A.I & ML

[R20A0566]

LECTURE NOTES

B.TECH III YEAR–II SEM
(R20) (2022-2023)

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

(Autonomous Institution – UGC, Govt. of India)

Recognized under 2(f) and 12 (B) of UGC ACT 1956

(Affiliated to JNTUH, Hyderabad, Approved by AICTE - Accredited by NBA & NAAC – 'A' Grade - ISO 9001:2015 Certified)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad – 500100, Telangana State, India

MALLA REDDY COLLEGE OF ENGINEERING & TECHNOLOGY

III YEAR B. Tech EEE– II SEM **L/T/P/C 3/-/-/3**

(R20A0566) ARTIFICIAL INTELLIGENCE & MACHINE LEARNING

COURSE OBJECTIVES:

- **To train the students to understand different types of AI agents.**
- **To understand various AI search algorithms.**
- **Fundamentals of knowledge representation, building of simple knowledge-based systems and to apply knowledge representation.**
- **To introduce the basic concepts and techniques of machine learning and the need for Machine learning techniques for real world problem**
- **To provide understanding of various Machine learning algorithms and the way to evaluate the performance of ML algorithms**

UNIT - I:

Introduction: AI problems, Agents and Environments, Structure of Agents, Problem

Solving Agents Basic Search Strategies: Problem Spaces, Uninformed Search (Breadth-First, Depth-First Search, Depth-first with Iterative Deepening), Heuristic Search (Hill

Climbing, Generic Best-First, A*), Constraint Satisfaction (Backtracking, Local Search)

UNIT - II:

Advanced Search: Constructing Search Trees, Stochastic Search, AO* Search Implementation, Minimax Search, Alpha-Beta Pruning Basic Knowledge Representation and Reasoning: Propositional Logic, First-Order Logic, Forward Chaining and Backward Chaining, Introduction to Probabilistic Reasoning, Bayes Theorem

UNIT - III:

Machine-Learning: Introduction. Machine Learning Systems, Forms of Learning: Supervised and Unsupervised Learning, reinforcement – theory of learning – feasibility of learning – Data Preparation– training versus testing and split.

UNIT - IV:

Supervised Learning:

Regression: Linear Regression, multi linear regression, Polynomial Regression, logistic regression, Non-linear Regression, Model evaluation methods. Classification: – support vector machines (SVM), Naïve Bayes classification

UNIT - V:

Unsupervised learning

Nearest neighbor models – K-means – clustering around medoids – silhouettes – hierarchical clustering – k-d trees, Clustering trees – learning ordered rule lists – learning unordered rule.

Reinforcement learning- Example: Getting Lost -State and Action Spaces

TEXT BOOKS:

1. Russell, S. and Norvig, P, Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall, 2010.

2. MACHINE LEARNING An Algorithmic Perspective 2nd Edition,

StephenMarsland,2015, by Taylor & Francis Group, LLC

3. Introduction to Machine Learning, The Wikipedia Guide

REFERENCES:

1. Artificial Intelligence, Elaine Rich, Kevin Knight, Shivasankar B. Nair, The McGraw Hill publications, Third Edition, 2009. 2. George F. Luger,

2. Artificial Intelligence: Structures and Strategies for Complex Problem Solving, Pearson Education, 6th ed., 2009.

3. Introduction to Machine Learning, Second Edition, Ethem Alpaydın, the MIT Press, Cambridge, Massachusetts, London, England.

4. Machine Learning, Tom M. Mitchell, McGraw-Hill Science, ISBN: 0070428077

5. Understanding Machine Learning:From Theory to Algorithms, c 2014 by ShaiShalev-Shwartz and Shai Ben-David, Published 2014 by Cambridge University Press.

COURSE OUTCOMES:

1. Understand the informed and uninformed problem types and apply search strategies to solve them.

2. Apply difficult real- l i f e problems in a state space representation so as to solvethose using AI techniques like searching and game playing.

3. Apply machine learning techniques in the design of computer systems

4. To differentiate between various categories of ML algorithms

5. Design and make modifications to existing machine learning algorithms.

INDEX

UNIT I:

Problem Solving by Search-I: Introduction to AI, Intelligent Agents

Problem Solving by Search –II: Problem-Solving Agents, Searching for Solutions, Uninformed Search Strategies: Breadth-first search, Uniform cost search, Depth-first search, Iterative deepening Depth-first search, Bidirectional search, Informed (Heuristic) Search Strategies: Greedy best-first search, A* search, Heuristic Functions, Beyond Classical Search: Hill-climbing search, Simulated annealing search, Local Search in Continuous Spaces.

Introduction:

- Artificial Intelligence is concerned with the design of intelligence in an artificial device. The term was coined by John McCarthy in 1956.
- Intelligence is the ability to acquire, understand and apply the knowledge to achieve goals in the world.
- AI is the study of the mental faculties through the use of computational models
- AI is the study of intellectual/mental processes as computational processes.
- AI program will demonstrate a high level of intelligence to a degree that equals or exceeds the intelligence required of a human in performing some task.
- AI is unique, sharing borders with Mathematics, Computer Science, Philosophy, Psychology, Biology, Cognitive Science and many others.
- Although there is no clear definition of AI or even Intelligence, it can be described as an attempt to build machines that like humans can think and act, able to learn and use knowledge to solve problems on their own.

History of AI:

Important research that laid the groundwork for AI:

- In 1931, Goedel layed the foundation of Theoretical Computer Science**1920-30s**: He published the first universal formal language and showed that math itself is either flawed or allows for unprovable but true statements.
- In 1936, Turing reformulated Goedel's result and church's extension thereof.

- In 1956, John McCarthy coined the term "Artificial Intelligence" as the topic of the **Dartmouth Conference**, the first conference devoted to the subject.
- In 1957, The **General Problem Solver (GPS)** demonstrated by Newell, Shaw & Simon
- In 1958, John McCarthy (MIT) invented the Lisp language.
- In 1959, Arthur Samuel (IBM) wrote the first game-playing program, for checkers, to achieve sufficient skill to challenge a world champion.
- In 1963, Ivan Sutherland's MIT dissertation on Sketchpad introduced the idea of interactive graphics into computing.
- In 1966, Ross Quillian (PhD dissertation, Carnegie Inst. of Technology; now CMU) demonstrated semantic nets
- In 1967, Dendral program (Edward Feigenbaum, Joshua Lederberg, Bruce Buchanan, Georgia Sutherland at Stanford) demonstrated to interpret mass spectra on organic chemical compounds. First successful knowledge-based program for scientific reasoning.
- In 1967, Doug Engelbart invented the mouse at SRI
- In 1968, Marvin Minsky & Seymour Papert publish Perceptrons, demonstrating limits of simple neural nets.
- In 1972, Prolog developed by Alain Colmerauer.
- In Mid 80's, Neural Networks become widely used with the Backpropagation algorithm (first described by Werbos in 1974).
- 1990, Major advances in all areas of AI, with significant demonstrations in machine learning, intelligent tutoring, case-based reasoning, multi-agent planning, scheduling, uncertain reasoning, data mining, natural language understanding and translation, visio n, virtual reality, games, and other topics.
- In 1997, Deep Blue beats the World Chess Champion Kasparov
- In 2002, iRobot, founded by researchers at the MIT Artificial Intelligence Lab, introduced **Roomba**, a vacuum cleaning robot. By 2006, two million had been sold.

Foundations of Artificial Intelligence:

- **Philosophy**

e.g., foundational issues (can a machine think?), issues of knowledge and believe, mutual knowledge

- **Psychology and Cognitive Science**
  e.g., problem solving skills
- **Neuro-Science**
  e.g., brain architecture
- **Computer Science And Engineering**
  e.g., complexity theory, algorithms, logic and inference, programming languages, and system building.
- **Mathematics and Physics**
  e.g., statistical modeling, continuous mathematics,
- **Statistical Physics, and Complex Systems**.

Sub Areas of AI:

1) **Game Playing**
   Deep Blue Chess program beat world champion Gary Kasparov
2) **Speech Recognition**
   PEGASUS spoken language interface to American Airlines' EAASY SABRE reseration system, which allows users to obtain flight information and make reservations over the telephone. The 1990s has seen significant advances in speech recognition so that limited systems are now successful.
3) **Computer Vision**
   Face recognition programs in use by banks, government, etc. The ALVINN system from CMU autonomously drove a van from Washington, D.C. to San Diego (all but 52 of 2,849 miles), averaging 63 mph day and night, and in all weather conditions. Handwriting recognition, electronics and manufacturing inspection, photo interpretation, baggage inspection, reverse engineering to automatically construct a 3D geometric model.

4) **Expert Systems**

Application-specific systems that rely on obtaining the knowledge of human experts in an area and programming that knowledge into a system.

a. **Diagnostic Systems** : MYCIN system for diagnosing bacterial infections of the blood and suggesting treatments. Intellipath pathology diagnosis system (AMA approved). Pathfinder medical diagnosis system, which suggests tests and makes diagnoses. Whirlpool customer assistance center.

b. System Configuration

DEC's XCON system for custom hardware configuration. Radiotherapy treatment planning.

c. Financial Decision Making

Credit card companies, mortgage companies, banks, and the U.S. government employ AI systems to detect fraud and expedite financial transactions. For example, AMEX credit check.

d. Classification Systems

Put information into one of a fixed set of categories using several sources of information. E.g., financial decision making systems. NASA developed a system for classifying very faint areas in astronomical images into either stars or galaxies with very high accuracy by learning from human experts' classifications.

5) **Mathematical Theorem Proving**

Use inference methods to prove new theorems.

6) **Natural Language Understanding**

AltaVista's translation of web pages. Translation of Catepillar Truck manuals into 20 languages.

7) **Scheduling and Planning**

Automatic scheduling for manufacturing. DARPA's DART system used in Desert Storm and Desert Shield operations to plan logistics of people and supplies. American Airlines rerouting contingency planner. European space agency planning and scheduling of spacecraft assembly, integration and verification.

8) **Artificial Neural Networks:**

9) **Machine Learning**

Application of AI:

AI algorithms have attracted close attention of researchers and have also been applied successfully to solve problems in engineering. Nevertheless, for large and complex problems, AI algorithms consume considerable computation time due to stochastic feature of the search approaches

1) Business; financial strategies

2) Engineering: check design, offer suggestions to create new product, expert systems for all engineering problems

3) Manufacturing: assembly, inspection and maintenance

4) Medicine: monitoring, diagnosing

5) Education: in teaching

6) Fraud detection

7) Object identification

8) Information retrieval

9) Space shuttle scheduling

Building AI Systems:

1) **Perception**

Intelligent biological systems are physically embodied in the world and experience the world through their sensors (senses). For an autonomous vehicle, input might be images from a camera and range information from a rangefinder. For a medical diagnosis system, perception is the set of symptoms and test results that have been obtained and input to the system manually.

2) **Reasoning**

Inference, decision-making, classification from what is sensed and what the internal "model" is of the world. Might be a neural network, logical deduction system, Hidden Markov Model induction, heuristic searching a problem space, Bayes Network inference, genetic algorithms, etc.

Includes areas of knowledge representation, problem solving, decision theory, planning, game theory, machine learning, uncertainty reasoning, etc.

3) **Action**

Biological systems interact within their environment by actuation, speech, etc. All behavior is centered around actions in the world. Examples include controlling the steering of a Mars rover or autonomous vehicle, or suggesting tests and making diagnoses for a medical diagnosis system. Includes areas of robot actuation, natural language generation, and speech synthesis.

The definitions of AI:

a) "The exciting new effort to make computers think . . . *machines with minds,* in the full and literal sense" (Haugeland, 1985) "The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning..."(Bellman, 1978)	b) "The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
c) "The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1 99 1)	d) "A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1 990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

The definitions on the top, **(a)** and **(b)** are concerned with **reasoning**, whereas those on the bottom**, (c)** and **(d)** address **behavior.**The definitions on the left, **(a)** and **(c)** measure success in terms of human performance, and those on the right, **(b)** and **(d)** measure the ideal concept of intelligence called rationality

Intelligent Systems:

In order to design intelligent systems, it is important to categorize them into four categories (Luger and Stubberfield 1993), (Russell and Norvig, 2003)

1. Systems that think like humans
2. Systems that think rationally
3. Systems that behave like humans
4. Systems that behave rationally

	Human- Like	**Rationally**
Think:	**Cognitive Science Approach** ***"Machines that think like humans"***	**Laws of thought Approach** ***" Machines that think Rationally"***
Act:	**Turing Test Approach** ***"Machines that behave like humans"***	**Rational Agent Approach** ***"Machines that behave Rationally"***

Scientific Goal: To determine which ideas about knowledge representation, learning, rule systems search, and so on, explain various sorts of real intelligence.

Engineering Goal: To solve real world problems using AI techniques such as Knowledge representation, learning, rule systems, search, and so on.

Traditionally, computer scientists and engineers have been more interested in the engineering goal, while psychologists, philosophers and cognitive scientists have been more interested in the scientific goal.

Cognitive Science: Think Human-Like

a. Requires a model for human cognition. Precise enough models allow simulation by computers.

b. Focus is not just on behavior and I/O, but looks like reasoning process.

c. Goal is not just to produce human-like behavior but to produce a sequence of steps of the reasoning process, similar to the steps followed by a human in solving the same task.

Laws of thought: Think Rationally

a. The study of mental faculties through the use of computational models; that it is, the study of computations that make it possible to perceive reason and act.

b. Focus is on inference mechanisms that are probably correct and guarantee an optimal solution.

c. Goal is to formalize the reasoning process as a system of logical rules and procedures of inference.

d. Develop systems of representation to allow inferences to be like

"*Socrates is a man. All men are mortal. Therefore Socrates is mortal"*

Turing Test: Act Human-Like

a. The art of creating machines that perform functions requiring intelligence when performed by people; that it is the study of, how to make computers do things which, at the moment, people do better.

b. Focus is on action, and not intelligent behavior centered around the representation of the world

c. Example: Turing Test

- 3 rooms contain: a person, a computer and an interrogator.
- The interrogator can communicate with the other 2 by teletype (to avoid the machine imitate the appearance of voice of the person)
- The interrogator tries to determine which the person is and which the machine is.
- The machine tries to fool the interrogator to believe that it is the human, and the person also tries to convince the interrogator that it is the human.
- If the machine succeeds in fooling the interrogator, then conclude that the machine is intelligent.

Rational agent: Act Rationally

a. Tries to explain and emulate intelligent behavior in terms of computational process; that it is concerned with the automation of the intelligence.

b. Focus is on systems that act sufficiently if not optimally in all situations.

c. Goal is to develop systems that are rational and sufficient

The difference between strong AI and weak AI:

Strong AI makes the bold claim that computers can be made to think on a level (at least) equal to humans.

Weak AI simply states that some "thinking-like" features can be added to computers to make them more useful tools... and this has already started to happen (witness expert systems, drive-by-wire cars and speech recognition software).

AI Problems:

AI problems (speech recognition, NLP, vision, automatic programming, knowledge representation, etc.) can be paired with techniques (NN, search, Bayesian nets, production systems, etc.).AI problems can be classified in two types:

1. Common-place tasks(Mundane Tasks)
2. Expert tasks

Common-Place Tasks:

1. *Recognizing* people, objects.
2. Communicating (through *natural language*).
3. *Navigating* around obstacles on the streets.

These tasks are done matter of factly and routinely by people and some other animals.

Expert tasks:

1. Medical diagnosis.
2. Mathematical problem solving
3. Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them. The second range of tasks requires skill development and/or intelligence and only some specialists can perform them well. However, when we look at what computer systems have been able to achieve to date, we see that their achievements include performing sophisticated tasks like medical diagnosis, performing symbolic integration, proving theorems and playing chess.

1. *Intelligent Agent's:*

Agents and environments:

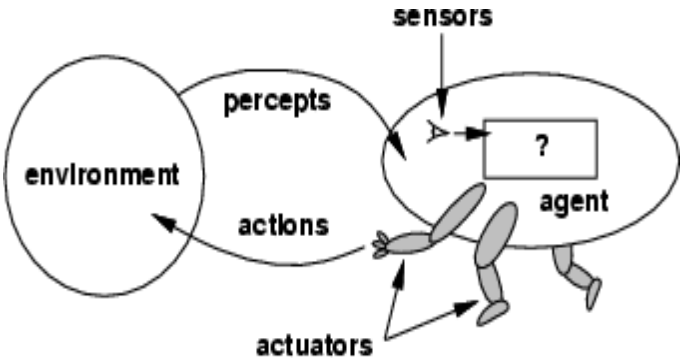


Fig 2.1: Agents and Environments

Agent:

An *Agent* is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators.

- ✓ A *human agent* has eyes, ears, and other organs for sensors and hands, legs, mouth, and other body parts for actuators.
- ✓ A *robotic agent* might have cameras and infrared range finders for sensors and various motors for actuators.
- ✓ A *software agent* receives keystrokes, file contents, and network packets as sensory inputs and acts on the environment by displaying on the screen, writing files, and sending network packets.

