algorithms can evaluate the strengths and weaknesses of different moves and make optimal decisions based on the current state of the game.

CT-2

(6) $\exists z \forall y (\sim \exists x\, g(z, x, y) \wedge s(y) \wedge R(z))$

$\Rightarrow \forall y (\sim \exists x\, g(c, x, y) \wedge s(y) \wedge R(c))$

c is Skolem const.

$= \forall y (\forall x \sim g(c, x, y) \wedge s(y) \wedge R(c))$

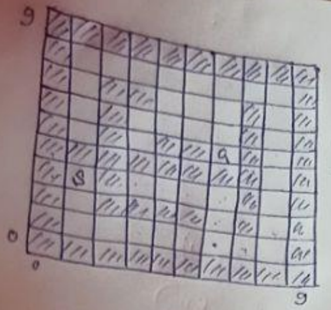
$= \forall y \forall x (\sim g(c, x, y) \wedge s(y) \wedge R(c))$

$= \underbrace{\{\sim g(c, x, y)\}}_{(i)}, \underbrace{\{s(y)\}}_{(ii)}, \underbrace{\{R(c)\}}_{(iii)}.$

2022

(2) (a) ✓

(b) P.T.O.



$h([x,y]) = (|x_g - x_n|) + (|y_g - y_n|)$

Curr	Path cost	Heu	f	Queue
S	0	7	7	(1,2)
(1,2)	1	8	9	(1,1)
(1,1)	2	9	11	(2,1)
(2,1)	3	8	11	(3,1)
(3,1)	4	7	11	(4,1)
(4,1)	5	6	11	(5,1)
(5,1)	6	5	11	(6,1)
(6,1)	7	4	11	(6,2), (7,1)
(6,2)	8	3	11	(6,3), (7,1)
(6,3)	9	2	11	(5,3), (7,1)
(5,3)	10	3	13	(7,1), (4,3)

$f(4,3) = 11+4 = 15$

$f(7,1) = 8+5 = 13$

Curr	Path cost	Heu	f	Queue
(7,1)	8	5	13	(8,1), (4,3)

|

Can be done using PQ. :)

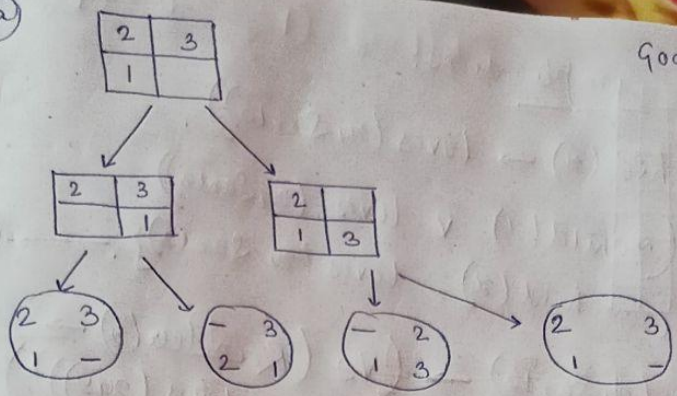
3) (a) UCS.

(b) (i) ✓

(ii) ✓

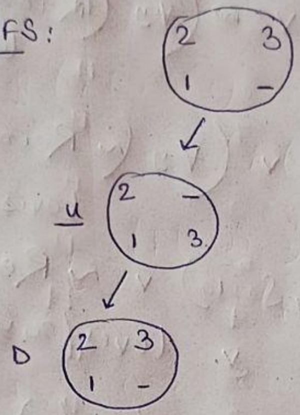
(8) (a)

Goal: 1 2
3 _



Search

By DFS:



∴, DFS cannot find the goal.

(b) FOL:

✓ $\forall x\,(child(x) \rightarrow loves(x, Santa))$.

✓ $\forall x\,(Loves(x, Santa) \rightarrow (\forall y\,(Reindeer(y) \rightarrow Loves(x, y))))$

✓ $Reindeer(Rudolph)$

✓ $Nose(Rudolph, Red)$

✓ $\forall x\,(Nose(x, Red) \rightarrow (Weird(x) \vee Clown(x)))$

✓ $\forall x\,(Reindeer(x) \rightarrow \sim Clown(x))$

✓ $\forall x\,(Weird(x) \rightarrow \sim Loves(John, x))$.

