Dayananda Sagar University

School of Engineering, Hosur Main Road, Kudlu Gate, Bengaluru-560 068

ARTIFICIAL INTELLIGENCE-II
20AM3602

20AM3602

SEMESTER – VI

Course Code: 20AM3602

Prof. Pradeep Kumar K

Dept. of CS&E (AIML)

DSU, Bangalore

20AM3602

Course Objectives:

- Gain a perspective of state space search in AI
- Investigate applications of AI techniques in intelligent agents, expert systems,
- Understanding Natural language and its importance in AI

Course outcomes:

CO No.	Outcomes	Bloom's Taxonomy Level
CO1	Demonstrate awareness and a fundamental understanding of various applications of AI techniques in intelligent agents, expert systems, artificial neural networks and other machine learning models.	L5
CO2	Demonstrate proficiency in applying scientific method to models using PROLOG and LISP	L4
CO3	Apply AI techniques to real-world problems to develop intelligent systems	L4
CO4	Design, analyse and demonstrate AI applications or systems that apply to real life problems.	L5

20AM3602

Text Documents Requirement

TEXT BOOKS:

1. George F Luger, William A Stubblefield. Artificial Intelligence: Structures and Strategies for Complex Problem Solving, 3rd Edition, Addison Wesley Longman, Inc

2. Introduction to Artificial Intelligence and Expert Systems by Dan W. Patterson, Pearson Education

3. Forsyth and Ponce, “Computer Vision – A Modern Approach”, Second Edition, Prentice Hall, 2011.

REFERENCES:

1. Daniel Jurafsky, James H. Martin. Speech and Language processing: An Introduction to Natural Language processing, computational Linuistics and Speech, Pearson Publication, 2014

2. Artificial Intelligence: Concepts and Applications, Wiley (1 January 2021); ISBN-10 : 8126519932.

Content of Syllabus

20AM3602

Module-1:	Contact Hours
AI as Representation and Search: The predicate Calculus, Using Inference Rules to produce Predicate Calculus Expressions, Structures and Strategies for State Space Search: Graph Theory, Strategies for State Space Search, using state space to represent Reasoning with predicate calculus Recursion-based search, production systems, Predicate Calculus and Planning	10
Module – 2:	
Programming Languages for AI An Introduction to PROLOG: Syntax for predicate Calculus programming, ADTs in PROLOG, A production system example in PROLOG. An Introduction to LISP: LISP - A brief overview, Search in LISP, Pattern matching in LISP	08
Module – 3:	
Expert Systems: Introduction to AI agents, Overview of expert system Technology, Rule Based expert systems, Model based reasoning, Case-based Reasoning, knowledge-Representation problem, An Expert system Shell in LISP.	08

Content of Syllabus

20AM3602

Module-4:	Contact Hours
Understanding Natural Language: Role of knowledge in Language Understanding, Language Understanding: A symbolic approach, Syntax, Combining Syntax and semantic in ATN parsers, Stochastic Tool for language Analysis, Natural Language Applications: Story Understanding and Question Answering	07
Module – 5:	
Pattern Recognition: Introduction, Recognition and classification process, learning Classification patterns, Recognizing and Understanding Speech Computer Vision Overview and State-of-the-art, Fundamentals of Image Formation, Transformation: Orthogonal, Euclidean, Affine, Projective, etc; Fourier Transform, Convolution and Filtering, Image Enhancement, Restoration, Histogram Processing	08

20AM3602

Module-1 : AI as Representation and Search:

- AI as Representation and Search: The predicate Calculus, Using Inference Rules to produce Predicate Calculus Expressions, Structures and Strategies for State Space Search: Graph Theory, Strategies for State Space Search, using state space to represent Reasoning with predicate calculus Recursion-based search, production systems, Predicate Calculus and Planning

20AM3602

Learning Outcome

- Introduce the predicate calculus as a representation language for artificial intelligence.
- Predicate calculus and its advantages include a well-defined formal semantics and sound and complete inference rules.
- Explanation begins with a brief (optional) review of the propositional calculus.
- Syntax and semantics of the predicate calculus.
- Predicate calculus inference rules and their use in problem solving.
- Finally, to demonstrates the use of the predicate calculus to implement a knowledge base for financial investment advice.

20AM3602

The Propositional Calculus (optional)

Symbols and Sentences

- The propositional calculus and the predicate calculus are first of all languages.
- Using their words, phrases, and sentences, we can represent and reason about properties and relationships in the world.
- The first step in describing a language is to introduce the pieces that make it up: its set of symbols.

20AM3602

PROPOSITIONAL CALCULUS SYMBOLS

The symbols of propositional calculus are the propositional symbols: P, Q, R, S, …

Truth symbols: true, false

Connectives: ∧, ∨, ¬, →, ≡

20AM3602

PROPOSITIONAL CALCULUS SENTENCES

- Every propositional symbol and truth symbol is a sentence.

For example: true, P, Q, and R are sentences.

- The negation of a sentence is a sentence.

For example: ¬ P and ¬ false are sentences.

- The conjunction, or and, of two sentences is a sentence.

For example: P ∧ ¬ P is a sentence.

- The disjunction, or or, of two sentences is a sentence.

For example: P ∨ ¬ P is a sentence.

- The implication of one sentence from another is a sentence.

For example: P → Q is a sentence.

- The equivalence of two sentences is a sentence.

For example: P ∨ Q ≡ R is a sentence.

- Legal sentences are also called well-formed formulas or WFFs.

- In expressions of the form P ∧ Q, P and Q are called the conjuncts.
- In P ∨ Q, P and Q are referred to as disjuncts. In an implication,
- P → Q, P is the premise or antecedent and Q, the conclusion or consequent.

- In propositional calculus sentences, the symbols ( ) and [ ] are used to group symbols into subexpressions and so to control their order of evaluation and meaning.

For example,: (P ∨ Q) ≡ R is quite different from P ∨ (Q ≡ R).

- An expression is a sentence, or well-formed formula, of the propositional calculus if and only if it can be formed of legal symbols through some sequence of these rules.

For example: **((P ∧ Q) → R) ≡ ¬ P ∨ ¬ Q ∨ R** is a well-formed sentence in the propositional calculus because:

- P, Q, and R are propositions and thus sentences.
- P ∧ Q, the conjunction of two sentences, is a sentence.
- (P ∧ Q) → R, the implication of a sentence for another, is a sentence.
- ¬ P and ¬ Q, the negations of sentences, are sentences.
- ¬ P ∨ ¬ Q, the disjunction of two sentences, is a sentence.
- ¬ P ∨ ¬ Q ∨ R, the disjunction of two sentences, is a sentence.
- ((P ∧ Q) → R) ≡ ¬ P ∨ ¬ Q ∨ R, the equivalence of two sentences, is a sentence.

20AM3602

PROPOSITIONAL CALCULUS SEMANTICS

- An interpretation of a set of propositions is the assignment of a truth value, either T or F, to each propositional symbol. The symbol true is always assigned T, and the symbol false is assigned F.

The interpretation or truth value for sentences is determined by:

- The truth assignment of negation, ¬ P, where P is any propositional symbol, is F if the assignment to P is T, and T if the assignment to P is F.
- The truth assignment of conjunction, ∧, is T only when both conjuncts have truth value T; otherwise it is F.
- The truth assignment of disjunction, ∨, is F only when both disjuncts have truth value F; otherwise it is T.
- The truth assignment of implication, →, is F only when the premise or symbol before the implication is T and the truth value of the consequent or symbol after the implication is F; otherwise it is T.
- The truth assignment of equivalence, ≡, is T only when both expressions have the same truth assignment for all possible interpretations; otherwise it is F.