Percept:

We use the term percept to refer to the agent's perceptual inputs at any given instant.

Percept Sequence:

An agent's percept sequence is the complete history of everything the agent has ever perceived.

Agent function:

Mathematically speaking, we say that an agent's behavior is described by the agent function that maps any **given percept sequence to an action**.

Agent program

Internally, the agent function for an artificial agent will be implemented by an agent program. It is important to keep these two ideas distinct. The agent function is an abstract mathematical description; the agent program is a concrete implementation, running on the agent architecture.

To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in **Fig**

This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and whether there is dirt in the square. It can choose to move left, move right, suck up the dirt, or do nothing. One very simple agent function is the following: if the current square is dirty, then suck, otherwise move to the other square. A partial tabulation of this agent function is shown in **Fig 2.1.6**.

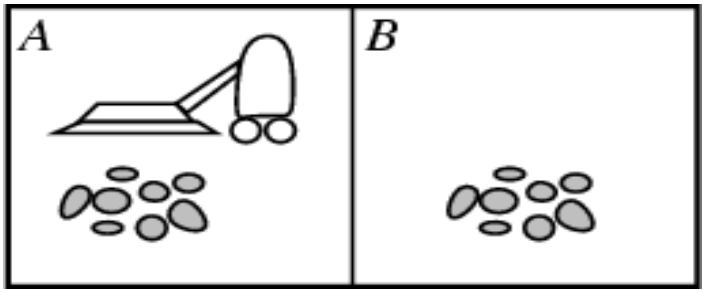


Fig 2.1.5: A vacuum-cleaner world with just two locations.

Agent function

Percept Sequence	**Action**
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right

[A, Clean], [A, Dirty]	Suck
…	

Fig 2.1.6: Partial tabulation of a simple agent function for the example: vacuum-cleaner world shown in the **Fig 2.1.5**

```
Function REFLEX-VACCUM-AGENT ([location, status]) returns an action If

status=Dirty then return Suck

else if location = A then return Right

else if location = B then return Left
```

Fig 2.1.6(i): The REFLEX-VACCUM-AGENT program is invoked for each new percept (location, status) and returns **an** action each time

Strategies of Solving Tic-Tac-Toe Game Playing

Tic-Tac-Toe Game Playing:

Tic-Tac-Toe is a simple and yet an interesting board game. Researchers have used various approaches to study the Tic-Tac-Toe game. For example, Fok and Ong and Grim et al. have used artificial neural network based strategies to play it. Citrenbaum and Yakowitz discuss games like Go-Moku, Hex and Bridg-It which share some similarities with Tic-Tac-Toe.

a

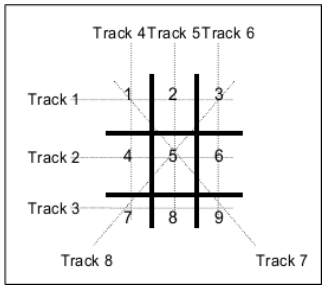


Fig 1.

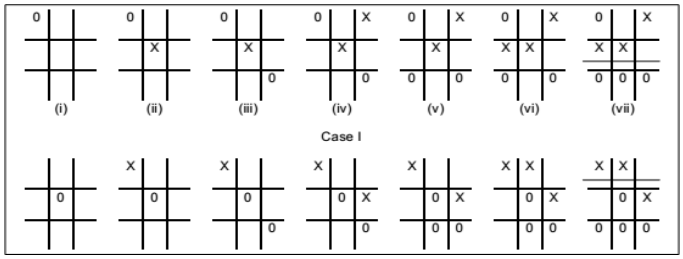
A Formal Definition of the Game:

The board used to play the Tic-Tac-Toe game consists of 9 cells laid out in the form of a 3x3 matrix (Fig. 1). The game is played by 2 players and either of them can start. Each of the two players is assigned a unique symbol (generally 0 and X). Each player alternately gets a turn to make a move. Making a move is compulsory and cannot be deferred. In each move a player places the symbol assigned to him/her in a hitherto blank cell.

Let a track be defined as any row, column or diagonal on the board. Since the board is a square matrix with 9 cells, all rows, columns and diagonals have exactly 3 cells. It can be easily observed that there are 3 rows, 3 columns and 2 diagonals, and hence a total of 8 tracks on the board (Fig. 1). The goal of the game is to fill all the three cells of any track on the board with the symbol assigned to one before the opponent does the same with the symbol assigned to him/her. At any point of the game, if there exists a track whose all three cells have been marked by the same symbol, then the player to whom that symbol have been assigned wins and the game terminates. If there exist no track whose cells have been marked by the same symbol when there is no more blank cell on the board then the game is drawn.

Let the priority of a cell be defined as the number of tracks passing through it. The priorities of the nine cells on the board according to this definition are tabulated in Table 1. Alternatively, let the priority of a track be defined as the sum of the priorities of its three cells. The priorities of the eight tracks on the board according to this definition are tabulated in Table 2. The prioritization of the cells and the tracks lays the foundation of the heuristics to be used in this study. These heuristics are somewhat similar to those proposed by Rich and Knight.

a



Strategy 1:

Algorithm:

1. View the vector as a ternary number. Convert it to a decimal number.

2. Use the computed number as an index into Move-Table and access the vector stored there.

3. Set the new board to that vector.

Procedure:

1) Elements of vector:

0: Empty

1: X

2: O

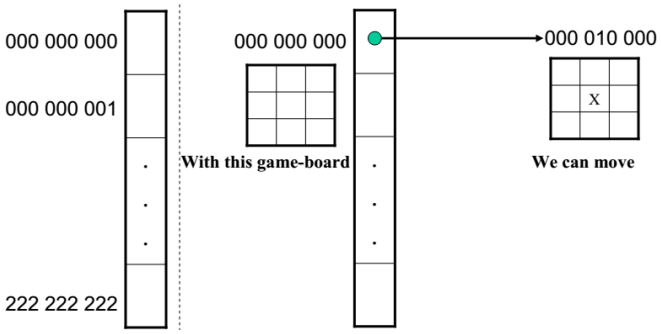
→ the vector is a ternary number

2) Store inside the program a move-table (lookuptable):

a) Elements in the table: 19683 (3^9)

b) Element = A vector which describes the most suitable move from the

❖ Data Structure:



Comments:

1. A lot of space to store the Move-Table.

2. A lot of work to specify all the entries in the Move-Table.

3. Difficult to extend

Explanation of Strategy 2 of solving Tic-tac-toe problem

Stratergy 2:

Data Structure:

1) Use vector, called board, as Solution 1
2) However, elements of the vector:

   2: Empty

   3: X

   5: O
3) Turn of move: indexed by integer

   1,2,3, etc

Function Library:

1. Make2:

   a) Return a location on a game-board.

      IF (board[5] = 2)

      RETURN 5; //the center cell.

      ELSE

      RETURN any cell that is not at the board's corner;

      // (cell: 2,4,6,8)

   b) Let P represent for X or O

   c) can_win(P) :

      P has filled already at least two cells on a straight line (horizontal, vertical, or diagonal)

   d) cannot_win(P) = NOT(can_win(P))

2. Posswin(P):

   IF (cannot_win(P))

   RETURN 0;

   ELSE

   RETURN index to the empty cell on the line of

   can_win(P)

a

Let odd numbers are turns of X

Let even numbers are turns of O

3. Go(n): make a move

Algorithm:

1. Turn = 1: (X moves)

Go(1) //make a move at the left-top cell

2. Turn = 2: (O moves)

IF board[5] is empty THEN

Go(5)

ELSE

Go(1)

3. Turn = 3: (X moves)

IF board[9] is empty THEN

Go(9)

ELSE

Go(3).

4. Turn = 4: (O moves)

IF Posswin (X) <> 0 THEN

Go (Posswin (X))

//Prevent the opponent to win

ELSE Go (Make2)

5. Turn = 5: (X moves)

IF Posswin(X) <> 0 THEN

Go(Posswin(X))

//Win for X.

ELSE IF Posswin(O) <> THEN

Go(Posswin(O))

//Prevent the opponent to win

ELSE IF board[7] is empty THEN

Go(7)

ELSE Go(3).

Comments:

1. Not efficient in time, as it has to check several conditions before making each move.
2. Easier to understand the program's strategy.
3. Hard to generalize.

Introduction to Problem Solving, General problem solving

Problem solving is a process of generating solutions from observed data.

−a problem is characterized by a set of goals,

−a set of objects, and

−a set of operations.

These could be ill-defined and may evolve during problem solving.

Searching Solutions:

To build a system to solve a problem:

1. Define the problem precisely
2. Analyze the problem
3. Isolate and represent the task knowledge that is necessary to solve the problem
4. Choose the best problem-solving techniques and apply it to the particular problem.

Defining the problem as State Space Search:

The state space representation forms the basis of most of the AI methods.

- Formulate a problem as a **state space search** by showing the legal problem states, the legal operators, and the initial and goal states.
- A **state** is defined by the specification of the values of all attributes of interest in the world
- An **operator** changes one state into the other; it has a precondition which is the value of certain attributes prior to the application of the operator, and a set of effects, which are the attributes altered by the operator
- The **initial state** is where you start
- The **goal state** is the partial description of the solution

Formal Description of the problem:

1. Define a state space that contains all the possible configurations of the relevant objects.

2. Specify one or more states within that space that describe possible situations from which the problem solving process may start **(initial state)**
3. Specify one or more states that would be acceptable as solutions to the problem**. (goal states)**

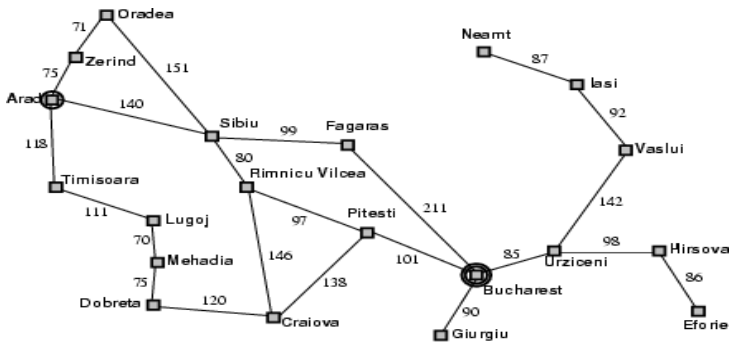
Specify a set of rules that describe the actions **(operations)** available

State-Space Problem Formulation:

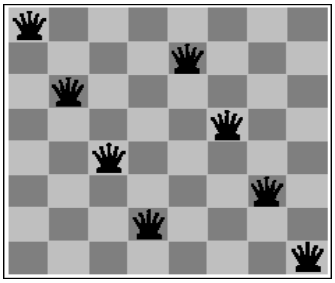
Example: A problem is defined by four items:

1. **initial state** e.g., "at Arad"
2. **actions or successor function** : *S(x)* = set of action–state pairs
   e.g., *S(Arad)* = {<*Arad* → *Zerind, Zerind*>, … }
3. **goal test (or set of goal states)**
   e.g., *x* = "at Bucharest", *Checkmate(x)*
4. **path cost (additive)**
   e.g., sum of distances, number of actions executed, etc.
   c(x,a,y) is the step cost, assumed to be ≥ 0

A solution is a sequence of actions leading from the initial state to a goal state



Example: 8-**queens problem**



1. **Initial State**: Any arrangement of 0 to 8 queens on board.

2. **Operators**: add a queen to any square.
3. **Goal Test**: 8 queens on board, none attacked.
4. **Path cost**: not applicable or Zero (because only the final state counts, search cost might be of interest).

State Spaces versus Search Trees:

- State Space
  - Set of valid states for a problem
  - Linked by operators
  - e.g., 20 valid states (cities) in the Romanian travel problem
- Search Tree
  - Root node = initial state
  - Child nodes = states that can be visited from parent
  - Note that the depth of the tree can be infinite
    - E.g., via repeated states
  - Partial search tree
    - Portion of tree that has been expanded so far
  - Fringe
    - Leaves of partial search tree, candidates for expansion

Search trees = data structure to search state-space

Properties of Search Algorithms

Which search algorithm one should use will generally depend on the problem domain.
There are four important factors to consider:

1. ***Completeness*** – Is a solution guaranteed to be found if at least one solution exists?

2. ***Optimality*** – Is the solution found guaranteed to be the best (or lowest cost) solution if there exists more than one solution?

3. ***Time Complexity*** – The upper bound on the time required to find a solution, as a function of the complexity of the problem.

4. ***Space Complexity*** – The upper bound on the storage space (memory) required at any point during the search, as a function of the complexity of the problem.

General problem solving, Water-jug problem, 8-puzzle problem

General Problem Solver*:*

The General Problem Solver (GPS) was the first useful AI program, written by Simon, Shaw, and Newell in 1959. As the name implies, it was intended to solve nearly any problem**.**

Newell and Simon defined each problem as a space. At one end of the space is the starting point; on the other side is the goal. The problem-solving procedure itself is conceived as a set of operations to cross that space, to get from the starting point to the goal state, one step at a time.

The General Problem Solver, the program tests various actions (which Newell and Simon called operators) to see which will take it closer to the goal state. An operator is any activity that changes the state of the system. The General Problem Solver always chooses the operation that appears to bring it closer to its goal.

Example: Water Jug Problem

Consider the following problem:

> A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?

State Representation and Initial State :

> We will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \leq x \leq 4$, and $0 \leq y \leq 3$. Our initial state: (0, 0)

Goal Predicate - state = (2, y) where $0 \leq y \leq 3$.

Operators -we must defi ne a set of operators that will take us from one state to another:

1. Fill 4-gal jug	(x,y) x < 4	→ (4,y)

2. Fill 3-gal jug	(x,y) y < 3	→ (x,3)
3. Empty 4-gal jug on ground	(x,y) x > 0	→ (0,y)
4. Empty 3-gal jug on ground	(x,y) y > 0	→ (x,0)
5. Pour water from 3-gal jug	(x,y)	→*!* (4, y - (4 - x))
to ll 4-gal jug	0 < x+y 4 and y > 0	
6. Pour water from 4-gal jug	(x,y)	→ (x - (3-y), 3)
to ll 3-gal-jug	0 < x+y 3 and x > 0	
7. Pour all of water from 3-gal jug	(x,y)	→ (x+y, 0)
into 4-gal jug	0 < x+y 4 and y 0	
8. Pour all of water from 4-gal jug	(x,y)	→ (0, x+y)
into 3-gal jug	0 < x+y 3 and x 0	

Through Graph Search, the following solution is found :

Gals in 4-gal jug	Gals in 3-gal jug	Rule Applied
0	0	
		1. Fill 4
4	0	
		6. Pour 4 into 3 to ll
1	3	
		4. Empty 3
1	0	
		8. Pour all of 4 into 3
0	1	
		1. Fill 4
4	1	
		6. Pour into 3
2	3	

Second Solution:

Number of Steps	*Rules applied*	*4-g jug*	*3-g jug*
1	**Initial State**	0	0
2	R2 {Fill 3-g jug}	0	3
3	R7 {Pour all water from 3 to 4-g jug }	3	0
4	R2 {Fill 3-g jug}	3	3
5	R5 {Pour from 3 to 4-g jug until it is full}	4	2
6	R3 {Empty 4-gallon jug}	0	2
7	R7 {Pour all water from 3 to 4-g jug}	2	0
			Goal State

Control strategies

Control Strategies means how to decide which rule to apply next during the process of searching for a solution to a problem.

Requirement for a good Control Strategy

1. **It should cause motion**

   In water jug problem, if we apply a simple control strategy of starting each time from the top of rule list and choose the first applicable one, then we will never move towards solution.

2. **It should explore the solution space in a systematic manner**

   If we choose another control strategy, let us say, choose a rule randomly from the applicable rules then definitely it causes motion and eventually will lead to a solution. But one may arrive to same state several times. This is because control strategy is not systematic.

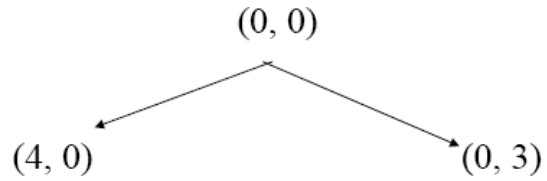
Systematic Control Strategies (Blind searches):

Breadth First Search:

Let us discuss these strategies using water jug problem. These may be applied to any search problem.

Construct a tree with the initial state as its root.

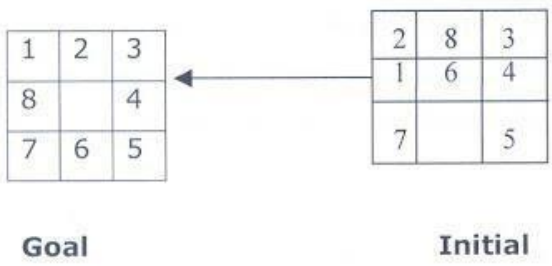
Generate all the offspring of the root by applying each of the applicable rules to the initial state.



Now for each leaf node, generate all its successors by applying all the rules that are appropriate.

8 Puzzle Problem.

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell. Such a puzzle is illustrated in following diagram.