Query: $\sim child(John)$.

(stopped).

2022 PYQ:

(5) (a) ✓

(b) $\{ g(h(x,y), w), g(h(g(v), a), f(v)),$
$g(h(g(v)), f(b)) \}$

$\therefore$ $S_0 = \{ g(h(x,y), w), g(h(g(v), a), f(v)),$
$g(h(g(v)), f(b)) \}$

$D_0 = \{ x, g(v) \}$

$\sigma = \{ x = g(v) \}$

$S_1 = \{ g(h(g(v), y), w), g(h(g(v), a), f(v)),$
$g(h(g(v)), f(b)) \}$

$D_1 = \{ y, a \}$

$\sigma = \{ a = y \}$

$S_2 = \{ g(h(g(v), y), w), g(h(g(v), y), f(v)), g(h(g(v)), f(b)) \}$

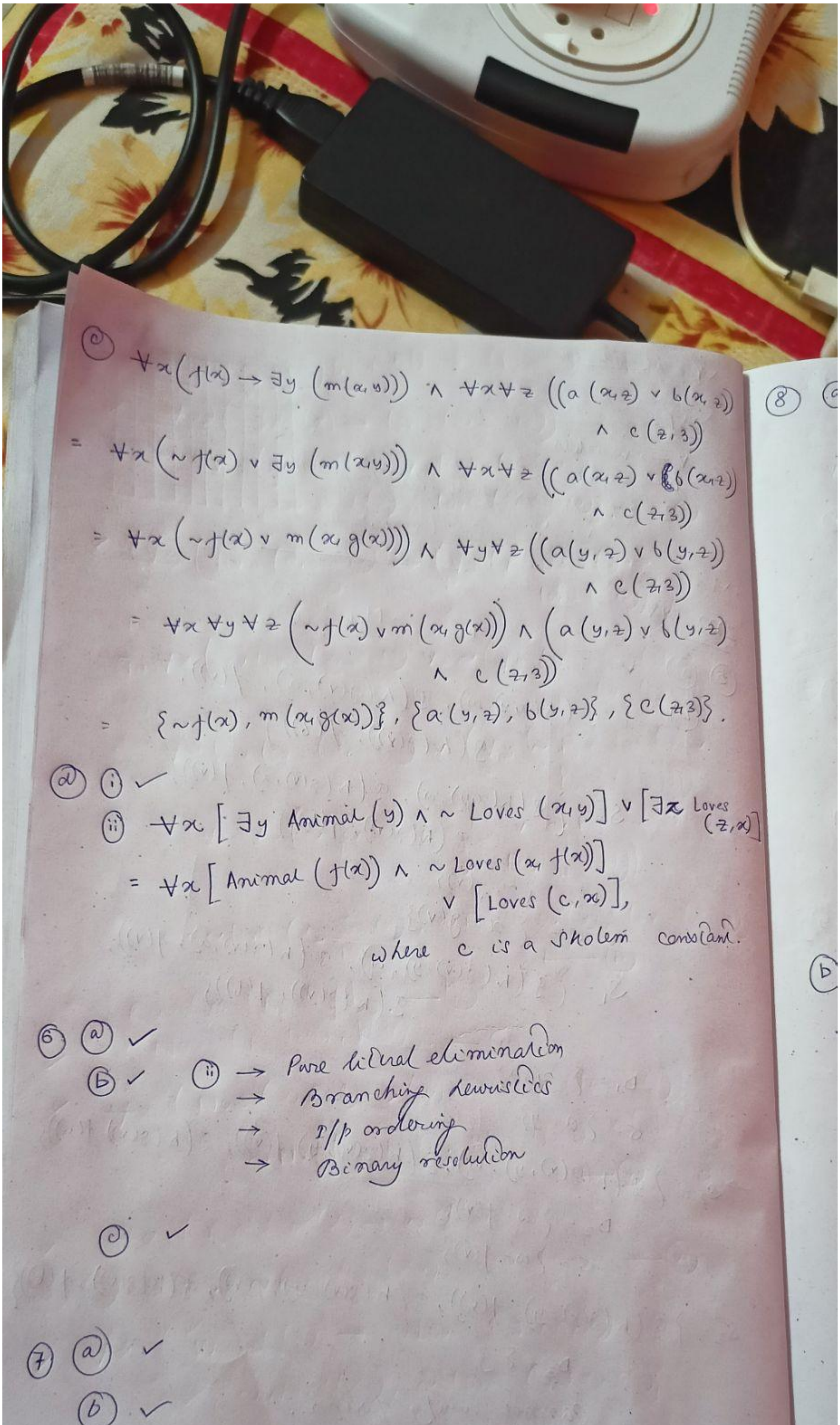
$D_2 = \{ w, f(v) \}$

$\sigma = \{ w = f(v) \}$

$S_3 = \{ g(h(g(v), y), f(v)), g(h(g(v), y), f(v)), g(h(g(v)), f(b)) \}$

$D_3 = \{ \}$

Cannot unify further.



(c) $\forall x(f(x) \rightarrow \exists y\ (m(x,y))) \wedge \forall x \forall z\ ((a(x,z) \vee b(x,z)) \wedge c(z,3))$

$= \forall x(\sim f(x) \vee \exists y\ (m(x,y))) \wedge \forall x \forall z\ ((a(x,z) \vee b(x,z)) \wedge c(z,3))$

$= \forall x\ (\sim f(x) \vee m(x, g(x))) \wedge \forall y \forall z ((a(y,z) \vee b(y,z)) \wedge c(z,3))$

$= \forall x \forall y \forall z\ (\sim f(x) \vee m(x, g(x))) \wedge (a(y,z) \vee b(y,z) \wedge c(z,3))$

$= \{\sim f(x), m(x, g(x))\}, \{a(y,z), b(y,z)\}, \{c(z,3)\}.$

(d) (i) ✓

(ii) $\forall x\ [\exists y\ Animal(y) \wedge \sim Loves(x,y)] \vee [\exists z\ Loves(z,x)]$

$= \forall x\ [Animal(f(x)) \wedge \sim Loves(x, f(x))] \vee [Loves(c,x)]$,

where c is a skolem constant.