20AM3602

For propositional expressions P, Q, and R:

- $\neg (\neg P) \equiv P$
- $(P \vee Q) \equiv (\neg P \rightarrow Q)$
- the contrapositive law: $(P \rightarrow Q) \equiv (\neg Q \rightarrow \neg P)$
- de Morgan's law: $\neg (P \vee Q) \equiv (\neg P \wedge \neg Q)$ and $\neg (P \wedge Q) \equiv (\neg P \vee \neg Q)$
- the commutative laws: $(P \wedge Q) \equiv (Q \wedge P)$ and $(P \vee Q) \equiv (Q \vee P)$
- the associative law: $((P \wedge Q) \wedge R) \equiv (P \wedge (Q \wedge R))$
- the associative law: $((P \vee Q) \vee R) \equiv (P \vee (Q \vee R))$
- the distributive law: $P \vee (Q \wedge R) \equiv (P \vee Q) \wedge (P \vee R)$
- the distributive law: $P \wedge (Q \vee R) \equiv (P \wedge Q) \vee (P \wedge R)$

20AM3602

Understanding the Truth Table

P	Q	P∧Q
T	T	T
T	F	F
F	T	F
F	F	F

Figure 2.1 Truth table for the operator ∧.

P	Q	¬P	¬P∨Q	P⇒Q	(¬P∨Q)=(P⇒Q)
T	T	F	T	T	T
T	F	F	F	F	T
F	T	T	T	T	T
F	F	T	T	T	T

Figure 2.2 Truth table demonstrating the equivalence of $P \rightarrow Q$ and $\neg P \vee Q$.

20AM3602

The Predicate calculus

- In propositional calculus, each atomic symbol (P, Q, etc.) denotes a single proposition. **There is no way to access the components of an individual assertion.**
- Predicate calculus provides this ability. For example, instead of letting a single propositional symbol, P, denote the entire sentence “it rained on Tuesday,” we can create a predicate weather that describes a relationship between a date and the weather: weather(tuesday, rain).
- Through inference rules we can manipulate predicate calculus expressions, accessing their individual components and inferring new sentences.
- Predicate calculus also allows expressions to contain variables.
- Variables are used to create general assertions about classes of entities.
- For example, we could state that for all values of X, where X is a day of the week, the statement weather(X, rain) is true; i.e., it rains every day. As we did with the propositional calculus, we will first define the syntax of the language and then discuss its semantics.

20AM3602

The Syntax of Predicates and Sentences

- Before defining the syntax of correct expressions in the predicate calculus, we need to define an alphabet and grammar for creating the symbols of the language.
- This corresponds to the **lexical aspect of a programming language definition**.
- Predicate calculus symbols, like the tokens in a programming language, are irreducible syntactic elements: they cannot be broken into their component parts by the operations of the language.
- Use of predicate calculus symbols as strings of letters and digits beginning with a letter.

20AM3602

PREDICATE CALCULUS SYMBOLS

The alphabet that makes up the symbols of the predicate calculus consists of:

- The set of letters, both upper- and lowercase, of the English alphabet.
- The set of digits, 0, 1, ..., 9.
- The underscore, _.

Symbols in the predicate calculus begin with a letter and are followed by any sequence of these legal characters.

- Legitimate characters in the alphabet of predicate calculus symbols include

a R 6 9 p _ z

Examples of characters not in the alphabet include

% @ / &

- Legitimate predicate calculus symbols include

George fire3 tom_and_jerry bill XXXX friends_of

- Examples of strings that are not legal symbols are

3jack no blanks allowed ab%cd ***71 duck!!!

- As with most programming languages, the use of “words” that suggest the symbol’s intended meaning assists us in understanding program code.

  l(g,k)

  and

  likes(george, kate) are formally equivalent

Parentheses “( )”, commas “,”, and periods “.” are used solely to construct well-formed expressions.

- Predicate calculus symbols may represent either

1. variables,
2. constants,
3. functions, or predicates.

- **Constants** name specific objects or properties in the world. Constant symbols must begin with a lowercase letter. Thus george, tree, tall, and blue are examples of well-formed constant symbols. The constants true and false are reserved as truth symbols.
- **Variable** symbols are used to designate general classes of objects or properties in the world. Variables are represented by symbols beginning with an uppercase letter. Thus George, BILL, and KAte are legal variables, whereas geORGE and bill are not.
- Predicate calculus also allows **functions** on objects in the world of discourse. Function symbols (like constants) begin with a lowercase letter.
- **Functions** denote a mapping of one or more elements in a set (called the domain of the function) into a unique element of a second set (the range of the function). Elements of the domain and range are objects in the world of discourse. In addition to common arithmetic functions such as addition and multiplication, **functions may define mappings between nonnumeric domains.**

20AM3602

SYMBOLS and TERMS

Predicate calculus symbols include:

- Truth symbols true and false (these are reserved symbols).
- Constant symbols are symbol expressions having the first character lowercase.
- Variable symbols are symbol expressions beginning with an uppercase character.
- Function symbols are symbol expressions having the first character lowercase.

Functions have an attached arity indicating the number of elements of the domain mapped onto each element of the range.

- A function expression consists of a function constant of arity n, followed by n terms, t1,, , t2 … tn, enclosed in parentheses and separated by commas.
- A predicate calculus term is either a constant, variable, or function expression.
- Thus, a predicate calculus term may be used to denote objects and properties in a problem domain. Examples of terms are:
  - cat
  - times(2,3)
  - X
  - blue
  - mother(sarah)
  - kate

- Symbols in predicate calculus may also represent predicates. Predicate symbols, like constants and function names, begin with a lowercase letter.
- A predicate names a relationship between zero or more objects in the world. The number of objects so related is the arity of the predicate. Examples of predicates are

likes equals on near part_of

- An atomic sentence, the most primitive unit of the predicate calculus language, is a predicate of arity n followed by n terms enclosed in parentheses and separated by commas. Examples of atomic sentences are

likes(george,kate) likes(X,george)

likes(george,susie) likes(X,X)

likes(george,sarah,tuesday) friends(bill,richard)

friends(bill,george) friends(father_of(david),father_of(andrew))

helps(bill,george) helps(richard,bill)

20AM3602

PREDICATES and ATOMIC SENTENCES

- Predicate symbols are symbols beginning with a lowercase letter.
- Predicates have an associated positive integer referred to as the arity or “argument number” for the predicate. Predicates with the same name but different arities are considered distinct.
- An atomic sentence is a predicate constant of arity n, followed by n terms, t1,t2….tn, enclosed in parentheses and separated by commas.
- The truth values, true and false, are also atomic sentences.
- Atomic sentences are also called atomic expressions, atoms, or propositions. We may combine atomic sentences using logical operators to form sentences in the predicate calculus.
- These are the same logical connectives used in propositional calculus: ∧, ∨, ¬, →, and ≡.

20AM3602

PREDICATE CALCULUS SENTENCES

Every atomic sentence is a sentence.

- If s is a sentence, then so is its negation, ¬ s.
- If s1 and s2 are sentences, then so is their conjunction, s1 ∧ s2.
- If s1 and s2 are sentences, then so is their disjunction, s1 ∨ s2.
- If s1 and s2 are sentences, then so is their implication, s1 → s2.
- If s1 and s2 are sentences, then so is their equivalence, s1 ≡ s2.
- If X is a variable and s a sentence, then ∀ X s is a sentence.
- If X is a variable and s a sentence, then ∃ X s is a sentence.

- predicate calculus includes two symbols, the variable quantifiers ∀ and ∃, that constrain the meaning of a sentence containing a variable. A quantifier is followed by a variable and a sentence, such as

∃ Y friends(Y, peter)

∀ X likes(X, ice_cream)

- The universal quantifier, ∀, indicates that the sentence is **true for all values** of the variable. In the example, ∀ X likes(X, ice_cream) is true for all values in the domain of the definition of X.
- The existential quantifier, ∃, indicates that the sentence is **true for at least one value** in the domain. ∃ Y friends(Y, peter) is true if there is at least one object, indicated by Y that is a friend of peter.

Using Inference Rules to produce Predicate Calculus Expressions

20AM3602

Inference Rules

- The semantics of the predicate calculus provides a basis for a **formal theory of logical inference.**
- The ability to infer new correct expressions from a set of true assertions is an important feature of the predicate calculus. These new expressions are correct in that they are consistent with all previous interpretations of the original set of expressions.
- An interpretation that makes a sentence true is said to satisfy that sentence. An interpretation that satisfies every member of a set of expressions is said to satisfy the set.
- **An expression X logically follows from a set of predicate calculus expressions S if every interpretation that satisfies S also satisfies X.**
- This notion gives us a basis for verifying the correctness of rules of inference: **the function of logical inference is to produce new sentences that logically follow a given set of predicate calculus sentences.**

- An inference rule is essentially a mechanical means of producing new predicate calculus sentences from other sentences.
- When every sentence X produced by an inference rule operating on a set S of logical expressions logically follows from S, the **inference rule is said to be sound.**
- If the inference rule is able to produce every expression that logically follows from S, then **it is said to be complete**. Modus ponens and resolution, are examples of inference rules that are sound and, when used with appropriate application strategies.
- Logical inference systems generally use sound rules of inference, examine heuristic reasoning and commonsense reasoning, both of which relax this requirement.