The program is to change the initial configuration into the goal configuration. A solution to the problem is an appropriate sequence of moves, such as "move tiles 5 to the right, move tile 7 to the left, move tile 6 to the down, etc".

Solution:

To solve a problem using a production system, we must specify the global database the rules, and the control strategy. For the 8 puzzle problem that correspond to these three components. These elements are the problem states, moves and goal. In this problem each tile configuration is a state. The set of all configuration in the space of problem states or the problem space, there are only 3, 62,880 different configurations o the 8 tiles and blank space. Once the problem states have been conceptually identified, we must construct a computer representation, or description of them . this description is then used as the database of a production system. For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. The initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

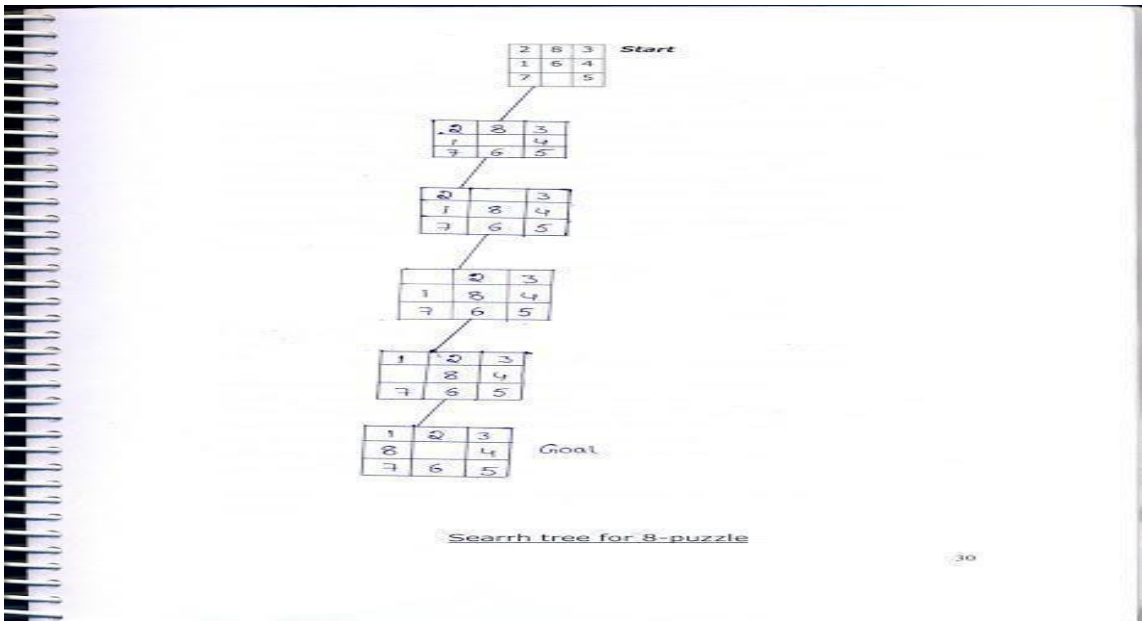
A move transforms one problem state into another state. The 8-puzzle is conveniently interpreted as having the following for moves. Move empty space (blank) to the left, move blank up, move blank to the right and move blank down,. These moves are modeled by production rules that operate on the state descriptions in the appropriate manner.

The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus the precondition for the rule associated with "move blank up" is derived from the requirement that the blank space must not already be in the top row.

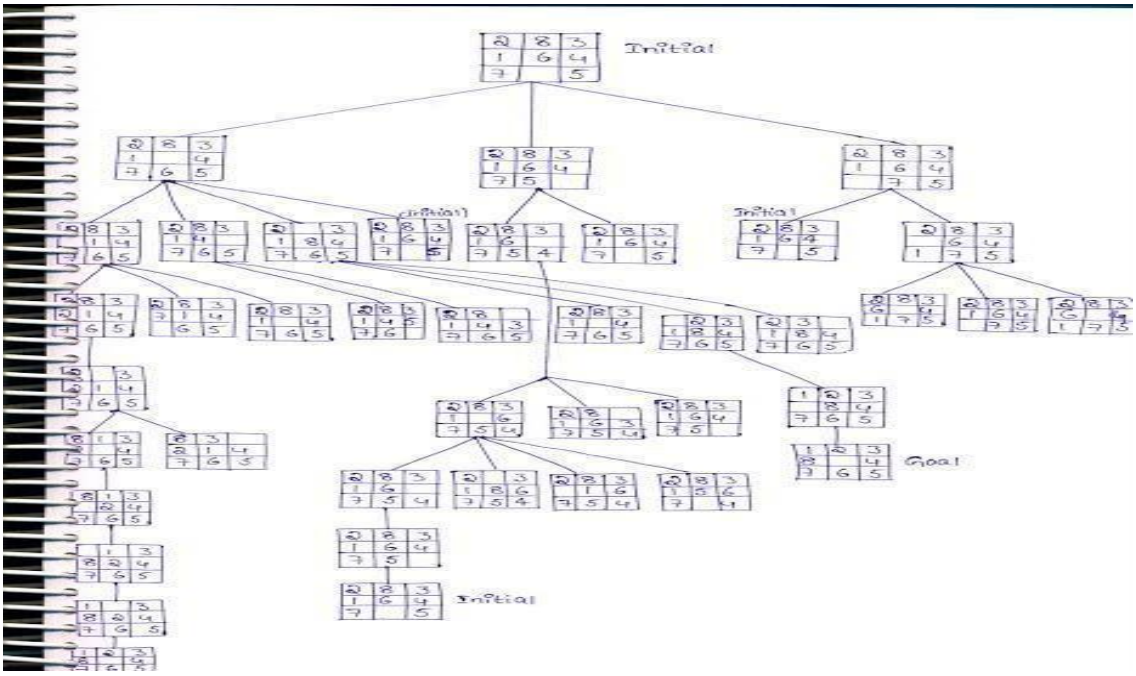
The problem goal condition forms the basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of rules that have been applied so that it can compose them into sequence representing the problem solution. A solution to the 8-puzzle problem is given in the following figure.

Example:- Depth – First – Search traversal and Breadth - First - Search traversal

for 8 – puzzle problem is shown in following diagrams.



a



Exhaustive Searches, BFS and DFS

Search is the systematic examination of states to find path from the start/root state to the goal state.

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed; using heuristic functions. The algorithms that use heuristic functions are called heuristic algorithms. Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.

Heuristic algorithms aremore efficient because they take advantage of feedback from the data to direct the search path.

Uninformed search

Also called blind, exhaustive or brute-force search, uses no information about the problem to guide the search and therefore may not be very efficient.

Informed Search:

Also called heuristic or intelligent search, uses information about the problem to guide the search, usually guesses the distance to a goal state and therefore efficient, but the search may not be always possible.

Uninformed Search Methods:

Breadth- First -Search:

Consider the state space of a problem that takes the form of a tree. Now, if we search the goal along each breadth of the tree, starting from the root and continuing up to the largest depth, we call it *breadth first search*.

- **Algorithm:**
    1. Create a variable called NODE-LIST and set it to initial state
    2. Until a goal state is found or NODE-LIST is empty do
        a. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit
        b. For each way that each rule can match the state described in E do:
            i. Apply the rule to generate a new state
            ii. If the new state is a goal state, quit and return this state
            iii. Otherwise, add the new state to the end of NODE-LIST

BFS illustrated:

Step 1: Initially fringe contains only one node corresponding to the source state A.

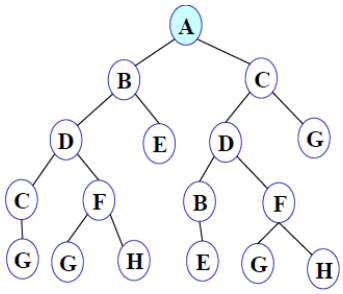


Figure 1

FRINGE: A

Step 2: A is removed from fringe. The node is expanded, and its children B and C are generated. They are placed at the back of fringe.

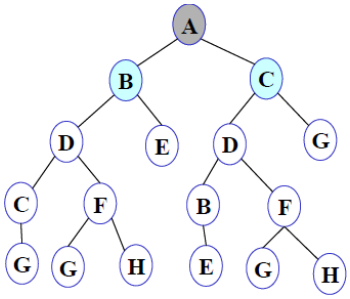


Figure 2

FRINGE: B C

Step 3: Node B is removed from fringe and is expanded. Its children D, E are generated and put at the back of fringe.

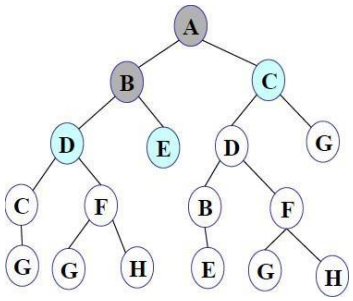


Figure 3

FRINGE: C D E

Step 4: Node C is removed from fringe and is expanded. Its children D and G are added to the back of fringe.

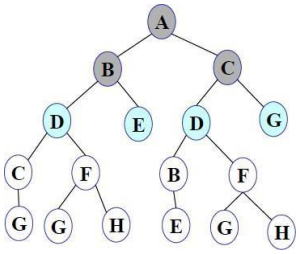


Figure 4

FRINGE: D E D G

Step 5: Node D is removed from fringe. Its children C and F are generated and added to the back of fringe.

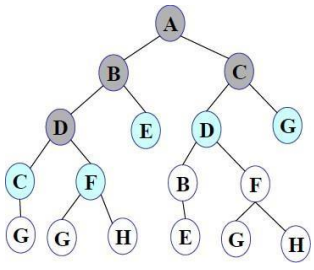


Figure 5

FRINGE: E D G C F

Step 6: Node E is removed from fringe. It has no children.

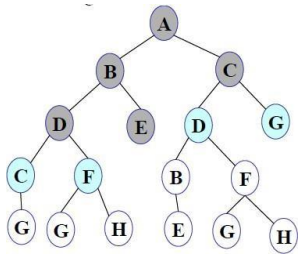


Figure 6

FRINGE: D G C F

Step 7: D is expanded; B and F are put in OPEN.

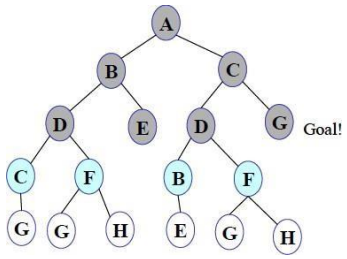


Figure 7

FRINGE: G C F B F

Step 8: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

Breadth first search is:

- One of the simplest search strategies
- Complete. If there is a solution, BFS is guaranteed to find it.
- If there are multiple solutions, then a minimal solution will be found
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- **Time complexity** : $O(b^d)$
- **Space complexity** : $O(b^d)$
- **Optimality** :Yes

b - branching factor(maximum no of successors of any node),

d – Depth of the shallowest goal node

Maximum length of any path (m) in search space

Advantages: Finds the path of minimal length to the goal.

Disadvantages:

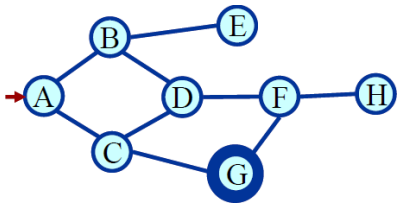
- Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node.
- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

Depth- First- Search.

We may sometimes search the goal along the largest depth of the tree, and move up only when further traversal along the depth is not possible. We then attempt to find alternative offspring of the parent of the node (state) last visited. If we visit the nodes of a tree using the above principles to search the goal, the traversal made is called depth first traversal and consequently the search strategy is called *depth first search*.

- **Algorithm:**
    1. Create a variable called NODE-LIST and set it to initial state
    2. Until a goal state is found or NODE-LIST is empty do
        a. Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit
        b. For each way that each rule can match the state described in E do:
            i. Apply the rule to generate a new state
            ii. If the new state is a goal state, quit and return this state
            iii. Otherwise, add the new state in front of NODE-LIST

DFS illustrated:



A State Space Graph

Step 1: Initially fringe contains only the node for A.

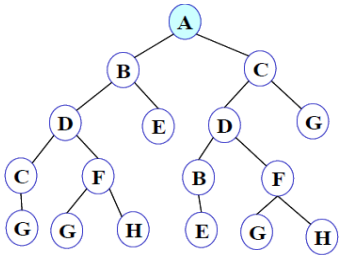


Figure 1

FRINGE: A

Step 2: A is removed from fringe. A is expanded and its children B and C are put in front of fringe.

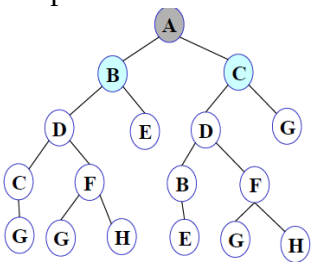


Figure 2

FRINGE: B C

Step 3: Node B is removed from fringe, and its children D and E are pushed in front of fringe.

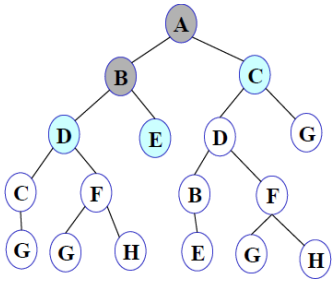


Figure 3

FRINGE: D E C

Step 4: Node D is removed from fringe. C and F are pushed in front of fringe.

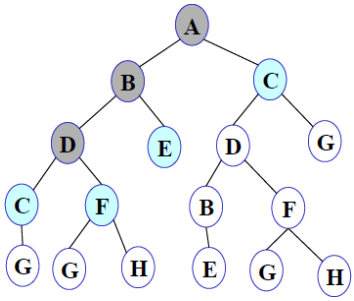


Figure 4

FRINGE: C F E C

Step 5: Node C is removed from fringe. Its child G is pushed in front of fringe.

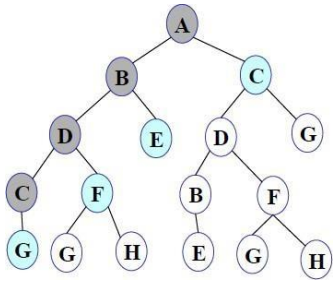


Figure 5

FRINGE: G F E C

Step 6: Node G is expanded and found to be a goal node.

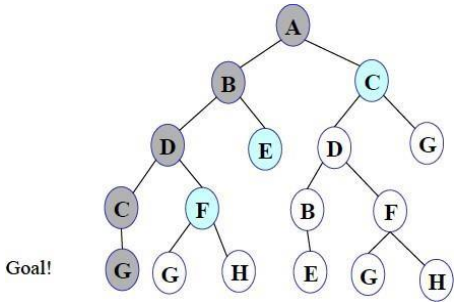


Figure 6

FRINGE: *G* F E C

The solution path A-B-D-C-G is returned and the algorithm terminates.

Depth first search is:

1. The algorithm takes exponential time.
2. If N is the maximum depth of a node in the search space, in the worst case the algorithm will take time $O(b^d)$.
3. The space taken is linear in the depth of the search tree, O(bN).

Note that the time taken by the algorithm is related to the maximum depth of the search tree. If the search tree has infinite depth, the algorithm may not terminate. This can happen if the search space is infinite. It can also happen if the search space contains cycles. The latter case can be handled by checking for cycles in the algorithm. Thus **Depth First Search is not complete.**

Exhaustive searches- Iterative Deeping DFS

Description:

- It is a search strategy resulting when you combine BFS and DFS, thus combining the advantages of each strategy, taking the completeness and optimality of BFS and the modest memory requirements of DFS.
- IDS works by looking for the best search depth d, thus starting with depth limit 0 and make a BFS and if the search failed it increase the depth limit by 1 and try a BFS again with depth 1 and so on – first d = 0, then 1 then 2 and so on – until a depth d is reached where a goal is found.

Algorithm:

```
procedure IDDFS(root)
   for depth from 0 to ∞
      found ← DLS(root, depth)
      if found ≠ null
         return found


procedure DLS(node, depth)
   if depth = 0 and node is a goal
      return node
   else if depth > 0
      foreach child of node
         found ← DLS(child, depth−1)
         if found ≠ null
            return found
   return null
```

Performance Measure:

- Completeness: IDS is like BFS, is complete when the branching factor b is finite.
- Optimality: IDS is also like BFS optimal when the steps are of the same cost.

- Time Complexity:

  - One may find that it is wasteful to generate nodes multiple times, but actually it is not that costly compared to BFS, that is because most of the generated nodes are always in the deepest level reached, consider that we are searching a binary tree and our depth limit reached 4, the nodes generated in last level = $2^4 = 16$, the nodes generated in all nodes before last level = $2^0 + 2^1 + 2^2 + 2^3 = 15$
  - Imagine this scenario, we are performing IDS and the depth limit reached depth d, now if you remember the way IDS expands nodes, you can see that nodes at depth d are generated once, nodes at depth d-1 are generated 2 times, nodes at depth d-2 are generated 3 times and so on, until you reach depth 1 which is generated d times, we can view the total number of generated nodes in the worst case as:
    - $N(IDS) = (b)d + (d-1)b^2 + (d-2)b^3 + \ldots. + (2)b^{d-1} + (1)b^d = O(b^d)$
  - If this search were to be done with BFS, the total number of generated nodes in the worst case will be like:
    - $N(BFS) = b + b^2 + b^3 + b^4 + \ldots. b^d + (b^{d+1} - b) = O(b^{d+1})$
  - If we consider a realistic numbers, and use b = 10 and d = 5, then number of generated nodes in BFS and IDS will be like
    - N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450
    - N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100
    - BFS generates like 9 time nodes to those generated with IDS.