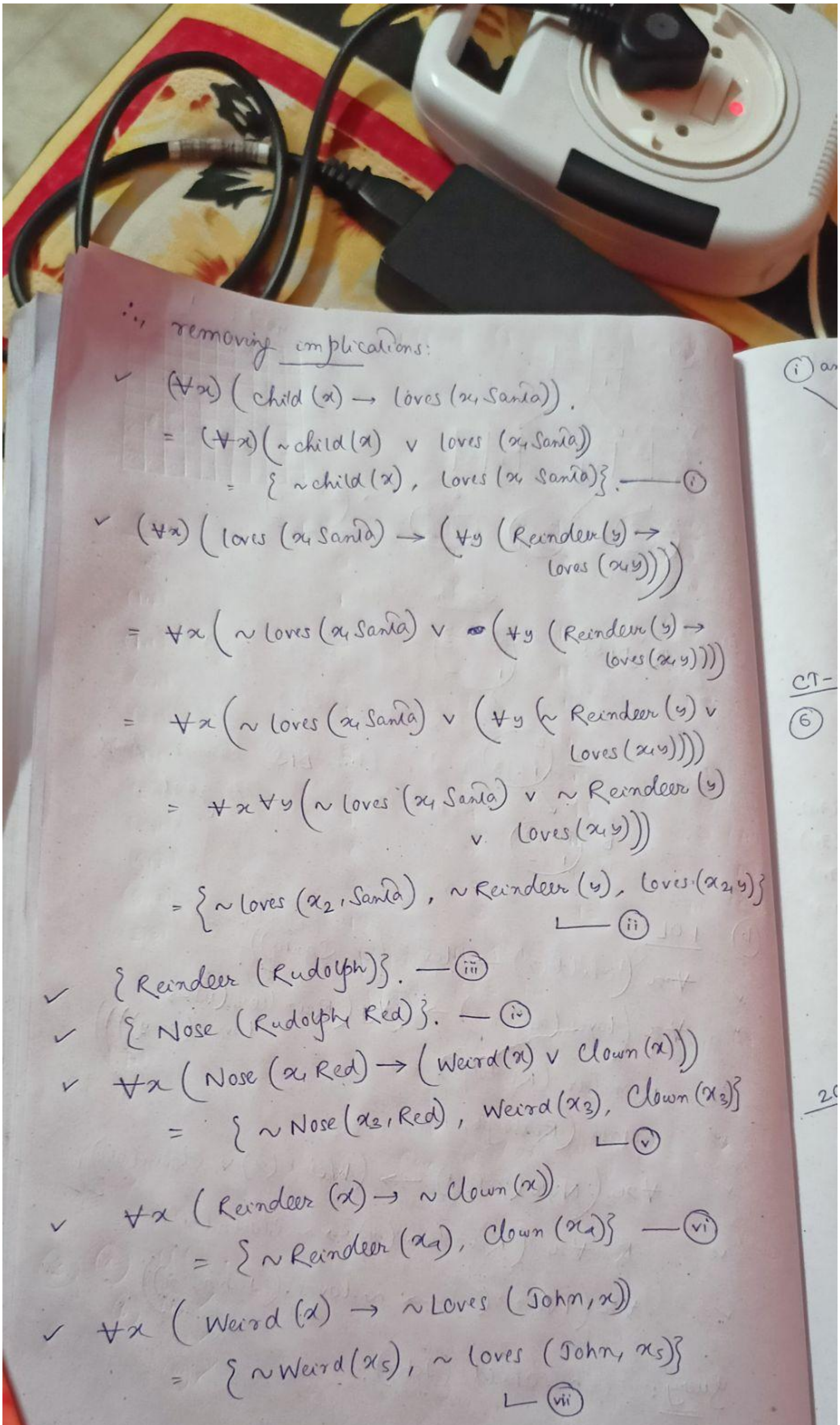
(6) (a) ✓

(b) ✓ (ii) → Pure literal elimination
→ Branching heuristics
→ I/p ordering
→ Binary resolution

(c) ✓

(7) (a) ✓

(b) ✓



∴, removing implications:

✓ $(\forall x)(child(x) \to loves(x, Santa))$.

$= (\forall x)(\sim child(x) \vee loves(x, Santa))$

$= \{\sim child(x), loves(x, Santa)\}$. — (i)

✓ $(\forall x)(loves(x, Santa) \to (\forall y(Reindeer(y) \to loves(x,y))))$

$= \forall x(\sim loves(x, Santa) \vee (\forall y(Reindeer(y) \to loves(x,y))))$

$= \forall x(\sim loves(x, Santa) \vee (\forall y(\sim Reindeer(y) \vee Loves(x,y))))$

$= \forall x \forall y(\sim loves(x, Santa) \vee \sim Reindeer(y) \vee loves(x,y))$

$= \{\sim loves(x_2, Santa), \sim Reindeer(y), loves(x_2,y)\}$ — (ii)

✓ $\{Reindeer(Rudolph)\}$. — (iii)

✓ $\{Nose(Rudolph, Red)\}$. — (iv)

✓ $\forall x(Nose(x, Red) \to (Weird(x) \vee Clown(x)))$

$= \{\sim Nose(x_3, Red), Weird(x_3), Clown(x_3)\}$ — (v)

✓ $\forall x(Reindeer(x) \to \sim Clown(x))$

$= \{\sim Reindeer(x_4), Clown(x_4)\}$ — (vi)

✓ $\forall x(Weird(x) \to \sim Loves(John, x))$

$= \{\sim Weird(x_5), \sim loves(John, x_5)\}$ — (vii)

Comment on : The aim of AI is to make machines intelligent.

The statement "The aim of AI is to make machines intelligent" is a common misconception about the field of AI. While the ultimate goal of AI is to create intelligent machines, it is important to note that the definition of intelligence is not always clear and there are different ways of achieving it.

In fact, many AI researchers focus on creating systems that can perform specific tasks or solve specific problems, rather than trying to create a general-purpose intelligence that can do anything a human can do. These systems are often based on specific algorithms or techniques that are optimised for the task at hand.

Furthermore, intelligence is not necessarily the only goal of AI research. Other goals might include improving efficiency, automating processes, or creating systems that can understand or interact with humans in more natural ways.

So while the statement that the aim of AI is to make machines intelligent is not entirely inaccurate, it is important to recognize that the field is much broader than this and encompasses a wide range of techniques, goals, and applications.