20AM3602

DEFINITION

SATISFY, MODEL, VALID, INCONSISTENT

For a predicate calculus expression X and an interpretation I:

- If X has a value of T under I and a particular variable assignment, then I is said to satisfy X.
- If I satisfies X for all variable assignments, then I is a model of X.
- X is satisfiable if and only if there exist an interpretation and variable assignment that satisfy it; otherwise, it is unsatisfiable.
- A set of expressions is satisfiable if and only if there exist an interpretation and variable assignment that satisfy every element.
- If a set of expressions is not satisfiable, it is said to be inconsistent.
- If X has a value T for all possible interpretations, X is said to be valid.

PROOF PROCEDURE

- A proof procedure is a combination of an inference rule and an algorithm for applying that rule to a set of logical expressions to generate new sentences.
- Using these definitions, we may formally define “logically follows.”

LOGICALLY FOLLOWS, SOUND, and COMPLETE

- A predicate calculus expression X logically follows from a set S of predicate calculus expressions if every interpretation and variable assignment that satisfies S also satisfies X.
- An inference rule is sound if every predicate calculus expression produced by the rule from a set S of predicate calculus expressions also logically follows from S.
- An inference rule is complete if, given a set S of predicate calculus expressions, the rule can infer every expression that logically follows from S

20AM3602

MODUS PONENS, MODUS TOLLENS, AND ELIMINATION, AND INTRODUCTION, and UNIVERSAL INSTANTIATION

Modus Ponens by affirming the truth of an argument (the conclusion becomes the affirmation), and Modus Tollens by denial (again, the conclusion is the denial).

- If the sentences P and P → Q are known to be true, then modus ponens lets us infer Q. known to be false, we can infer that P is false: ¬ P.
- **Elimination** allows us to infer the truth of either of the conjuncts from the truth of a conjunctive sentence. For instance, P ∧ Q lets us conclude P and Q are true.
- **Introduction** lets us infer the truth of a conjunction from the truth of its conjuncts. For instance, if P and Q are true, then P ∧ Q is true.
- **Universal instantiation** states that if any universally quantified variable in a true sentence, say p(X), is replaced by an appropriate term from the domain, the result is a true sentence. Thus, if a is from the domain of X, ∀ X p(X) lets us infer p(a).

Consider as an example the common syllogism “all men are mortal and Socrates is a man; therefore Socrates is mortal.” “All men are mortal” may be represented in predicate calculus by:

- ∀ X (man(X) → mortal(X)).
- “Socrates is a man” is
- man(socrates).
- Because the X in the implication is universally quantified, we may substitute any value in the domain for X and still have a true statement under the inference rule of universal instantiation. By substituting socrates for X in the implication, we infer the expression
- man(socrates) → mortal(socrates).
- An algorithm called ***unification*** can be used by an automated problem solver to determine that socrates may be substituted for X in order to apply modus ponens.

20AM3602

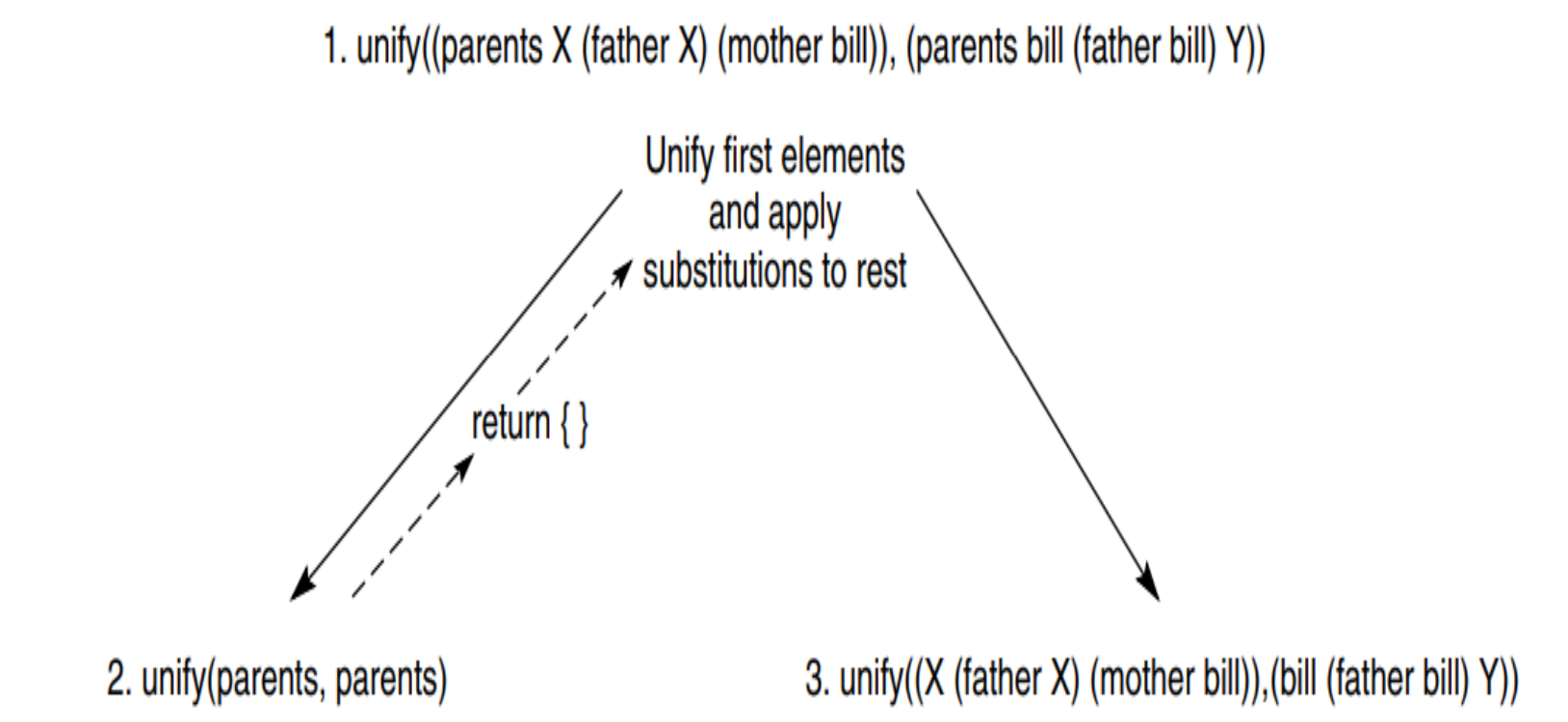
Unification

- Unification is an algorithm for determining the substitutions needed to make two predicate calculus expressions match.
- In the previous subsection, where socrates in man(socrates) was substituted for X in ∀ X(man(X) ⇒mortal(X)).
- This allowed the application of modus ponens and the conclusion mortal(socrates).
- p(X) and p(Y) are equivalent, Y may be substituted for X to make the sentences match.
- Unification and inference rules such as modus ponens allow us to make inferences on a set of logical assertions.
- The logical database must be expressed in an appropriate form.