- Space Complexity:

  - IDS is like DFS in its space complexity, taking $O(bd)$ of memory.

  Weblinks:

  i. *https://www.youtube.com/watch?v=7QcoJjSVT38*

  ii. https://mhesham.wordpress.com/tag/iterative-deepening-depth-first-search

Conclusion:

- We can conclude that IDS is a hybrid search strategy between BFS and DFS inheriting their advantages.
- IDS is faster than BFS and DFS.
- It is said that "IDS is the preferred uniformed search method when there is a large search space and the depth of the solution is not known".

Heuristic Searches:

A Heuristic technique helps in solving problems, even though there is no guarantee that it will never lead in the wrong direction. There are heuristics of every general applicability as well as domain specific. The strategies are general purpose heuristics. In order to use them in a specific domain they are coupler with some domain specific heuristics. There are two major ways in which domain - specific, heuristic information can be incorporated into rule-based search procedure.

A heuristic function is a function that maps from problem state description to measures desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.

Greedy Best First Search

Greedy best-first search tries to expand the node that is closest to the goal, on the: grounds that this is likely to lead to a solution quickly. Thus, it evaluates nodes by using just the heuristic function:

$$f(n) = h(n).$$

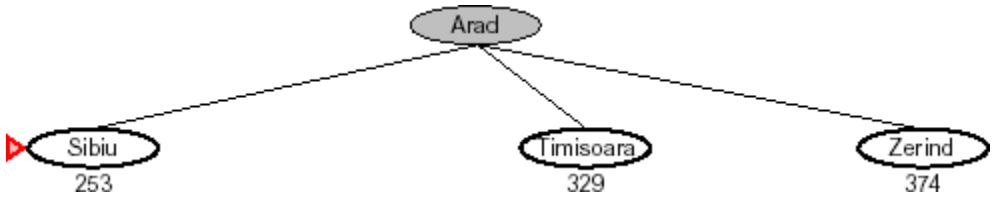
Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad. We need to know the straight-line distances to Bucharest from various cities as shown in **Figure 8.1**. For example, the initial state is In (Arad), and the straight line distance heuristic h_{SLD} (In (Arad)) is found to be 366. Using the **straight-line distance** heuristic **h_{SLD},** the goal state can be reached faster.

Arad	366	**Mehadia**	241	**Hirsova**	151
Bucharest	0	**Neamt**	234	**Urziceni**	80
Craiova	160	**Oradea**	380	**Iasi**	226
Drobeta	242	**Pitesti**	100	**Vaslui**	199
Eforie	161	**Rimnicu Vilcea**	193	**Lugoj**	244
Fagaras	176	**Sibiu**	253	**Zerind**	374
Giurgiu	77	**Timisoara**	329		

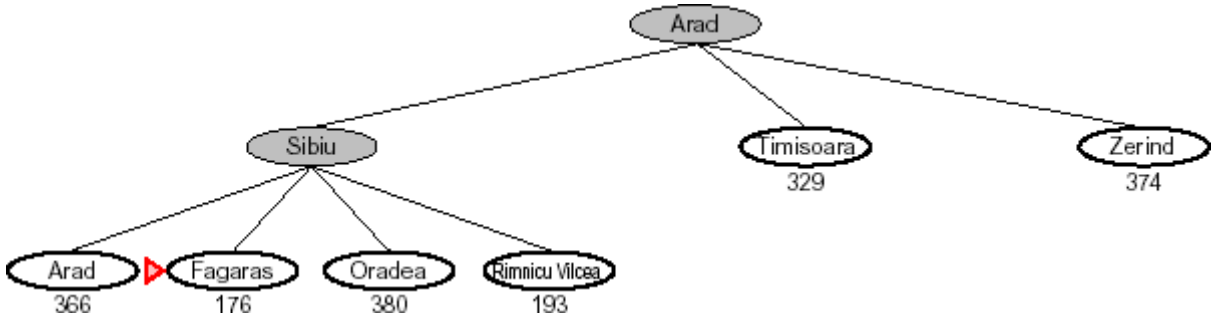
Figure 8.1: Values of h_{SLD}-straight-line distances to B u c h a r e s t.

The Initial State

After Expanding Arad



After Expanding Sibiu



After Expanding Fagaras

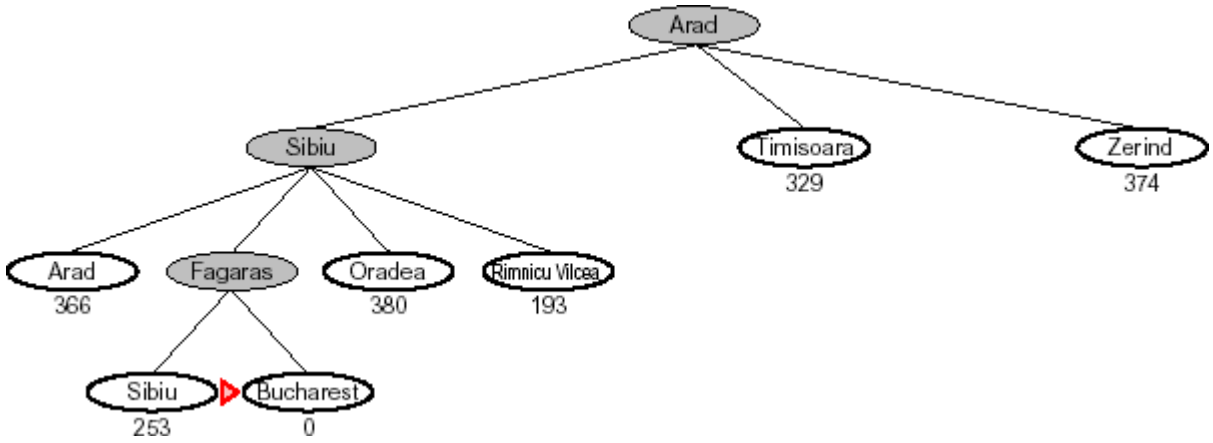


Figure 8.2: Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Figure 8.2 shows the progress of greedy best-first search using h_{SLD} to find a path from Arad to Bucharest. The first node to be expanded from Arad will be Sibiu, because it is closer to Bucharest than either Zerind or Timisoara. The next node to be expanded will be Fagaras, because it is closest.

Fagaras in turn generates Bucharest, which is the goal.

Evaluation Criterion of Greedy Search

- **Complete: NO [can get stuck in loops, e.g., Complete in finite space with repeated-state checking**]
- **Time Complexity: O (bm) [but a good heuristic can give dramatic improvement**]
- **Space Complexity: O (bm) [keeps all nodes in memory**]
- **Optimal: NO**

Greedy best-first search is not optimal, and it is incomplete. The worst-case time and space complexity is **O (b^m),** where m is the maximum depth of the search space.

HILL CLIMBING PROCEDURE:

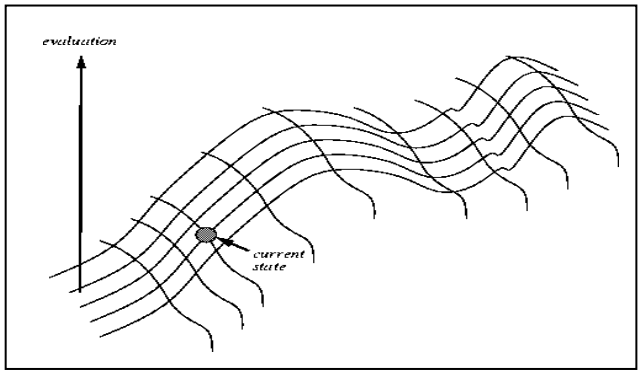
Hill Climbing Algorithm

We will assume we are trying to maximize a function. That is, we are trying to find a point in the search space that is better than all the others. And by "better" we mean that the evaluation is higher. We might also say that the solution is of better quality than all the others.

The idea behind hill climbing is as follows.

1. Pick a random point in the search space.
2. Consider all the neighbors of the current state.
3. Choose the neighbor with the best quality and move to that state.
4. Repeat 2 thru 4 until all the neighboring states are of lower quality.
5. Return the current state as the solution state.

We can also present this algorithm as follows (it is taken from the AIMA book (Russell, 1995) and follows the conventions we have been using on this course when looking at blind and heuristic searches).



a

Algorithm:

Function HILL-CLIMBING(*Problem*) **returns** a solution state

Inputs: *Problem*, problem

Local variables: *Current*, a node

Next, a node

Current = MAKE-NODE(INITIAL-STATE[*Problem*])

Loop do

Next = a highest-valued successor of *Current*

If VALUE[Next] < VALUE[Current] **then return** *Current*

Current = Next

End

You should note that this algorithm does not maintain a search tree. It only returns a final solution. Also, if two neighbors have the same evaluation and they are both the best quality, then the algorithm will choose between them at random.

Problems with Hill Climbing

The main problem with hill climbing (which is also sometimes called ***gradient descent***) is that we are not guaranteed to find the best solution. In fact, we are not offered any guarantees about the solution. It could be abysmally bad.

You can see that we will eventually reach a state that has no better neighbours but there are better solutions elsewhere in the search space. The problem we have just described is called a ***local maxima***.

Simulated annealing search

A hill-climbing algorithm that never makes "downhill" moves towards states with lower value (or higher cost) is guaranteed to be incomplete, because it can stuck on a local maximum. In contrast, a purely random walk –that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient. Simulated annealing is an algorithm that combines hill-climbing with a random walk in some way that yields both efficiency and completeness.

Figure 10.7 shows simulated annealing algorithm. It is quite similar to hill climbing. Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is

always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount ΔE by which the evaluation is worsened. The probability also decreases as the "temperature" T goes down: "bad moves are more likely to be allowed at the start when temperature is high, and they become more unlikely as T decreases. One can prove that if the schedule lowers T slowly enough, the algorithm will find a global optimum with probability approaching 1.

Simulated annealing was first used extensively to solve VLSI layout problems. It has been applied widely to factory scheduling and other large-scale optimization tasks.

function ***S I M U L A T E D - A N NEALING****(problem, schedule)* **returns** a **solution state**
inputs: *problem,* a problem
schedule, a mapping from time to "temperature"
local variables: *current,* a node
next, a node
T, a "temperature" controlling the probability of downward steps
current ← **MAKE-NODE(INITIAL-STATE[***problem***])**
for t← 1 **to** ∞ **do**
T ←*schedule[t]*
if *T* = 0 **then return** *current*
next ← a randomly selected successor of *current*
ΔE ← ***VALUE****[next]* – ***VALUE****[current]*
if ΔE > 0 **then** *current* ← *next*
else *current* ← *next* only with probability $e^{\Delta E / T}$

LOCAL SEARCH IN CONTINUOUS SPACES

- We have considered algorithms that work only in discrete environments, but real-world environment are continuous.
- Local search amounts to maximizing a continuous objective function in a multi-dimensional vector space.
- This is hard to do in general.
- Can immediately retreat
  - ✓ Discretize the space near each state

- ✓ Apply a discrete local search strategy (e.g., stochastic hill climbing, simulated annealing)

- Often resists a closed-form solution
  - ✓ Fake up an empirical gradient
  - ✓ Amounts to greedy hill climbing in discretized state space
- Can employ Newton-Raphson Method to find maxima.
- Continuous problems have similar problems: plateaus, ridges, local maxima, etc.

Best First Search:

- A combination of depth first and breadth first searches.
- Depth first is good because a solution can be found without computing all nodes and breadth first is good because it does not get trapped in dead ends.
- The best first search allows us to switch between paths thus gaining the benefit of both approaches. At each step the most promising node is chosen. If one of the nodes chosen generates nodes that are less promising it is possible to choose another at the same level and in effect the search changes from depth to breadth. If on analysis these are no better than this previously unexpanded node and branch is not forgotten and the search method reverts to the

OPEN is a priorityqueue of nodes that have been evaluated by the heuristic function but which have not yet been expanded into successors. The most promising nodes are at the front.

CLOSED are nodes that have already been generated and these nodes must be stored because a graph is being used in preference to a tree.

Algorithm:

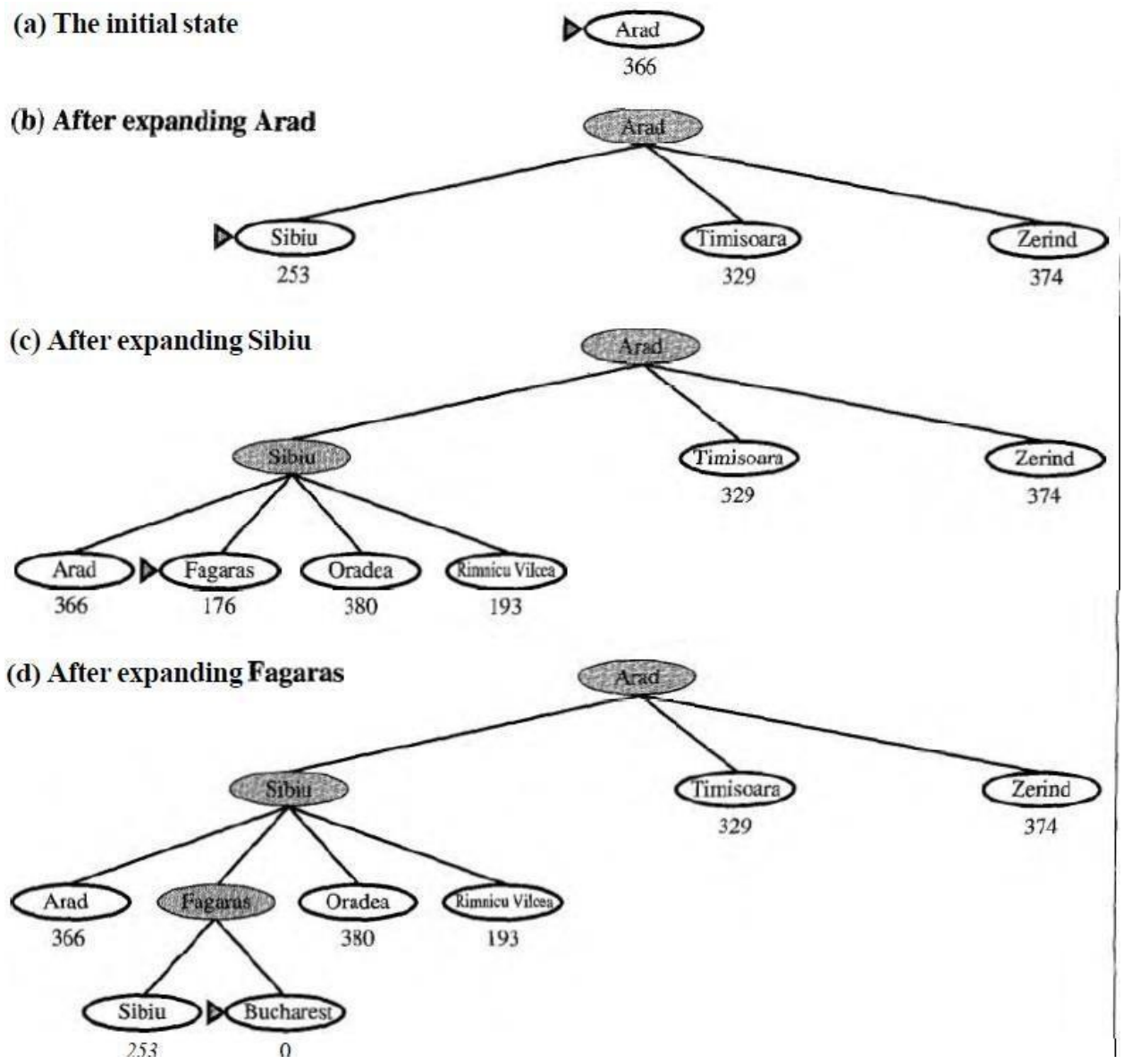
1. Start with OPEN holding the initial state
2. Until a goal is found or there are no nodes left on open do.

   - Pick the best node on OPEN
   - Generate its successors
   - For each successor Do

- If it has not been generated before ,evaluate it ,add it to OPEN and record its parent
- If it has been generated before change the parent if this new path is better and in that case update the cost of getting to any successor nodes.

3. If a goal is found or no more nodes left in OPEN, quit, else return to 2.

Example:



1. It is not optimal.
2. It is incomplete because it can start down an infinite path and never return to try other possibilities.

3. The worst-case time complexity for greedy search is $O(b^m)$, where m is the maximum depth of the search space.
4. Because greedy search retains all nodes in memory, its space complexity is the same as its time complexity

A* Algorithm

The Best First algorithm is a simplified form of the *A** algorithm.