AI programming focuses on cognitive skills that include the following:

Learning. This aspect of AI programming focuses on acquiring data and creating rules for how to turn it into actionable information. The rules, which are called algorithms, provide computing devices with step-by-step instructions for how to complete a specific task.

Reasoning. This aspect of AI programming focuses on choosing the right algorithm to reach a desired outcome.

Self-correction. This aspect of AI programming is designed to continually fine-tune algorithms and ensure they provide the most accurate results possible.

Creativity. This aspect of AI uses neural networks, rules-based systems, statistical methods and other AI techniques to generate new images, new text, new music and new ideas.

What are the different types of AI ?

Based on capabilities:

i) Narrow AI:

- Narrow AI is a type of AI which is able to perform a dedicated task with intelligence.The most common and currently available AI is Narrow AI in the world of Artificial Intelligence.

ii) General AI:

- General AI is a type of intelligence which could perform any intellectual task with efficiency like a human.
- The idea behind the general AI to make such a system which could be smarter and think like a human by its own.
- Currently, there is no such system that exists which could come under general AI and can perform any task as perfect as a human.

iii) Super AI:

- Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.
- Some key characteristics of strong AI include capability include the ability to think, to reason,solve the puzzle, make judgments, plan, learn, and communicate by its own.
- Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.

Based on functionality:

1. Reactive Machines

- Purely reactive machines are the most basic types of Artificial Intelligence.
- Such AI systems do not store memories or past experiences for future actions.

- These machines only focus on current scenarios and react on it as per possible best action.
- IBM's Deep Blue system is an example of reactive machines.
- Google's AlphaGo is also an example of reactive machines.

2. Limited Memory

- Limited memory machines can store past experiences or some data for a short period of time.
- These machines can use stored data for a limited time period only.
- Self-driving cars are one of the best examples of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3. Theory of Mind

- Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans.
- This type of AI machines are still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

4. Self-Awareness

- Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness.
- These machines will be smarter than human mind.
- Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

There are several types of AI systems, each with its own characteristics and applications. Here are some of the most common types:

Rule-based systems: These systems use a set of rules to make decisions or take actions. The rules are based on if-then statements and are programmed by experts in a specific domain. Rule-based systems are often used in expert systems, which are designed to solve complex problems in specific domains such as medicine or finance.

Decision tree learning: This is a type of supervised learning algorithm that uses a tree-like model to make decisions. The tree is constructed by splitting data based on certain features until a decision can be made.

Artificial neural networks: These are a type of machine learning algorithm that are inspired by the structure and function of the human brain. They consist of layers of interconnected nodes that process and learn from data.

Fuzzy logic systems: These systems are designed to handle uncertainty and imprecision in data. They use a type of logic that allows for partial truths, rather than the binary true/false logic of traditional computing.

Evolutionary algorithms: These are a type of optimization algorithm that use techniques inspired by natural selection to find optimal solutions to problems.

Expert systems: These are AI systems that are designed to replicate the decision-making ability of a human expert in a specific domain. They are often used in areas such as medicine, finance, and engineering.

Robotics: AI is also used in robotics to enable machines to perceive their environment, make decisions, and take actions.

Limitations of Turing Test .

The Turing test, proposed by Alan Turing in 1950, is a test of a machine's ability to exhibit intelligent behavior that is indistinguishable from that of a human. While the Turing test has been influential in the development of artificial intelligence, it has some limitations:

It is a subjective measure of intelligence: The Turing test relies on the subjective judgment of human judges to determine whether a machine's behavior is indistinguishable from that of a human. This means that there is no objective measure of intelligence, and the test results can be influenced by factors such as the background and biases of the judges.

It does not measure all aspects of intelligence: The Turing test primarily measures a machine's ability to engage in natural language conversation, which is only one aspect of intelligence. It does not measure other important aspects such as problem-solving ability, creativity, or emotional intelligence.

It does not require understanding: The Turing test only requires a machine to mimic human behavior, not to truly understand the meaning behind it. This means that a machine could pass the Turing test without truly understanding the concepts it is using.

It may not be a good measure of true intelligence: The Turing test is designed to measure a machine's ability to mimic human behavior, but it is not clear that this is a good measure of true intelligence. Intelligence is a complex and multifaceted concept, and it is not clear that simply being able to mimic human behavior is a sufficient criterion for intelligence.