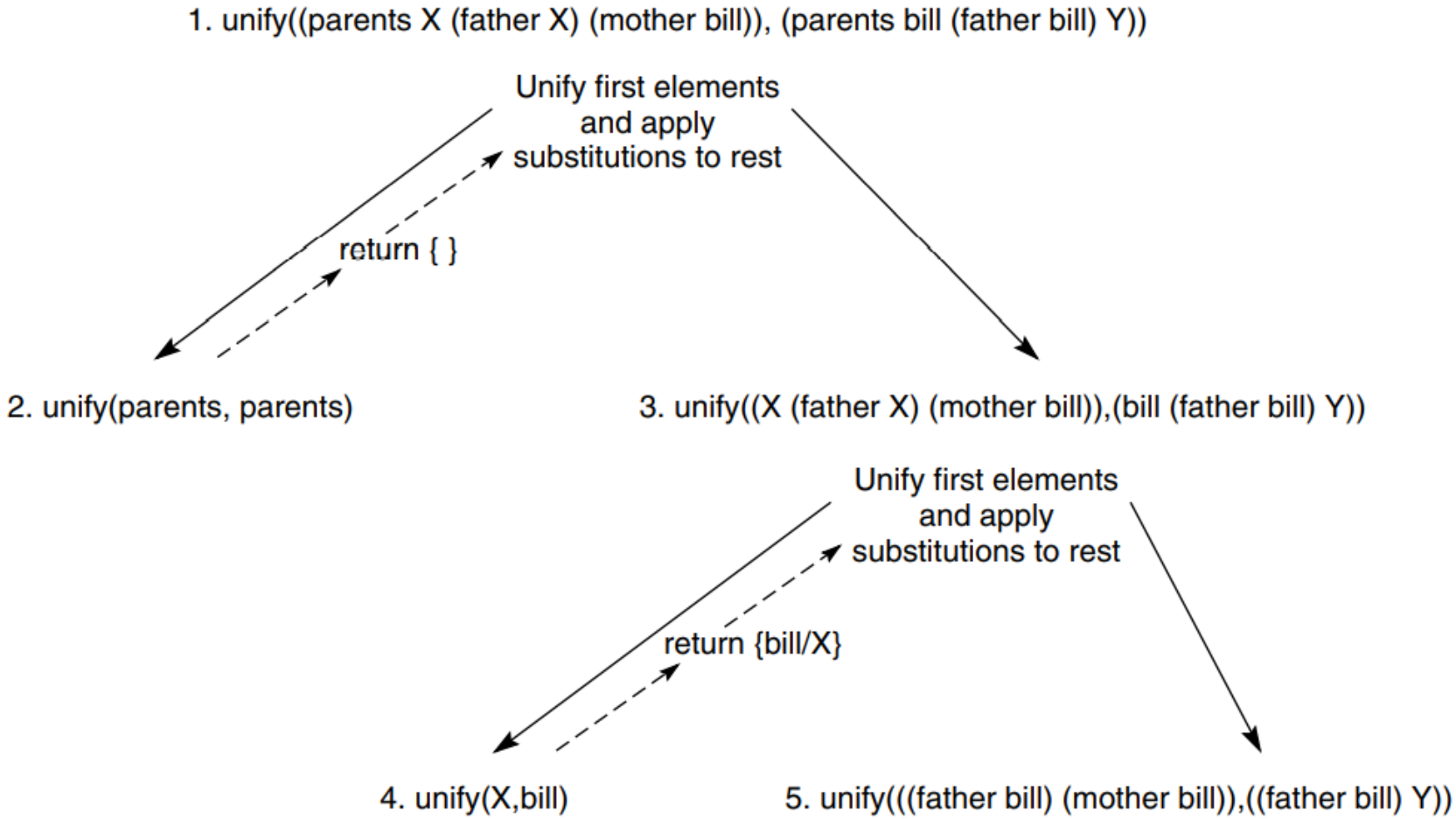
A Unification Example

- The behavior of the preceding algorithm may be clarified by tracing the call
- unify((parents X (father X) (mother bill)), (parents bill (father bill) Y)).
- When unify is first called, because neither argument is an atomic symbol, the function will attempt to recursively unify the first elements of each expression, calling
- unify(parents, parents).
- This unification succeeds, returning the empty substitution, {}.
- Applying this to the remainder of the expressions creates no change; the algorithm then calls
- unify((X (father X) (mother bill)), (bill (father bill) Y)).

A tree depiction of the execution at this stage appears



- In the second call to unify, neither expression is atomic, so the algorithm separates each expression into its first component and the remainder of the expression. This leads to the call
- unify(X, bill).
- This call succeeds, because both expressions are atomic and one of them is a variable. The call returns the substitution {bill/X}. This substitution is applied to the remainder of each expression and unify is called on the results,
- unify(((father bill) (mother bill)), ((father bill)Y)).
- The result of this call is to unify (father bill) with (father bill). This leads to the calls
- unify(father, father)
- unify(bill, bill)
- unify(( ), ( ))


1. unify((parents X (father X) (mother bill)), (parents bill (father bill) Y))
Unify first elements and apply substitutions to rest
return { }
2. unify(parents, parents)
3. unify((X (father X) (mother bill)),(bill (father bill) Y))
Unify first elements and apply substitutions to rest
return {bill/X}
4. unify(X,bill)
5. unify(((father bill) (mother bill)),((father bill) Y))

All of these succeed, returning the empty set of substitutions. Unify is then called on the remainder of the expressions:

unify(((mother bill)), (Y)).

This, in turn, leads to calls

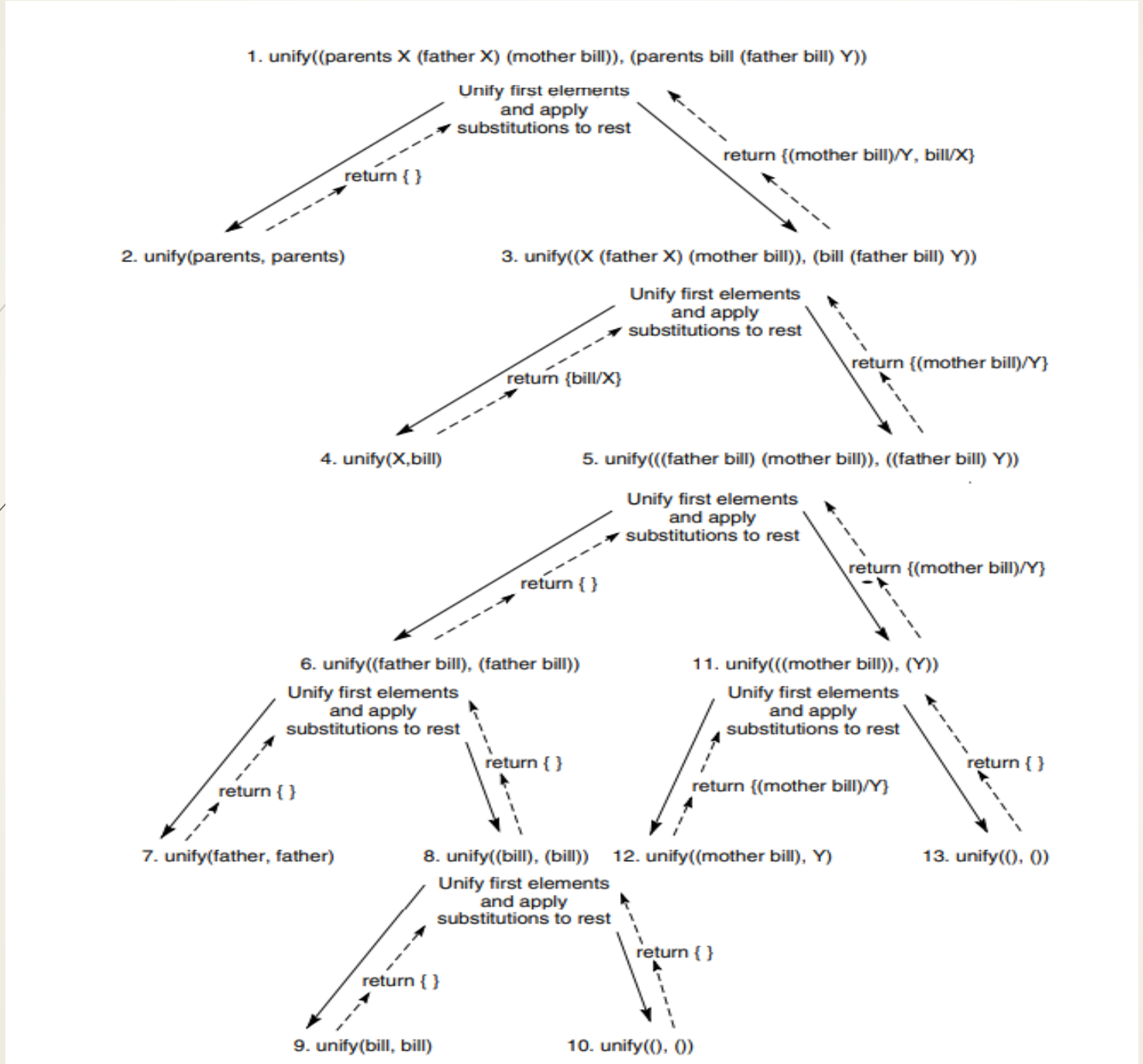
unify((mother bill), Y)

unify(( ),( ))

In the first of these, (mother bill) unifies with Y. Notice that unification substitutes the whole structure (mother bill) for the variable Y. Thus, unification succeeds and returns the substitution {(mother bill)/Y}. The call

unify(( ),( ))

returns { }. All the substitutions are composed as each recursive call terminates, to return the answer {bill/X (mother bill)/Y}. A trace of the entire execution appears. Each call is numbered to indicate the order in which it was made; the substitutions returned by each call are noted on the arcs of the tree.