The **A* search algorithm** (pronounced "Ay-star") is a tree search algorithm that finds a path from a given initial node to a given goal node (or one passing a given goal test). It employs a "heuristic estimate" which ranks each node by an estimate of the best route that goes through that node. It visits the nodes in order of this heuristic estimate.

Similar to greedy best-first search but is more accurate because A* takes into account the nodes that have already been traversed.

From *A** we note that ***f = g + h*** where

g is a measure of the distance/cost to go from the initial node to the current node

h is an estimate of the distance/cost to solution from the current node.

Thus ***f*** is an estimate of how long it takes to go from the initial node to the solution

Algorithm:

1. Initialize : **Set OPEN = (S); CLOSED = ( )**
g(s)= 0, f(s)=h(s)

2. Fail : **If OPEN = ( ), Terminate and fail.**

3. Select : **select the minimum cost state, n, from OPEN,**
save n in CLOSED

4. Terminate : **If n €G, Terminate with success and return f(n)**

5. Expand : **for each successor, m, of n**

a) If m € [OPEN U CLOSED]
Set g(m) = g(n) + c(n , m)
Set f(m) = g(m) + h(m)
Insert m in OPEN

b) If m € [OPEN U CLOSED]
Set g(m) = min { g(m) , g(n) + c(n , m)}

Set f(m) = g(m) + h(m)

If f(m) has decreased and m € CLOSED

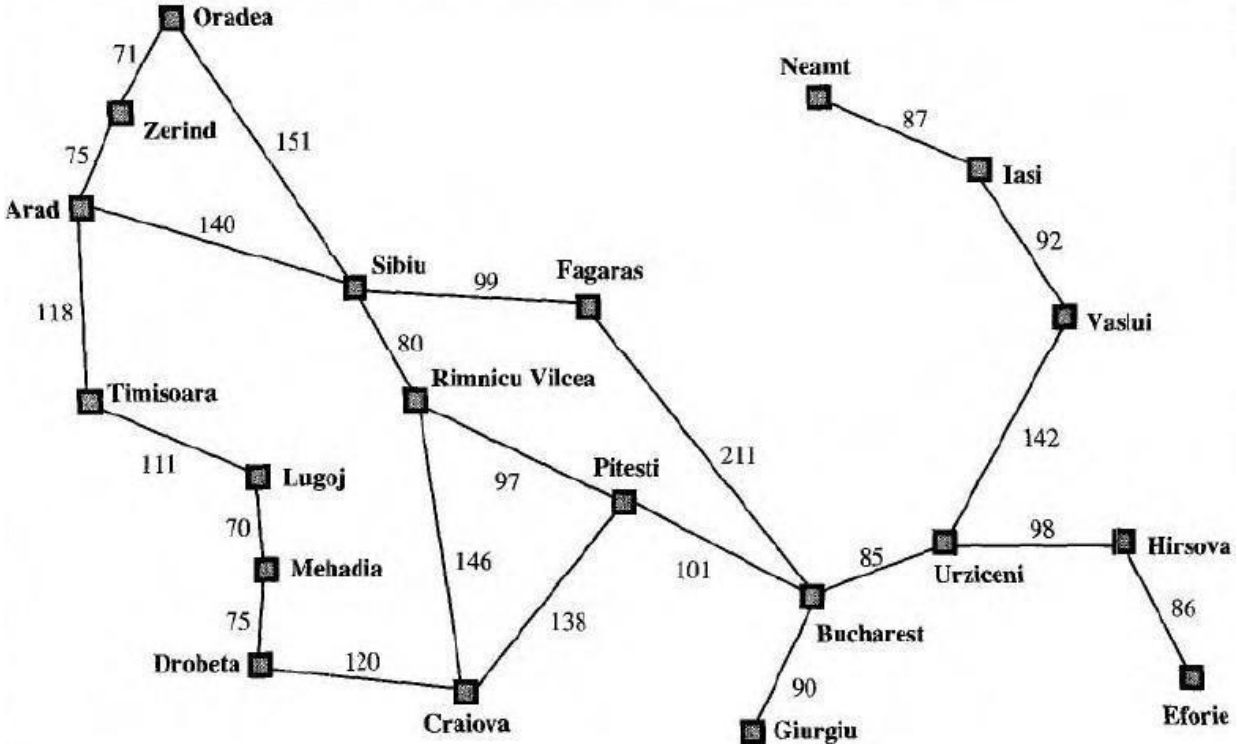
Move m to OPEN.

Description:

- A* begins at a selected node. Applied to this node is the "cost" of entering this node (usually zero for the initial node). A* then estimates the distance to the goal node from the current node. This estimate and the cost added together are the heuristic which is assigned to the path leading to this node. The node is then added to a priority queue, often called "open".
- The algorithm then removes the next node from the priority queue (because of the way a priority queue works, the node removed will have the lowest heuristic). If the queue is empty, there is no path from the initial node to the goal node and the algorithm stops. If the node is the goal node, A* constructs and outputs the successful path and stops.
- If the node is not the goal node, new nodes are created for all admissible adjoining nodes; the exact way of doing this depends on the problem at hand. For each successive node, A* calculates the "cost" of entering the node and saves it with the node. This cost is calculated from the cumulative sum of costs stored with its ancestors, plus the cost of the operation which reached this new node.
- The algorithm also maintains a 'closed' list of nodes whose adjoining nodes have been checked. If a newly generated node is already in this list with an equal or lower cost, no further processing is done on that node or with the path associated with it. If a node in the closed list matches the new one, but has been stored with a *higher* cost, it is removed from the closed list, and processing continues on the new node.
- Next, an estimate of the new node's distance to the goal is added to the cost to form the heuristic for that node. This is then added to the 'open' priority queue, unless an identical node is found there.
- Once the above three steps have been repeated for each new adjoining node, the original node taken from the priority queue is added to the 'closed' list. The next node is then popped from the priority queue and the process is repeated

The heuristic costs from each city to Bucharest:

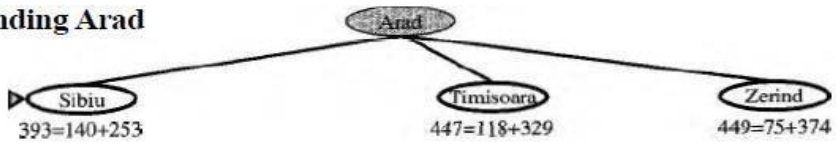
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374



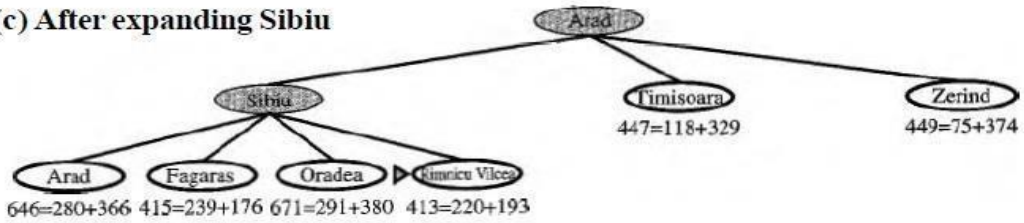
(a) The initial state

366=0+366

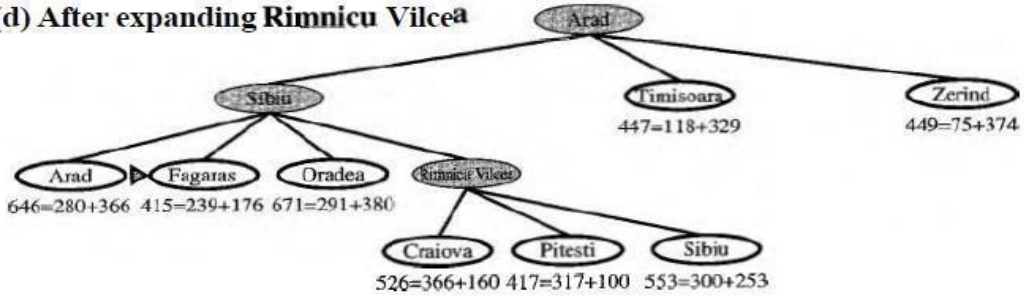
(b) After expanding Arad



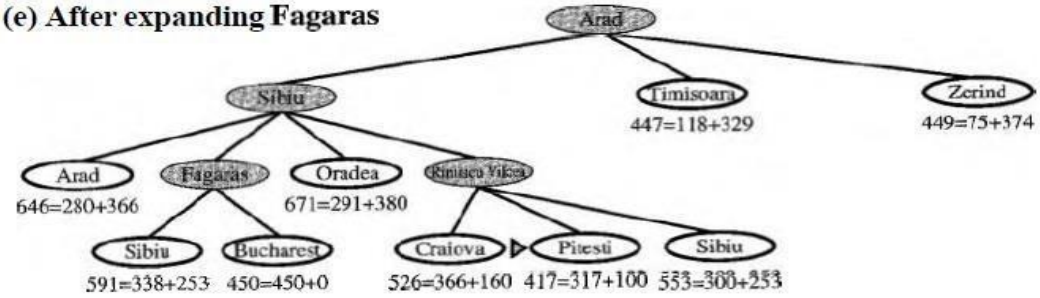
(c) After expanding Sibiu



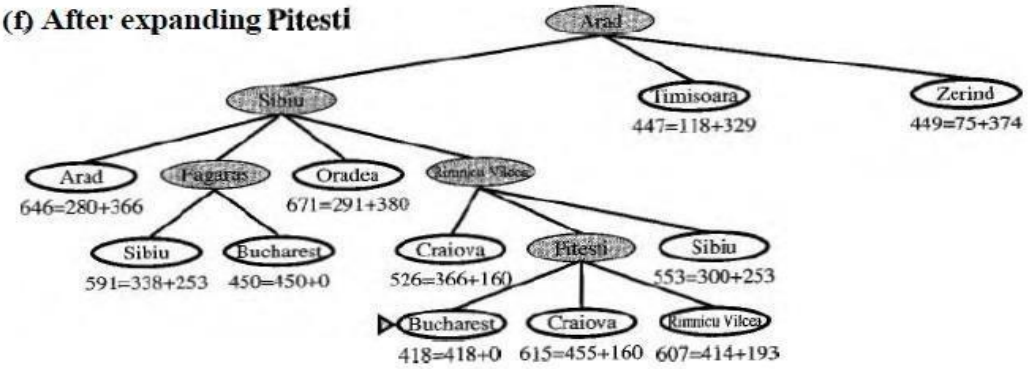
(d) After expanding Rimnicu Vilcea



(e) After expanding Fagaras



(f) After expanding Pitesti



A* search properties:

- The algorithm A* is admissible. This means that provided a solution exists, the first solution found by A* is an optimal solution. A* is admissible under the following conditions:
- Heuristic function: for every node n , $h(n) \leq h^*(n)$.
  - A* is also complete.
- A* is optimally efficient for a given heuristic.
- A* is much more efficient that uninformed search.

Iterative Deeping A* Algorithm:

Algorithm:

Let L be the list of visited but not expanded node, and

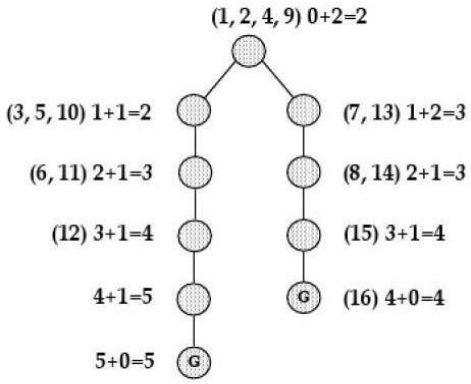
C the maximum depth

1) Let C=0
2) Initialize Lto the initial state (only)
3) If List empty increase C and goto 2),

   else

   extract a node n from the front of L
4) If n is a goal node,

   SUCCEED and return the path from the initial state to n
5) Remove n from L. If the level is smaller than C, insert at the front of L all the children n'



   of n with f(n') ≤ C
6) Goto 3)

- IDA* is complete & optimal Space usage is linear in the depth of solution. Each iteration is depth first search, and thus it does not require a priority queue.

- Iterative deepening A* (IDA*) eliminates the memory constraints of A* search algorithm without sacrificing solution optimality.
- Each iteration of the algorithm is a depth-first search that keeps track of the cost, f(n) = g(n) + h(n), of each node generated.
- As soon as a node is generated whose cost exceeds a threshold for that iteration, its path is cut off, and the search backtracks before continuing.
- The cost threshold is initialized to the heuristic estimate of the initial state, and in each successive iteration is increased to the total cost of the lowest-cost node that was pruned during the previous iteration.
- The algorithm terminates when a goal state is reached whose total cost dees not exceed the current threshold.

UNIT II

Constraint Satisfaction Problems

Refer:

https://www.cnblogs.com/RDaneelOlivaw/p/8072603.html

Sometimes a problem is not embedded in a long set of action sequences but requires picking the best option from available choices. A good general-purpose problem solving technique is to list the constraints of a situation (either negative constraints, like limitations, or positive elements that you want in the final solution). Then pick the choice that satisfies most of the constraints.

Formally speaking, a **constraint satisfaction problem (or CSP**) is defined by a set of variables, X1;X2; : : : ;Xn, and a set of constraints, C1;C2; : : : ;Cm. Each variable Xi has a nonempty domain Di of possible values. Each constraint Ci involves some subset of t variables and specifies the allowable combinations of values for that subset. A state of the problem is defined by an assignment of values to some or all of the variables, {Xi = vi;Xj = vj ; : : :} An assignment that does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is mentioned, and a solution to a CSP is a complete assignment that satisfies all the constraints. Some CSPs also require a solution that maximizes an objectivefunction.

CSP can be given an **incremental formulation** as a standard search problem as follows:

1. **Initial state**: the empty assignment fg, in which all variables are unassigned.
2. **Successor function**: a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
3. **Goal test**: the current assignment is complete.
4. **Path cost**: a constant cost for every step

Examples:

1. The best-known category of continuous-domain CSPs is that of **linear programming** problems, where constraints must be linear inequalities forming a *convex* region.
2. **Crypt arithmetic** puzzles.

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$

Example: The map coloring problem.

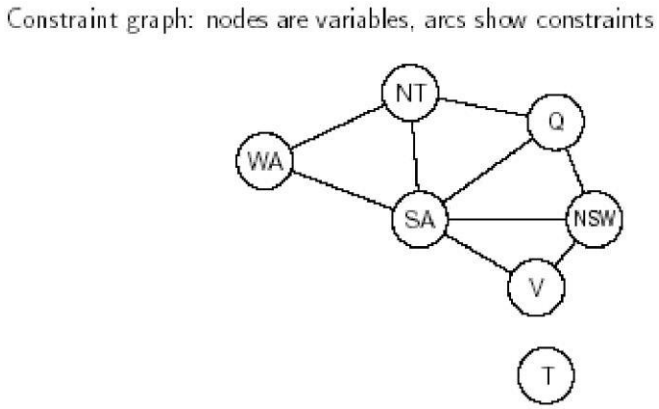
The task of coloring each region red, green or blue in such a way that no neighboring regions have the same color.

We are given the task of coloring each region red, green, or blue in such a way that the neighboring regions must not have the same color.

To formulate this as CSP, we define the variable to be the regions: WA, NT, Q, NSW, V, SA, and T. The domain of each variable is the set {red, green, blue}. The constraints require neighboring regions to have distinct colors: for example, the allowable combinations for WA and NT are the pairs {(red,green),(red,blue),(green,red),(green,blue),(blue,red),(blue,green)}. (The constraint can also be represented as the inequality WA ≠ NT). There are many possible solutions, such as {WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = red}.Map of Australia showing each of its states and territories

Variables WA, NT, Q, NSW, V, SA, T
Domains $D_i = \{red, green, blue\}$
Constraints: adjacent regions must have different colors
e.g., $WA \neq NT$ (if the language allows this), or
$(WA, NT) \in \{(red, green), (red, blue), (green, red), (green, blue), \ldots\}$

Constraint Graph: A CSP is usually represented as an undirected graph, called constraint graph where the nodes are the variables and the edges are the binary constraints.



The map-coloring problem represented as a constraint graph.

CSP can be viewed as a standard search problem as follows:

- **Initial state** : the empty assignment {},in which all variables are unassigned.
- **Successor function:** a value can be assigned to any unassigned variable, provided that it does not conflict with previously assigned variables.
- **Goal test:** the current assignment is complete.
- **Path cost:** a constant cost(E.g.,1) for every step.

Game Playing

Adversarial search, or game-tree search, is a technique for analyzing an adversarial game in order to try to determine who can win the game and what moves the players should make in order to win. Adversarial search is one of the oldest topics in Artificial Intelligence. The original ideas for adversarial search were developed by Shannon in 1950 and independently by Turing in 1951, in the context of the game of chess—and their ideas still form the basis for the techniques used today.

2-Person Games:

- Players: We call them Max and Min.
- Initial State: Includes board position and whose turn it is.
- Operators: These correspond to legal moves.