Overall, while the Turing test has been an influential concept in the field of artificial intelligence, it is important to recognize its limitations and to continue to develop more comprehensive and objective measures of intelligence.

Horizon effect

The horizon effect, also known as the horizon problem, is a problem in artificial intelligence whereby, in many games, the number of possible states or positions is immense and computers can only feasibly search a small portion of them, typically a few plies down the game tree. Thus, for a computer searching only five plies, there is a possibility that it will make a detrimental move, but the effect is not visible because the computer does not search to the depth of the error (*i.e.*, beyond its "horizon").

When evaluating a large game tree using techniques such as minimax with alpha-beta pruning, search depth is limited for feasibility reasons. However, evaluating a partial tree may give a misleading result. When a significant change exists just over the horizon of the search depth, the computational device falls victim to the horizon effect.

In 1973 Hans Berliner named this phenomenon, which he and other researchers had observed, the "Horizon Effect."[1] He split the effect into two: the Negative Horizon Effect "results in creating diversions which ineffectively delay an unavoidable consequence or make an unachievable one appear achievable." For the "largely overlooked" Positive Horizon Effect, "the program grabs much too soon at a consequence that can be imposed on an opponent at leisure, frequently in a more effective form."

Greedy algorithms tend to suffer from the horizon effect.

The horizon effect can be mitigated by extending the search algorithm with a quiescence search. This gives the search algorithm ability to look beyond its horizon for a certain class of moves of major importance to the game state, such as captures in chess.

Rewriting the evaluation function for leaf nodes and/or analyzing more nodes will solve many horizon effect problems.

And tree AND Or tree difference:

An AND tree and an OR tree are both types of search trees used in artificial intelligence and computer science, but they differ in their structure and function.

An AND tree is a type of search tree that is used to represent a problem where multiple conditions must be met in order to reach a solution. Each node in an AND tree represents a condition, and the branches represent possible solutions to that condition. The tree is traversed in a depth-first manner, exploring all possible solutions until a complete solution is found.

On the other hand, an OR tree is a type of search tree that is used to represent a problem where there are multiple possible solutions. Each node in an OR tree represents a possible solution, and the branches represent the conditions that must be met to reach that solution. The tree is traversed in a breadth-first manner, exploring all possible solutions until a satisfactory solution is found.

In summary, the key difference between an AND tree and an OR tree is the type of problem they are used to represent. An AND tree is used when multiple conditions must be met in order to find a solution, while an OR tree is used when there are multiple possible solutions to a problem.

Write rules of substitution in FOL.

In first-order logic (FOL), substitutions are used to replace variables with terms. The rules for substitutions in FOL are as follows:

- Any term may be substituted for itself.
- A variable may be substituted for any term.
- A term may be substituted for a variable only if the variable does not occur in the term.
- If a term T1 can be substituted for a variable X in a formula F to obtain a new formula G, then for any formula H that contains G as a subformula, a term T2 may be substituted for T1 to obtain a formula that contains T2 in place of X.

These rules allow for the manipulation of formulas in FOL by replacing variables with terms to create new formulas that are equivalent to the original. It is important to note that the rules of substitutions must be used carefully to avoid introducing errors or changing the meaning of the formula.

Ancestry filtered form

Ancestry filtered form is a way of representing a logical formula in first-order logic (FOL) that involves eliminating certain variables and quantifiers. It is also known as Skolem normal form or prenex normal form.

To transform a FOL formula into ancestry filtered form, the following steps are taken:

Eliminate existential quantifiers by introducing Skolem functions. These functions introduce new constants or functions into the formula that represent the value of the existentially quantified variables in terms of the universally quantified variables.

Eliminate universal quantifiers by moving them to the front of the formula, separated by the appropriate connective (AND or OR). This results in a formula in prenex normal form.

Eliminate any redundant variables that appear only in the scope of quantifiers that bind other variables. This is called ancestry filtering, and it involves removing variables that are not needed to represent the formula.

The resulting formula is in ancestry filtered form, which is a simplified version of the original formula that retains the same logical meaning. Ancestry filtered form is useful for simplifying logical formulas and making them easier to work with in automated reasoning and other applications.

Ethical issues in AI :

https://www.weforum.org/agenda/2016/10/top-10-ethical-issues-in-artificial-intelligence/