1. unify((parents X (father X) (mother bill)), (parents bill (father bill) Y))
Unify first elements and apply substitutions to rest
return { }
return {(mother bill)/Y, bill/X}
2. unify(parents, parents)
3. unify((X (father X) (mother bill)), (bill (father bill) Y))
Unify first elements and apply substitutions to rest
return {bill/X}
return {(mother bill)/Y}
4. unify(X,bill)
5. unify(((father bill) (mother bill)), ((father bill) Y))
Unify first elements and apply substitutions to rest
return { }
return {(mother bill)/Y}
6. unify((father bill), (father bill))
11. unify(((mother bill)), (Y))
Unify first elements and apply substitutions to rest
return { }
return { }
7. unify(father, father)
8. unify((bill), (bill))
Unify first elements and apply substitutions to rest
return {(mother bill)/Y}
return { }
12. unify((mother bill), Y)
13. unify((), ())
Unify first elements and apply substitutions to rest
return { }
return { }
9. unify(bill, bill)
10. unify((), ())

20AM3602

Structures and Strategies for State Space Search

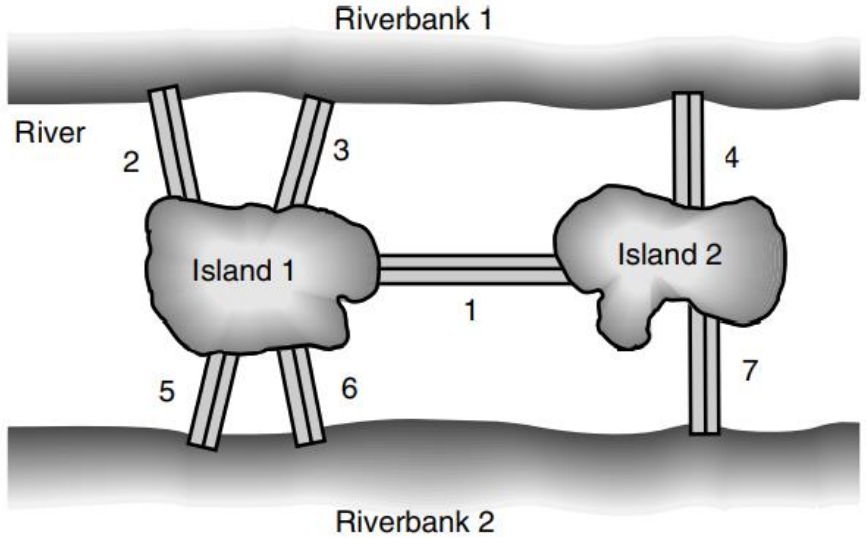
To successfully design and implement search algorithms, a programmer must be able to analyze and predict their behavior. Questions that need to be answered include:

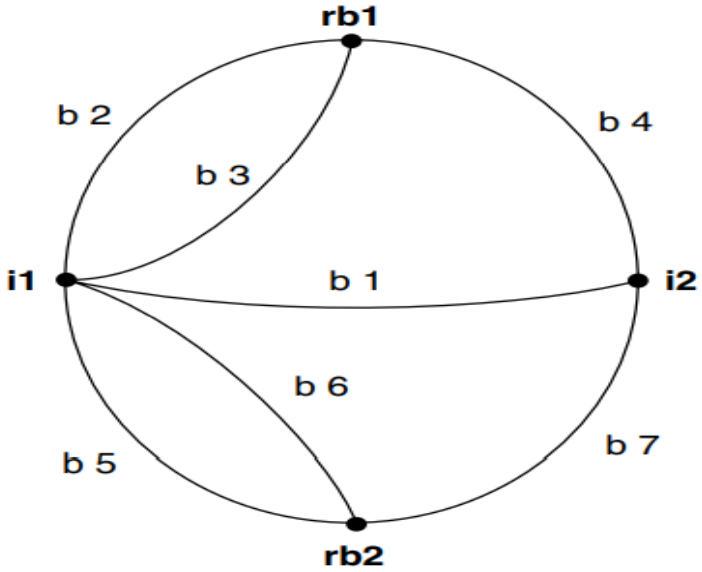
- Is the problem solver guaranteed to find a solution?
- Will the problem solver always terminate? Can it become caught in an infinite loop?
- When a solution is found, is it guaranteed to be optimal?
- What is the complexity of the search process in terms of time usage? Memory usage?
- How can the interpreter most effectively reduce search complexity?
- How can an interpreter be designed to most effectively utilize a representation language?

- The theory of state space search is the primary tool for answering these questions. By representing a problem as a state space graph, we can use graph theory to analyze the structure and complexity of both the problem and the search procedures that we employ to solve it.

- A graph consists of a set of nodes and a set of arcs or links connecting pairs of nodes.
- In the state space model of problem solving, the nodes of a graph are taken to represent discrete states in a problem-solving process, such as the results of logical inferences or the different configurations of a game board.
- The arcs of the graph represent transitions between states. These transitions correspond to logical inferences or legal moves of a game.
- In expert systems, for example, states describe our knowledge of a problem instance at some stage of a reasoning process. Expert knowledge, in the form of if . . . Then rules, allows us to generate new information; the act of applying a rule is represented as an arc between states.

- Graph theory is our best tool for reasoning about the structure of objects and relations; indeed, this is precisely the need that led to its creation in the early eighteenth century.
- The Swiss mathematician Leonhard Euler invented graph theory to solve the “bridges of Königsberg problem.” The city of Königsberg occupied both banks and two islands of a river. The islands and the riverbanks were connected by seven bridges, as indicated.





20AM3602

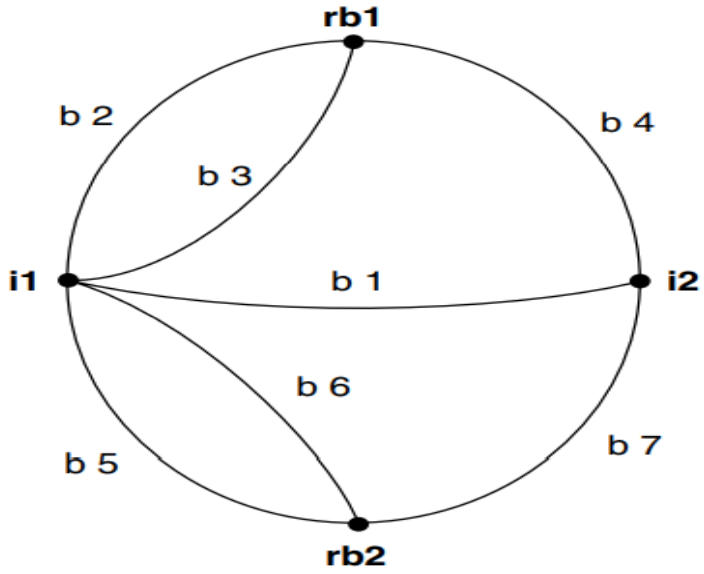
Analysis of bridges of Königsberg problem

- The riverbanks (rb1 and rb2) and islands (i1 and i2) are described by the nodes of a graph; the bridges are represented by labeled arcs between nodes (b1, b2, , b7).
- The graph representation preserves the essential structure of the bridge system, while ignoring extraneous features such as bridge lengths, distances, and order of bridges in the walk.
- The Königsberg bridge system using predicate calculus. The connect predicate corresponds to an arc of the graph, asserting that two land masses are connected by a particular bridge. Each bridge requires two connect predicates, one for each direction in which the bridge may be crossed. A predicate expression, connect(X, Y, Z) = connect (Y, X, Z), indicating that any bridge can be crossed in either direction, would allow removal of half the following connect facts:

20AM3602

connect facts

connect(i1, i2, b1)	connect(i2, i1, b1)
connect(rb1, i1, b2)	connect(i1, rb1, b2)
connect(rb1, i1, b3)	connect(i1, rb1, b3)
connect(rb1, i2, b4)	connect(i2, rb1, b4)
connect(rb2, i1, b5)	connect(i1, rb2, b5)
connect(rb2, i1, b6)	connect(i1, rb2, b6)
connect(rb2, i2, b7)	connect(i2, rb2, b7)



Euler noted that unless a graph contained either exactly zero or two nodes of odd degree, the walk was impossible.