- Terminal Test: A test applied to a board position which determines whether the game is over. In chess, for example, this would be a checkmate or stalemate situation.
- Utility Function: A function which assigns a numeric value to a terminal state. For example, in chess the outcome is win (+1), lose (-1) or draw (0). Note that by convention, we always measure utility relative to Max**.**

MiniMax Algorithm:

1. Generate the whole game tree.
2. Apply the utility function to leaf nodes to get their values.
3. Use the utility of nodes at level n to derive the utility of nodes at level n-1.
4. Continue backing up values towards the root (one layer at a time).
5. Eventually the backed up values reach the top of the tree, at which point Max chooses the move that yields the highest value. This is called the minimax decision because it maximises the utility for Max on the assumption that Min will play perfectly to minimise it.

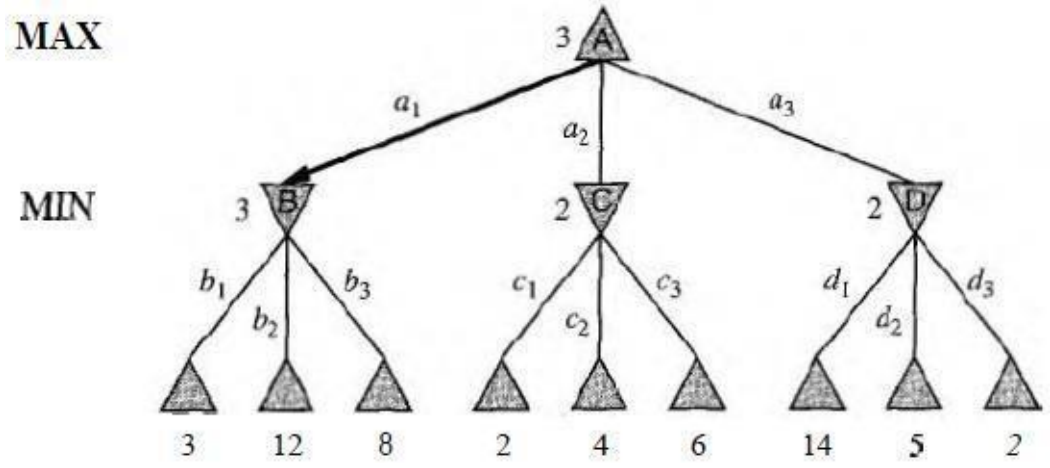
Algorithm: MINIMAX (Depth-First Version)

To determine the minimax value $V(J)$, do the following:

1. If J is terminal, return $V(J) = e(J)$; otherwise
2. Generate J's successors $J_1, J_2, \ldots, J_b$.
3. Evaluate $V(J_1), V(J_2), \ldots, V(J_b)$ from left to right.
4. If J is a MAX node, return $V(J) = \max[V(J_1), \ldots, V(J_b)]$.
5. If J is a MIN node, return $V(J) = \min[V(J_1), \ldots, V(J_b)]$.

```
function MINIMAX-DECISION(state) returns an action
    v ← MAX-VALUE(state)
    return the action in SUCCESSORS(state) with value v

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do
        v ← MIN(v, MAX-VALUE(s))
    return v
```

Example:



Properties of minimax:

- Complete : Yes (if tree is finite)
- Optimal : Yes (against an optimal opponent)
- Time complexity : $O(b^m)$
- Space complexity : O(bm) (depth-first exploration)
- For chess, b ≈ 35, m ≈100 for "reasonable" games

  → exact solution completely infeasible.

Limitations

- Not always feasible to traverse entire tree
- Time limitations

Alpha-beta pruning algorithm:

- **Pruning:** eliminating a branch of the search tree from consideration without exhaustive examination of each node
- α-β **Pruning:** the basic idea is to prune portions of the search tree that cannot improve the utility value of the max or min node, by just considering the values of nodes seen so far.
- *Alpha-beta pruning* is used on top of minimax search to detect paths that do not need to be explored. The intuition is:
- The MAX player is always trying to maximize the score. Call this α.
- The MIN player is always trying to minimize the score. Call this β .
- **Alpha cutoff**: Given a Max node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(n)
  (alpha increases and passes beta from below)
- **Beta cutoff.:** Given a Min node n, cutoff the search below n (i.e., don't generate or examine any more of n's children) if beta(n) <= alpha(n)
  (beta decreases and passes alpha from above)
- Carry alpha and beta values down during search Pruning occurs whenever alpha >= beta

Algorithm:

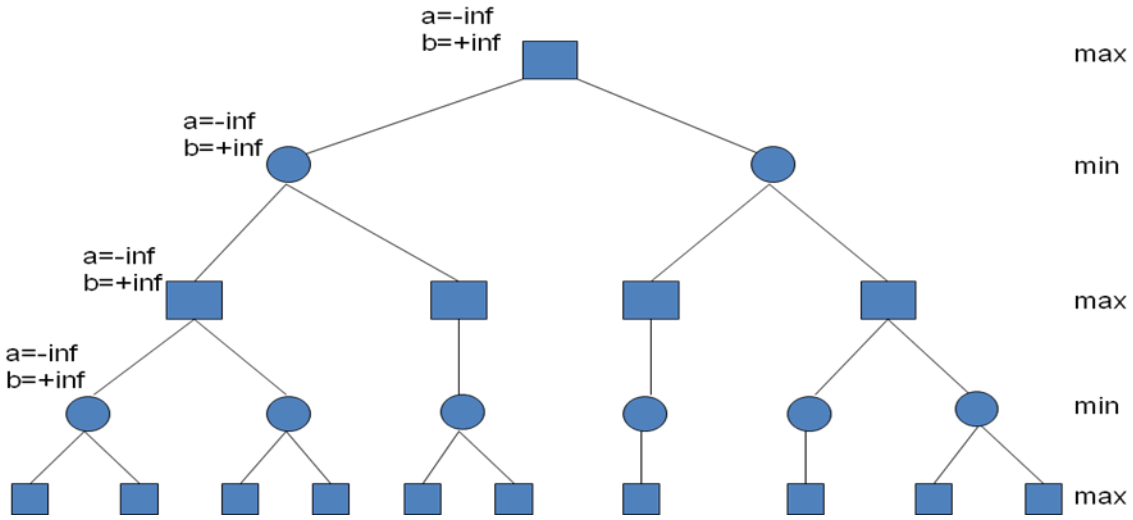
```
function ALPHA-BETA-SEARCH(state) returns an action
  inputs: state, current state in game

  v ← MAX-VALUE(state, −∞, +∞)
  return the action in SUCCESSORS(state) with value v
```

```
function MAX-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for a, s in SUCCESSORS(state) do
    v ← MAX(v, MIN-VALUE(s, α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

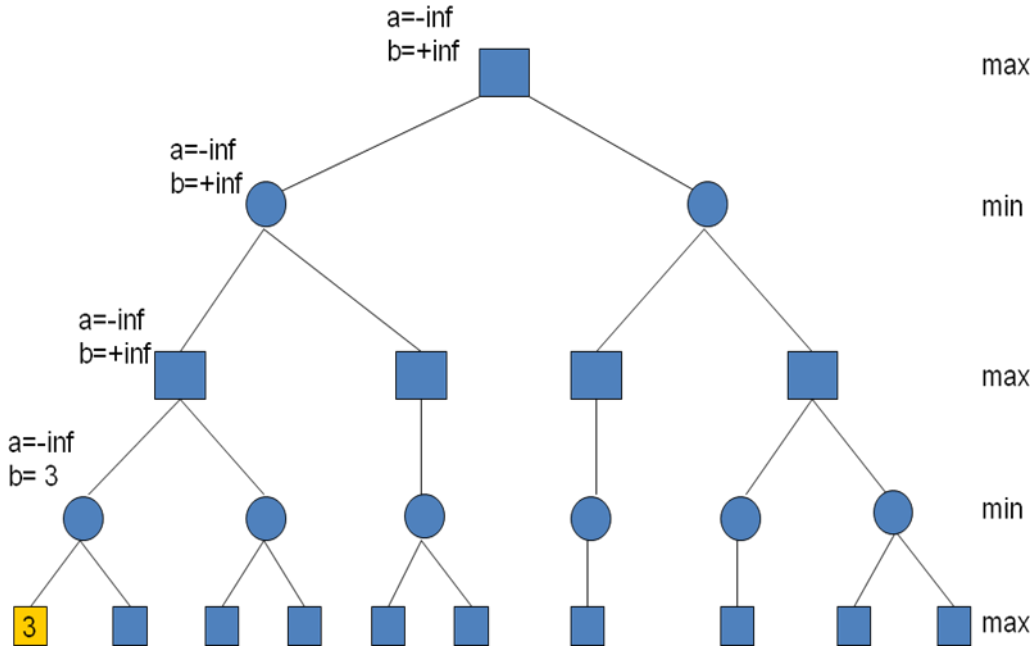
```
function MIN-VALUE(state, α, β) returns a utility value
  inputs: state, current state in game
          α, the value of the best alternative for MAX along the path to state
          β, the value of the best alternative for MIN along the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for a, s in SUCCESSORS(state) do
    v ← MIN(v, MAX-VALUE(%α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Example:

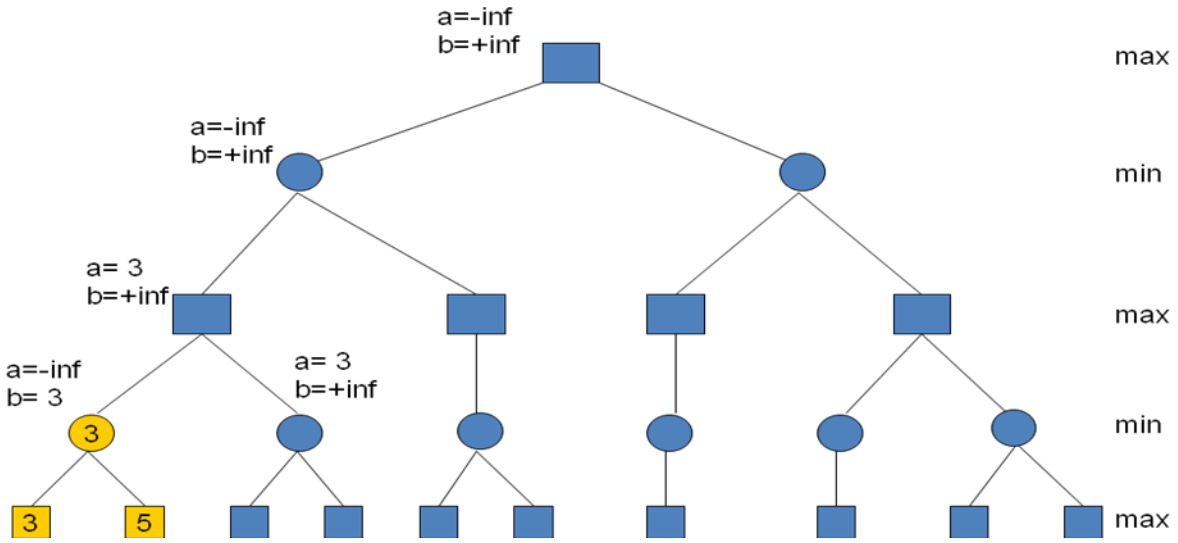
1) Setup phase: Assign to each left-most (or right-most) internal node of the tree, variables: alpha = -infinity, beta = +infinity



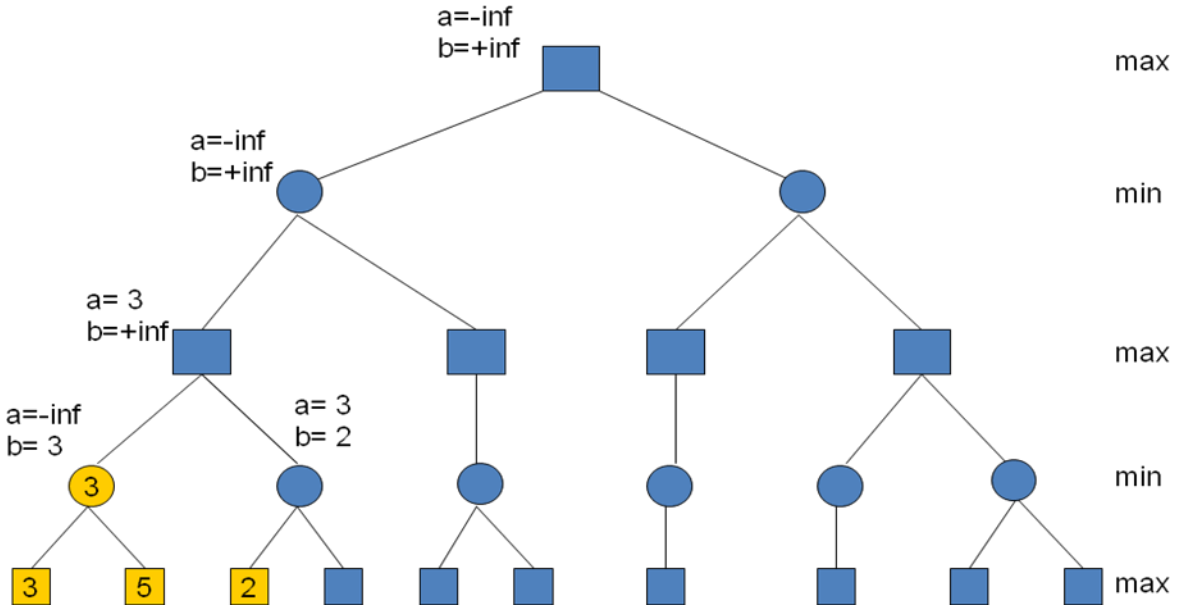
2) Look at first computed final configuration value. It's a 3. Parent is a min node, so set the beta (min) value to 3.



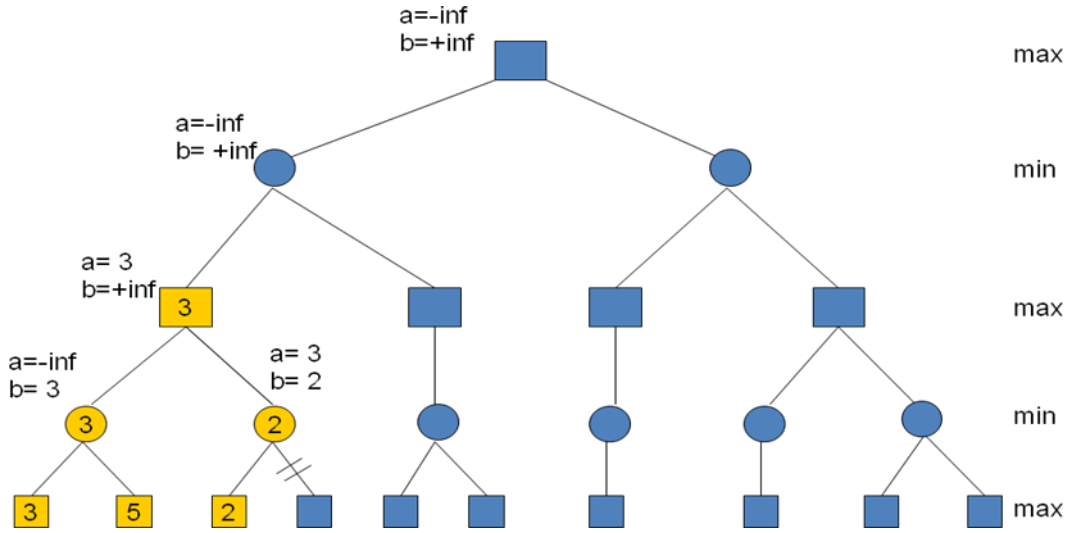
3) Look at next value, 5. Since parent is a min node, we want the minimum of 3 and 5 which is 3. Parent min node is done – fill alpha (max) value of its parent max node. Always set alpha for max nodes and beta for min nodes. Copy the state of the max parent node into the second unevaluated min child.



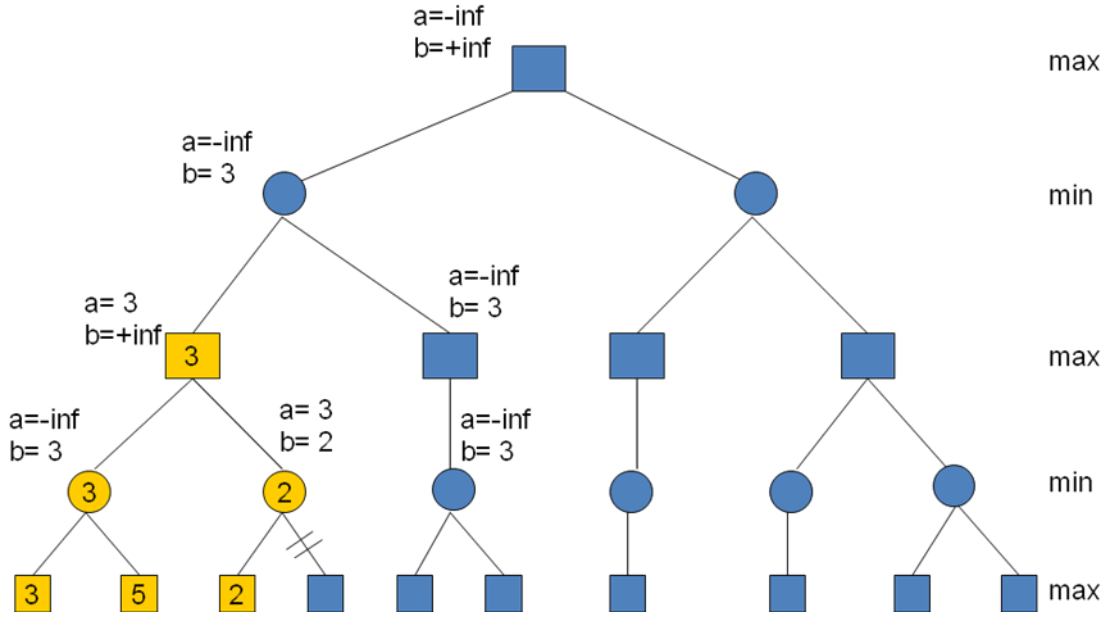
4) Look at next value, 2. Since parent node is min with b=+inf, 2 is smaller, change b.



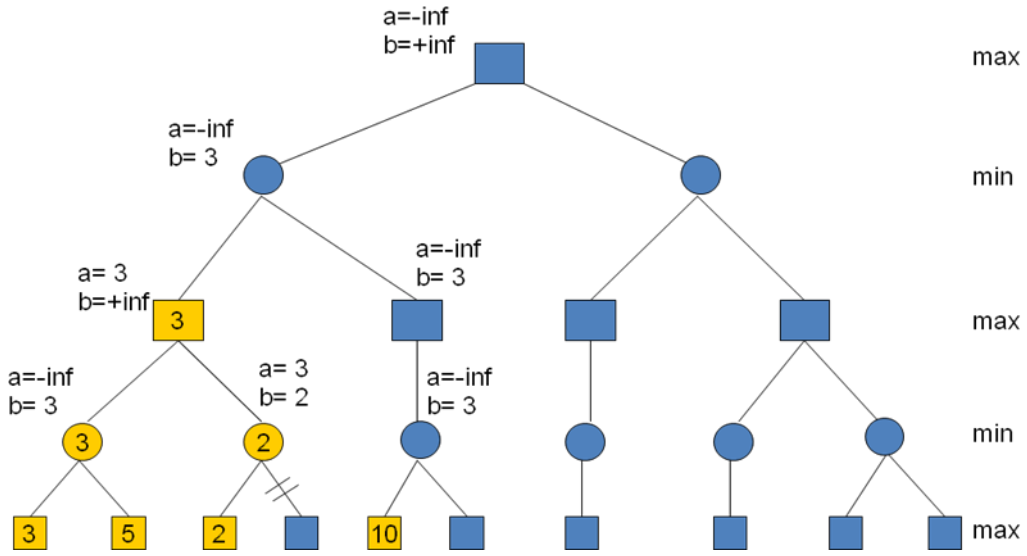
5) Now, the min parent node has a max value of 3 and min value of 2. The value of the 2nd child does not matter. If it is >2, 2 will be selected for min node. If it is <2, it will be selected for min node, but since it is <3 it will not get selected for the parent max node. Thus, we prune the right subtree of the min node. Propagate max value up the tree.



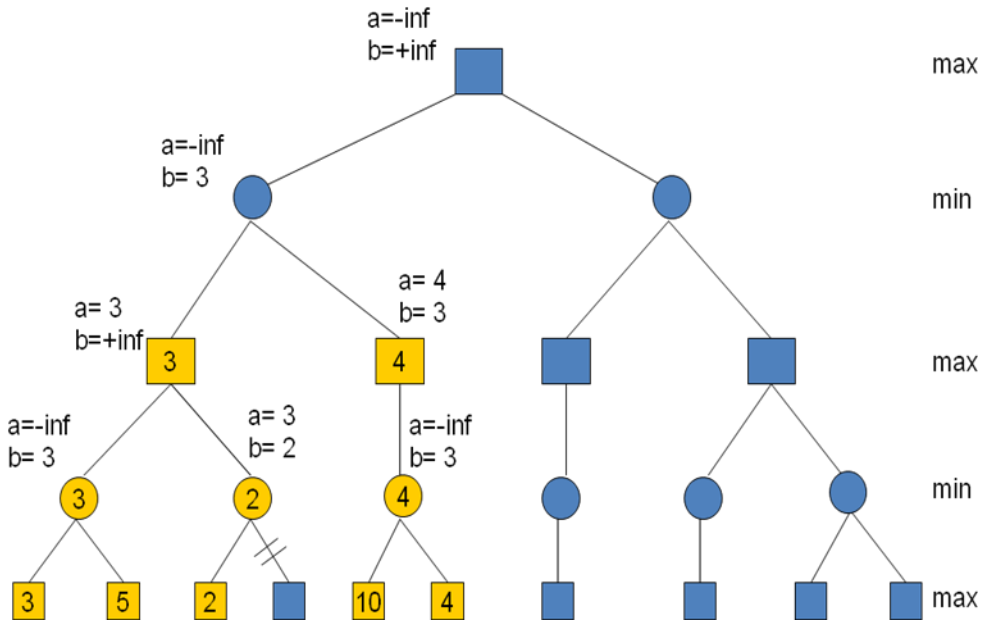
6) Max node is now done and we can set the beta value of its parent and propagate node state to sibling subtree's left-most path.



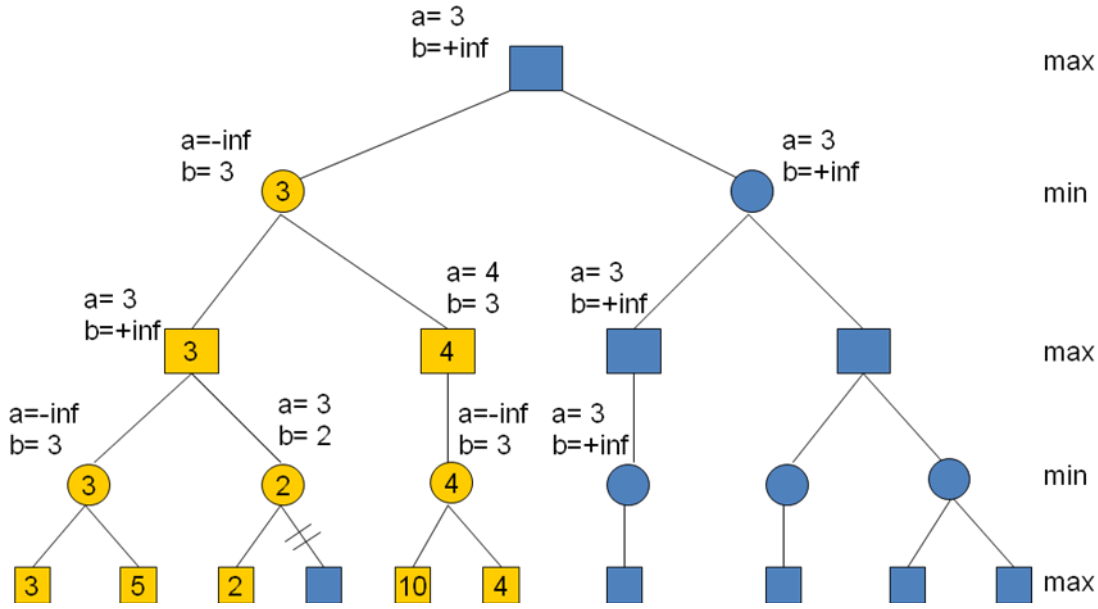
7) The next node is 10. 10 is not smaller than 3, so state of parent does not change. We still have to look at the 2nd child since alpha is still –inf.



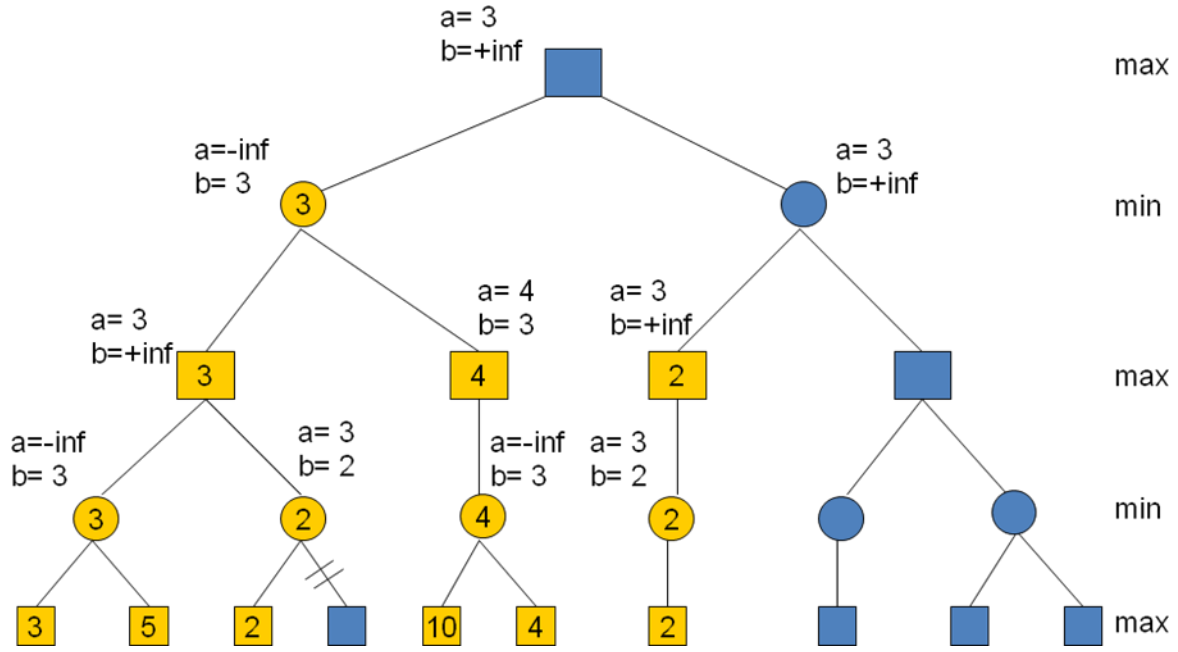
8) The next node is 4. Smallest value goes to the parent min node. Min subtree is done, so the parent max node gets the alpha (max) value from the child. Note that if the max node had a 2nd subtree, we can prune it since a>b.



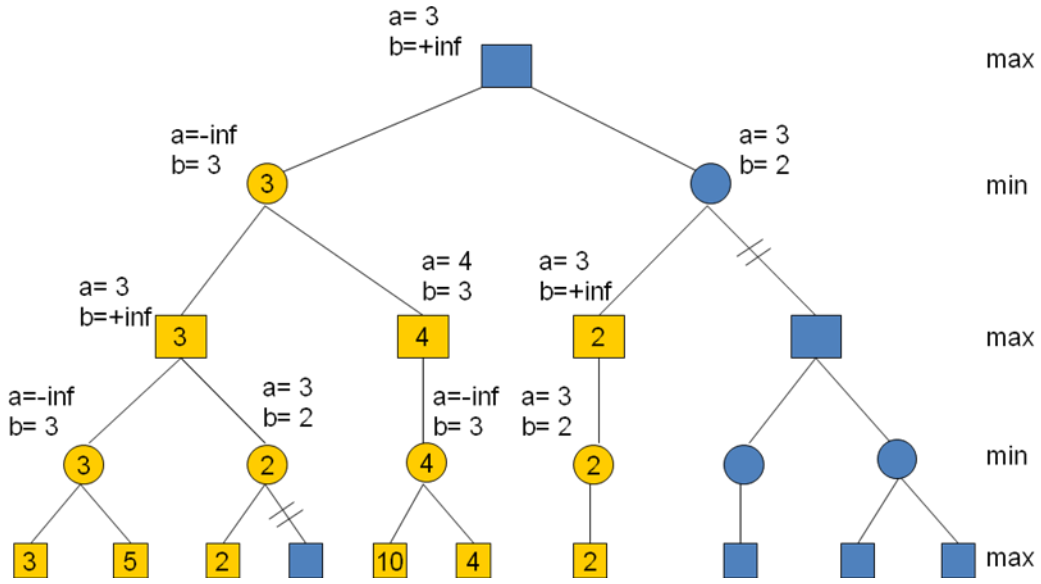
9) Continue propagating value up the tree, modifying the corresponding alpha/beta values. Also propagate the state of root node down the left-most path of the right subtree.



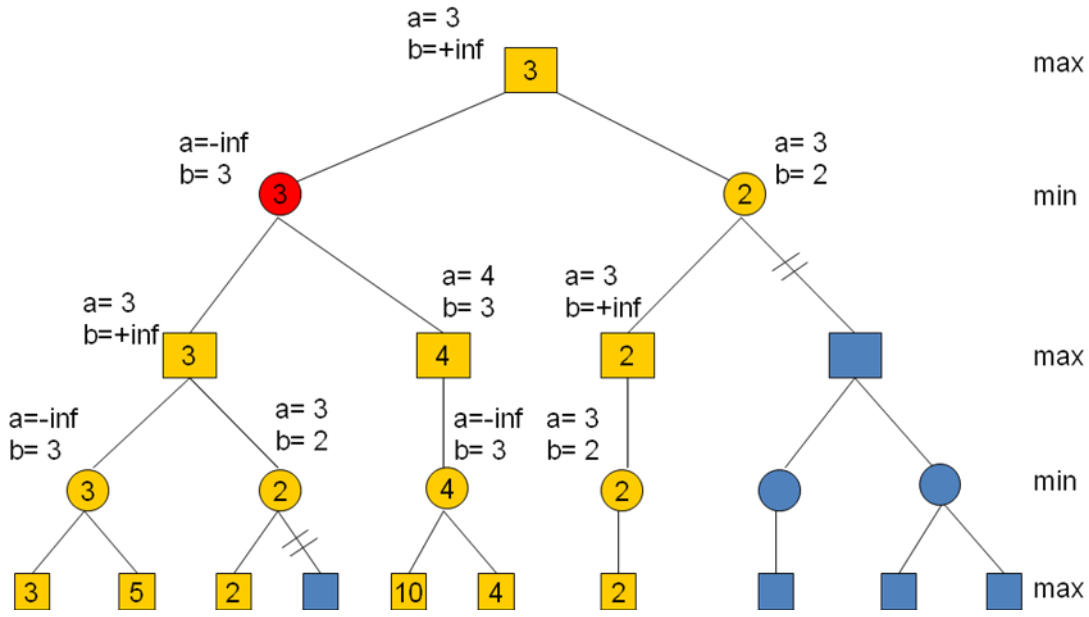
10) Next value is a 2. We set the beta (min) value of the min parent to 2. Since no other children exist, we propagate the value up the tree.



11) We have a value for the 3rd level max node, now we can modify the beta (min) value of the min parent to 2. Now, we have a situation that a>b and thus the value of the rightmost subtree of the min node does not matter, so we prune the whole subtree.



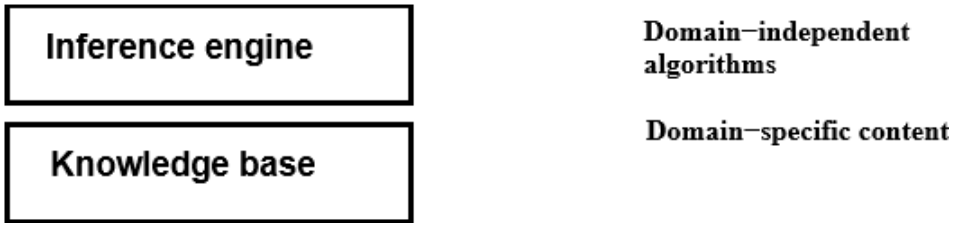
12) Finally, no more nodes remain, we propagate values up the tree. The root has a value of 3 that comes from the left-most child. Thus, the player should choose the left-most child's move in order to maximize his/her winnings. As you can see, the result is the same as with the mini-max example, but we did not visit all nodes of the tree.



UNIT III

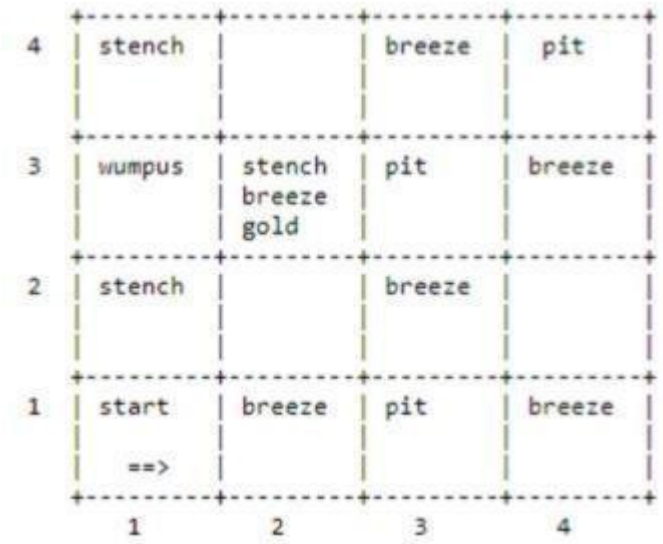
Knowledge Based Agents A knowledge-based agent needs a KB and an inference mechanism. It operates by storing sentences in its knowledge base, inferring new sentences with the inference mechanism, and using them to deduce which actions to take The interpretation of a sentence is the fact to which it refers.