If there were two odd-degree nodes, the walk could start at the first and end at the second; if there were no nodes of odd degree, the walk could begin and end at the same node.

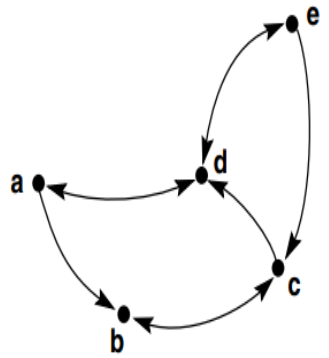
The walk is not possible for graphs containing any other number of nodes of odd degree, as is the case with the city of Königsberg. This problem is now called finding an Euler path through a graph.

20AM3602

Graph Theory

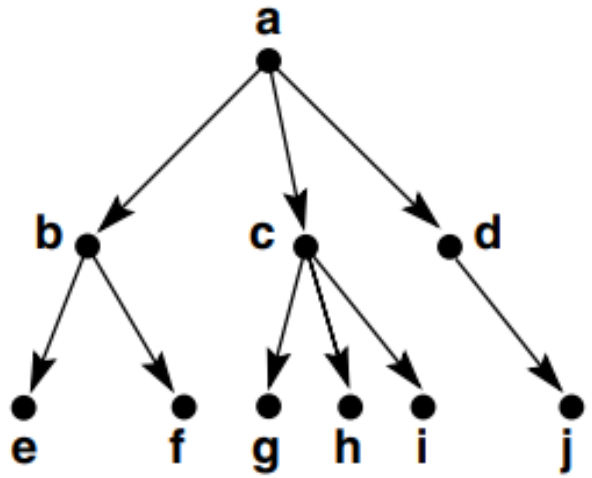
- A graph is a set of nodes or states and a set of arcs that connect the nodes.
- A labeled graph has one or more descriptors (labels) attached to each node that distinguish that node from any other node in the graph.
- In a state space graph, these **descriptors** identify states in a problem-solving process. If there are no descriptive differences between two nodes, they are considered the same.
- The arc between two nodes is indicated by the labels of the connected nodes.
- The arcs of a graph may also be labeled. Arc labels are used to indicate that an arc represents a named relationship (as in a semantic network) or to attach weights to arcs (as in the traveling salesperson problem).
- If there are different arcs between the same two nodes, these can also be distinguished through labeling.

- A graph is ***directed*** if arcs have an associated directionality.
- Arcs that can be crossed in either direction may have two arrows attached but more often have no direction indicators at all.
- Directed graph: arc (a, b) may only be crossed from node a to node b, but arc (b, c) is crossable in either direction.
- A ***path*** through a graph connects a sequence of nodes through successive arcs. The path is represented by an ordered list that records the nodes in the order they occur in the path. [a, b, c, d] represents the path through nodes a, b, c, and d, inthat order.
- A **rooted graph** has a unique node, called the root, such that there is a path from the root to all nodes within the graph. In drawing a rooted graph, the root is usually drawn at the top of the page, above the other nodes. The state space graphs for games are usually rooted graphs with the start of the game as the root.



Nodes = {a,b,c,d,e}

Arcs = {(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}



- A ***tree*** is a graph in which two nodes have at most one path between them. Trees often have roots, in which case they are usually drawn with the root at the top, like a rooted graph. Because each node in a tree has only one path of access from any other node, it is impossible for a path to loop or cycle through a sequence of nodes.
- For **rooted trees** or graphs, relationships between nodes include parent, child, and sibling. These are used in the usual familial fashion with the parent preceding its child along a directed arc. The children of a node are called siblings. Similarly, an ancestor comes before a descendant in some path of a directed graph.
- In example b is a parent of nodes e and f (which are, therefore, children of b and siblings of each other). Nodes a and c are ancestors of states g, h, and i, and g, h, and i are descendants of a and c.

20AM3602

Strategies for State Space Search

Data-Driven and Goal-Driven Search

A state space may be searched in two directions: from the given data of a **problem instance toward a goal** or **from a goal back to the data**.

In **data-driven search**, sometimes called forward chaining, the problem solver begins with the given facts of the problem and a set of legal moves or rules for changing state. Search proceeds by applying rules to facts to produce new facts, which are in turn used by the rules to generate more new facts. This process continues until it generates a path that satisfies the goal condition.

Goal- Driven Search: The goal that we want to solve. See what rules or legal moves could be used to generate this goal and determine what conditions must be true to use them. These conditions become the new goals, or subgoals, for the search. Search continues, working backward through successive subgoals until it works back to the facts of the problem. This finds the chain of moves or rules leading from data to a goal, although it does so in backward order. This approach is also called as **backward chaining,** and it recalls the simple childhood trick of trying to solve a maze by working back from the finish to the start.

Data-driven reasoning takes the facts of the problem and applies the rules or legal moves to produce new facts that lead to a goal;

Goal-driven reasoning focuses on the goal, finds the rules that could produce the goal, and chains backward through successive rules and subgoals to the given facts of the problem.

In the final analysis, both data-driven and goal-driven problem solvers search the same state space graph; however, the order and actual number of states searched can differ.

The preferred strategy is determined by the properties of the problem itself. These include the **complexity of the rules**, the **"shape"** of the state space, and the **nature** and **availability** of the problem data.

“I am a descendant of Thomas Jefferson.”

A solution is a path of direct lineage between the “I” and Thomas Jefferson.

This space may be searched in **two directions**, starting with the “I” and working along ancestor lines to Thomas Jefferson or starting with Thomas Jefferson and working through his descendants.

20AM3602

Depth-First and Breadth-First Search

Assignment work

Using state space to represent Reasoning with predicate calculus Recursion-based search

20AM3602

1. State Space Description of a Logic System

The first example of how a set of logic relationships may be viewed as defining a graph is from the propositional calculus. If p, q, r,... are propositions, assume the assertions:

q → p

r → p

v → q

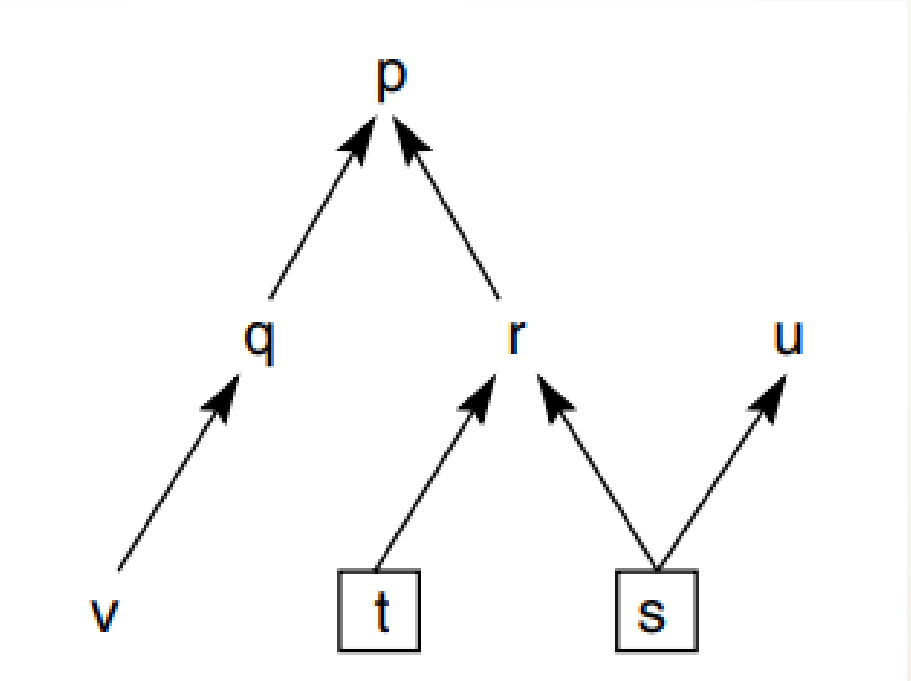
s → r

t → r

s → u

s

t



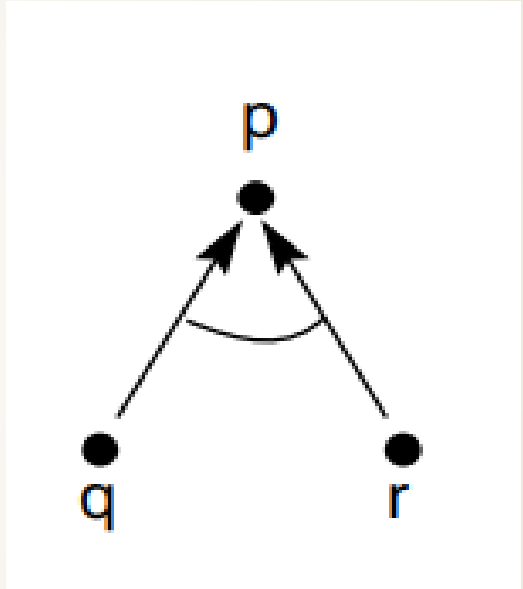
2. And/Or Graphs

In the propositional calculus: all of the assertions were implications of the form p → q.

Expressing the logic relationships defined by these operators requires an extension to the basic graph model which calls the and/or graph.

And/or graphs are an important tool for describing the search spaces generated by many AI problems, including those solved by logic-based theorem provers and expert systems.

The link connecting the arcs captures the idea that both q and r must be **true** to prove p. If the premises are connected by an or operator, they are regarded as or nodes in the graph. Arcs from or nodes to their parent node are not so connected. This captures the notion that the truth of any one of the premises is independently sufficient to determine the truth of the conclusion. An and/or graph is actually a specialization of a type of graph known as a hypergraph, which connects nodes by sets of arcs rather than by single arcs.

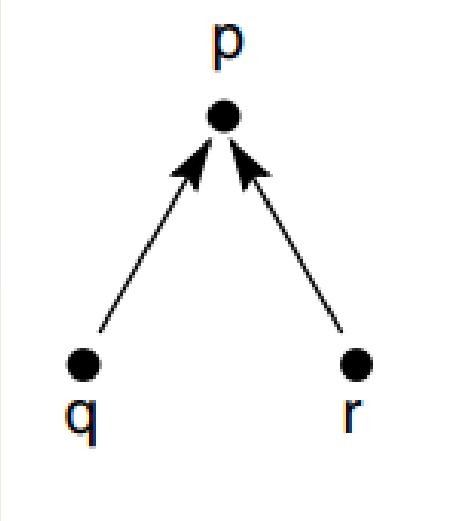


And/or graph of the expression q ∧ r → p.

Hypergraph

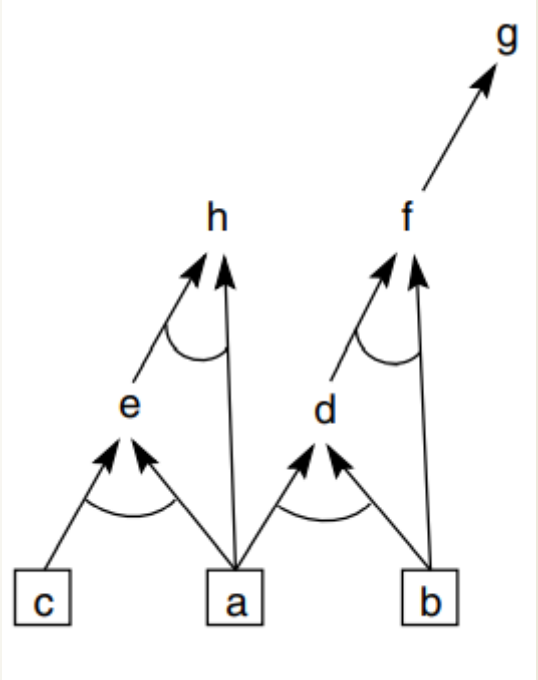
20AM3602

- A hypergraph consists of:
- N, a set of nodes.
- H, a set of hyperarcs defined by ordered pairs in which the first element of the pair is a single node from N and the second element is a subset of N.
- An ordinary graph is a special case of hypergraph in which all the sets of descendant nodes have a cardinality of 1.
- Hyperarcs are also known as k-connectors, where k is the cardinality of the set of descendant nodes. If k = 1, the descendant is thought of as an **OR** node. If k > 1, the elements of the set of descendants may be thought of as **AND** nodes.
- In this case, the connector is drawn between the individual edges from the parent to each of the descendant nodes.



And/or graph of the expression q v r → p

a
b
c
a ∧ b → d
a ∧ c → e
b ∧ d → f
f → g
a ∧ e → h



And/or graph of a set of propositional calculus expressions.

Production systems

20AM3602

- Knowledge representation formalism consists of collections of condition-action rules(Production Rules or Operators), a database which is modified in accordance with the rules, and a Production System Interpreter which controls the operation of the rules i.e The **control mechanism** of a Production System, determining the order in which Production Rules are fired.
- **A system that uses this form of knowledge representation is called a production system.**
- A production system consists of rules and factors. Knowledge is encoded in a declarative from which comprises of a set of rules of the form
- Situation ---------- Action
- SITUATION that implies ACTION.

The major components of an AI production system are

- i. A global database
- ii. A set of production rules and
- iii. A control system
- The **goal database** is the central data structure used by an AI production system.
- The **production rules** operate on the global database. Each rule has a precondition that is either satisfied or not by the database.
- If the **precondition** is satisfied, the rule can be applied. Application of the rule changes the database.
- The **control system** chooses which applicable rule should be applied and ceases computation when a termination condition on the database is satisfied. If several rules are to fire at the same time, the control system resolves the conflicts.

Four classes of production systems:-

1. A monotonic production system
2. A non monotonic production system
3. A partially commutative production system
4. A commutative production system.

- **Advantages of production systems:-**

1. Production systems provide an excellent tool for structuring AI programs.

2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.

3. The production rules are expressed in a natural form, so the statements contained in the knowledge base should the a recording of an expert thinking out loud.

- **Disadvantages of Production Systems:-**

1. One important disadvantage is the fact that it may be very difficult analyse the flow of control within a production system because the individual rules don't call each other.

2. Production systems describe the operations that can be performed in a search for a solution to the problem. They can be classified as follows.

3. Monotonic production system :- A system in which the application of a rule never prevents the later application of another rule, that could have also been applied at the time the first rule was selected.

20AM3602

Predicate Calculus and Planning

- Planning method based on logic

1. Propositional calculus
2. predicate calculus
3. Situation calculus
4. Event calculus

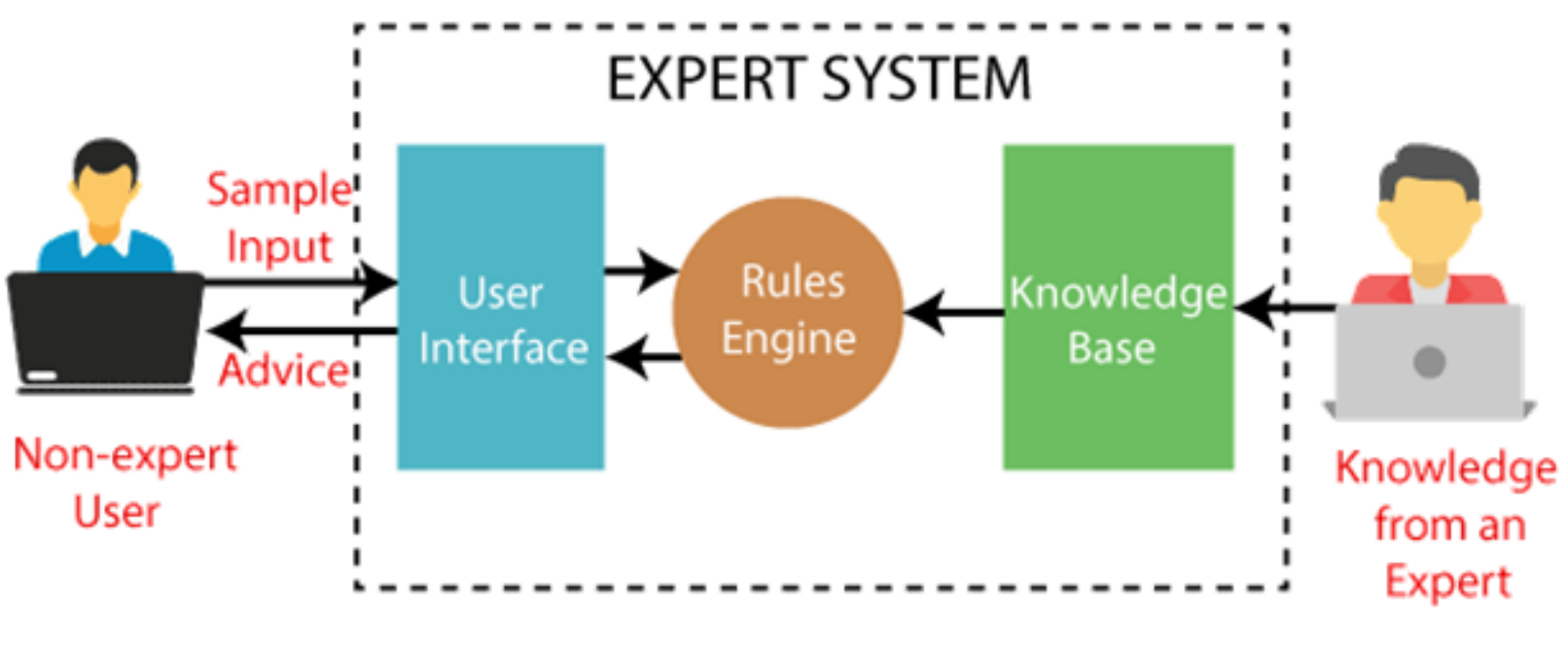
20AM3602

Expert System

- An expert system is a computer program that is designed to solve complex problems and to provide decision-making ability like a human expert.
- It performs this by extracting knowledge from its knowledge base using the reasoning and inference rules according to the user queries.
- This system helps in decision making for complex problems using **both facts and heuristics like a human expert**. It is called so because it contains the expert knowledge of a specific domain and can solve any complex problem of that particular domain.
- These systems are designed for a specific domain, such as **medicine, science,** etc.
- The performance of an expert system is based on the expert's knowledge stored in its knowledge base. The more knowledge stored in the KB, the more that system improves its performance.
- One of the common examples of an ES is a suggestion of spelling errors while typing in the Google search box.

20AM3602

Block diagram that represents the working of an expert system



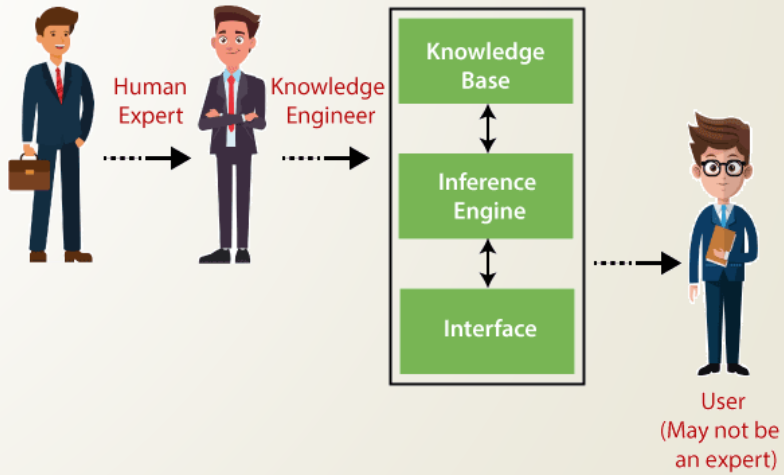
20AM3602

Characteristics of Expert System

- **High Performance:** The expert system provides high performance for solving any type of complex problem of a specific domain with high efficiency and accuracy.
- **Understandable:** It responds in a way that can be easily understandable by the user. It can take input in human language and provides the output in the same way.
- **Reliable:** It is much reliable for generating an efficient and accurate output.
- **Highly responsive:** ES provides the result for any complex query within a very short period of time.

An expert system mainly consists of three components:

- User Interface
- Inference Engine
- Knowledge Base



Components of ES:

20AM3602

- **User Interface:** it is an interface that helps a non-expert user to communicate with the expert system to find a solution.
- **Inference Engine(Rules of Engine):** The inference engine is known as the brain of the expert system as it is the main processing unit of the system. There are two types of inference engine:
  - Deterministic Inference engine: The conclusions drawn from this type of inference engine are assumed to be true. It is based on facts and rules.
  - Probabilistic Inference engine: This type of inference engine contains uncertainty in conclusions, and based on the probability.
  - Inference engine uses the below modes to derive the solutions:
  - **Forward Chaining:** It starts from the known facts and rules, and applies the inference rules to add their conclusion to the known facts.
  - **Backward Chaining:** It is a backward reasoning method that starts from the goal and works backward to prove the known facts.
- Knowledge Base: The knowledgebase is a type of storage that stores knowledge acquired from the different experts of the particular domain.
  - **Factual Knowledge:** The knowledge which is based on facts and accepted by knowledge engineers comes under factual knowledge.
  - **Heuristic Knowledge:** This knowledge is based on practice, the ability to guess, evaluation, and experiences.

Characteristics of an Expert System :

- Human experts are perishable, but an expert system is permanent.
- It helps to distribute the expertise of a human.
- One expert system may contain knowledge from more than one human experts thus making the solutions more efficient.
- It decreases the cost of consulting an expert for various domains such as medical diagnosis.
- They use a knowledge base and inference engine.
- Expert systems can solve complex problems by deducing new facts through existing facts of knowledge, represented mostly as if-then rules rather than through conventional procedural code.
- Expert systems were among the first truly successful forms of artificial intelligence (AI) software.

- Limitations :

- Do not have human-like decision-making power.
- Cannot possess human capabilities.
- Cannot produce correct result from less amount of knowledge.
- Requires excessive training.
- Advantages :

- Low accessibility cost.
- Fast response.
- Not affected by emotions, unlike humans.
- Low error rate.
- Capable of explaining how they reached a solution.

Disadvantages :

The expert system has no emotions.
Common sense is the main issue of the expert system.
It is developed for a specific domain.
It needs to be updated manually. It does not learn itself.
Not capable to explain the logic behind the decision.

20AM3602

Applications

- Different types of medical diagnosis like internal medicine, blood diseases and show on.
- Diagnosis of the complex electronic and electromechanical system.
- Diagnosis of a software development project.
- Planning experiment in biology, chemistry and molecular genetics.
- Forecasting crop damage.
- Diagnosis of the diesel-electric locomotive system.
- Identification of chemical compound structure.
- Scheduling of customer order, computer resources and various manufacturing task.
- Assessment of geologic structure from dip meter logs.
- Assessment of space structure through satellite and robot.
- The design of VLSI system.
- Teaching students specialize task.
- Assessment of log including civil case evaluation, product liability etc.

20AM3602

Assignment

- https://www.geeksforgeeks.org/difference-between-propositional-logic-and-predicate-logic/
- https://people.cs.pitt.edu/~milos/courses/cs1571-Fall06/lectures/Class20.pdf
- Production Systems : https://www.aalimec.ac.in/wp-content/uploads/2019/09/CS6659-AI_NOV2017QB.docx
- https://www.lancaster.ac.uk/stor-i-student-sites/harini-jayaraman/konigsberg-bridge-problem-and-the-evolution-of-mathematics/
- About Predicate Calculus & planning


Thank
You