Knowledge Bases:



Knowledge base = set of sentences in a formal language Declarative approach to building an agent (or other system): Tell it what it needs toknow - Thenitcan Askitselfwhattodo—answersshouldfollowfromtheKB Agents can be viewed at the knowledge leveli.e., what they know, regardless of howimplemented or at the implementation leveli.e.,data structuresinKBand algorithmsthatmanipulatethem. The Wumpus World:

A variety of "worlds" are being used as examples for Knowledge Representation, Reasoning, and Planning. Among them the Vacuum World, the Block World, and the Wumpus World. The Wumpus World was introduced by Genesereth, and is discussed in Russell-Norvig. The Wumpus World is a simple world (as is the Block World) for which to represent knowledge and to reason. It is a cave with a number of rooms, represented as a 4x4 square



Rules of the Wumpus World The neighborhood of a node consists of the four squares north, south, east, and west of the given square. In a square the agent gets a vector of percepts, with components Stench, Breeze, Glitter, Bump, Scream For example [Stench, None, Glitter, None, None] Stench is perceived at a square iff the wumpus is at this square or in its neighborhood. Breeze is perceived at a square iff a pit is in the neighborhood of this square. Glitter is perceived at a square iff gold is in this square Bump is perceived at a square iff the agent goes Forward into a wall Scream is perceived at a square iff the wumpus is killed anywhere in the cave An agent can do the following actions (one at a time): Turn (Right), Turn (Left), Forward, Shoot, Grab, Release, Climb The agent can go forward in the direction it is currently facing, or Turn Right, or Turn Left. Going forward into a wall will generate a Bump percept. The agent has a single arrow that it can shoot. It will go straight in the direction faced by the agent until it hits (and kills) the wumpus, or hits (and is absorbed by) a wall. The agent can grab a portable object at the current square or it can Release an object that it is holding. The agent can climb out of the cave if at the Start square.The Start square is (1,1) and initially the agent is facing east. The agent dies if it is in the same square asthe wumpus. The objective of the game is to kill the wumpus, to pick up the gold, and to climb out with it.

Representing our Knowledge about the Wumpus World Percept(x, y) Where x must be a percept vector and y must be a situation. It means that at situation y theagentperceives x.For convenience we introduce the following definitions: Percept([Stench,y,z,w,v],t) = > Stench(t) Percept([x,Breeze,z,w,v],t) = > Breeze(t) Percept([x,y,Glitter,w,v],t) = > AtGold(t) Holding(x, y)

Where x is an object and y is a situation. It means that the agent is holding the object x in situation y. Action(x, y) Where x must be an action (i.e. Turn (Right), Turn (Left), Forward,) and y must be a situation. It means that at situation y the agent takes action x. At(x,y,z) Where x is an object, y is a Location, i.e. a pair [u,v] with u and v in {1, 2, 3, 4}, and z is a situation. It means that the agent x in situation z is at location y. Present(x,s) Means that object x is in the current room in the situation s. Result(x, y) It means that the result of applying action x to the situation y is the situation Result(x,y).Notethat Result(x,y) is a term, not a statement. For example we can say Result(Forward, S0) = S1 Result(Turn(Right),S1) = S2 These definitions could be made more general. Since in the Wumpus World there is a single agent, there is no reason for us to make predicates and functions relative to a specific agent. In other"worlds" we should change things appropriately.

Validity And Satisfiability

A sentence is valid

if it is true in all models, e.g.,True,A∨¬A, A⇒A,(AA(A⇒B)) ⇒B Validity is connected to inference via the Deduction Theorem: KB |= αif and onlyif(KB⇒α) isvalid Asentenceissatisfiableifitistrue insome model e.g., A∨B, C Asentence isunsatisfiableifitistrueinnomodels e.g., A A¬A Satisfiability is connected to inference via the following: KB|=α iff(KBA¬α)isunsatisfiable i.e., prove α by reductionandabsurdum

Proof Methods

Proof methods divide into (roughly)two kinds:

Application of inference rules – Legitimate(sound)generationofnewsentencesfromold – Proof=asequenceofinferenceruleapplicationscanuseinferencerulesasoperatorsinastand ardsearch algorithm – Typicallyrequiretranslationofsentencesintoanormalform Model checking – Truthtableenumeration(alwaysexponentialinn) – Improvedbacktracking,e.g.,Davis–Putnam–Loge Mann–Loveland – Heuristic searchinmodelspace(soundbutincomplete) e.g.,min-conflicts-likehillclimbingalgorithms

Forward and Backward Chaining

Horn Form (restricted) KB = conjunction of Horn clauses Horn clause = – proposition symbol;or – (conjunctionofsymbols) ⇒ symbol Example KB: CA(B ⇒ A) A (CAD ⇒ B) Modus Ponens (for Horn Form): complete for Horn KBs

α1,...,αn,α1A⋯Aα⇒ β β

Canbeusedwithforwardchaining orbackwardchaining. These algorithms areverynaturalandruninlineartime.,

ForwardChaining

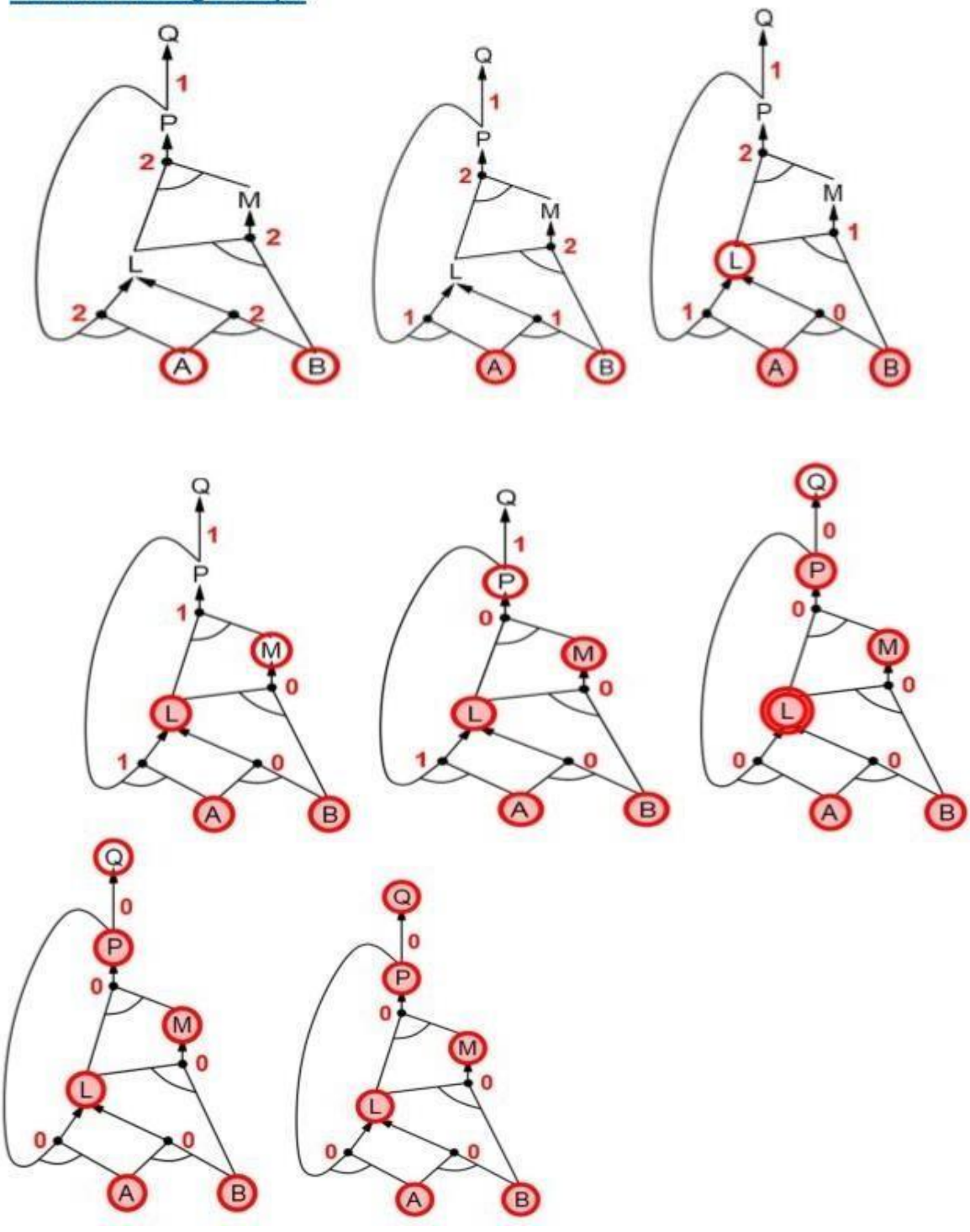
Idea: If anyrulewhosepremisesaresatisfiedintheKB, additsconclusiontotheKB,untilqueryisfound

ForwardChaining Algorithm

ForwardChaining Algorithm

```
function PL-FC-Entails?(KB,q) returns true or false
  inputs: KB, the knowledge base, a set of propositional Horn clauses
          q, the query, a proposition symbol
  local variables: count, a table, indexed by clause, initially the number of premises
                   inferred, a table, indexed by symbol, each entry initially false
                   agenda, a list of symbols, initially the symbols known in KB
  while agenda is not empty
      do p ← Pop(agenda)
      unless inferred[p] do
          inferred[p] ← true
          for each Horn clause c in whose premise p appears do
              decrement count[c]
```

ForwardChaining Example



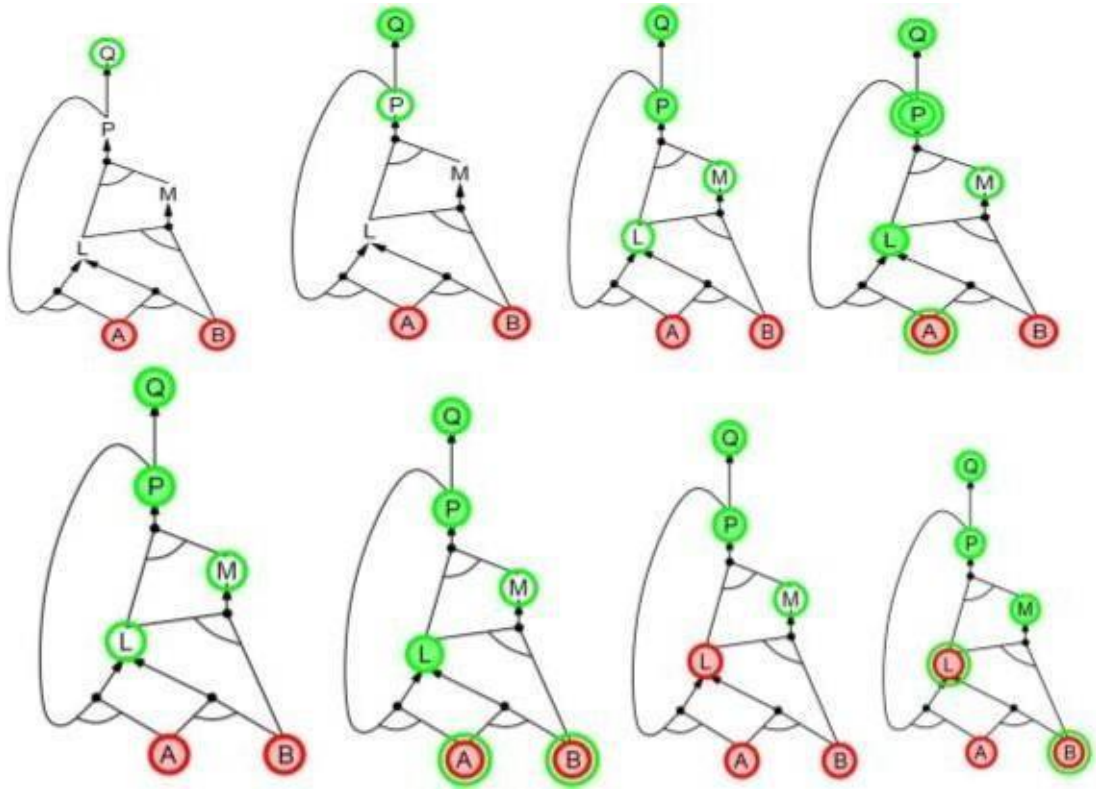
Proof of Completeness

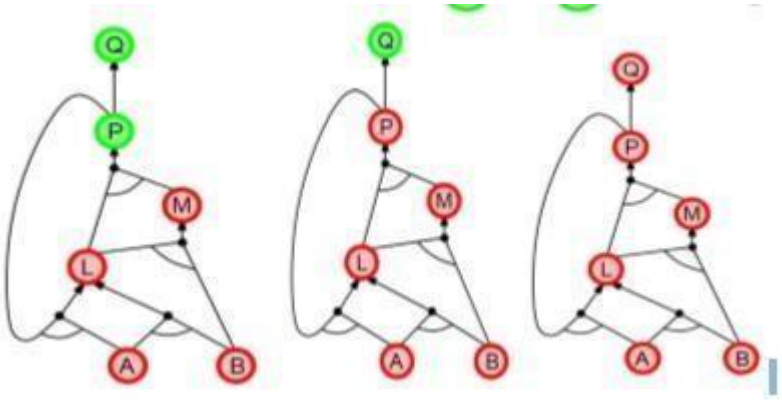
FC derives every atomic sentence that is entailed by KB 1. FCreachesafixedpointwherenonewatomicsentencesarederived 2. Considerthefinalstateasamodelm,assigningtrue/falsetosymbols 3. Every clause in the original

KB is true inm i. Proof:Supposeaclausea1A...Aak⇒bisfalsei nm Then a1A. . . Aakis true in m and b is false in m Thereforethealgorithmhasnotreachedafixedpoint ! 4. Hence m is a model ofKB 5. IfKB|=q,thenqistrueineverymodelofKB,includingm a. Generalidea: constructanymodelofKBby soundinference,checkα

Backward Chaining

Idea:workbackwardsfromthequeryq: to prove q byBC, check if q is known already, or prove by BC all premises of some rule concluding q Avoidloops: checkifnewsubgoalisalreadyonthegoalstack Avoid repeated work: check if new subgoal 1. has already been proved true, or 2. has alreadyfailed





Forward vs Backward Chaining

FC is data-driven, cf. automatic, unconscious processing, e.g.,objectrecognition,routinedecisions Maydolotsofworkthatisirrelevanttothegoal BC is goal-driven, appropriate forproblem-solving, e.g., Where are my keys? How do I get into a PhD program? Complexity of BC can be much less than linear in size of KB

FIRST ORDER LOGIC:

PROCEDURAL LANGUAGES AND PROPOSITIONAL LOGIC:

Drawbacks of Procedural Languages

 Programming languages (such as C++ or Java or Lisp) are by far the largest class of formal languages in common use. Programs themselves represent only computational processes. Data structures within programs can represent facts.

For example, a program could use a 4 × 4 array to represent the contents of the wumpus world. Thus, the programming language statement World[2,2]← Pit is a fairly natural way to assert that there is a pit in square [2,2].

What programming languages lack is any general mechanism for deriving facts from other facts; each update to a data structure is done by a domain-specific procedure whose details are derived by the programmer from his or her own knowledge of the domain.

 A second drawback of is the lack the expressiveness required to handle partial information . For example data structures in programs lack the easy way to say, “There is a pit in [2,2] or [3,1]” or “If the wumpus is in [1,1] then he is not in [2,2].”

Advantages of Propositional Logic

 The declarative nature of propositional logic, specify that knowledge and inference are separate, and inference is entirely domain-independent. Propositional logic is a declarative language because its semantics is based on a truth relation between sentences and possible worlds. It also has sufficient expressive power to deal with partial information, using disjunction and negation.

 Propositional logic has a third COMPOSITIONALITY property that is desirable in representation languages, namely, compositionality. In a compositional language, the meaning of a sentence is a function of the meaning of its parts. For example, the meaning of "S1,4A S1,2" is related to the meanings of "S1,4" and "S1,2.

Drawbacks of Propositional Logic Propositional logic lacks the expressive power to concisely describe an environment with many objects.

For example, we were forced to write a separate rule about breezes and pits for each square, such as $B1,1 \Leftrightarrow (P1,2 \vee P2,1)$.

 In English, it seems easy enough to say, "Squares adjacent to pits are breezy." The syntax and semantics of English somehow make it possible to describe the environment concisely

SYNTAX AND SEMANTICS OF FIRST-ORDER LOGIC

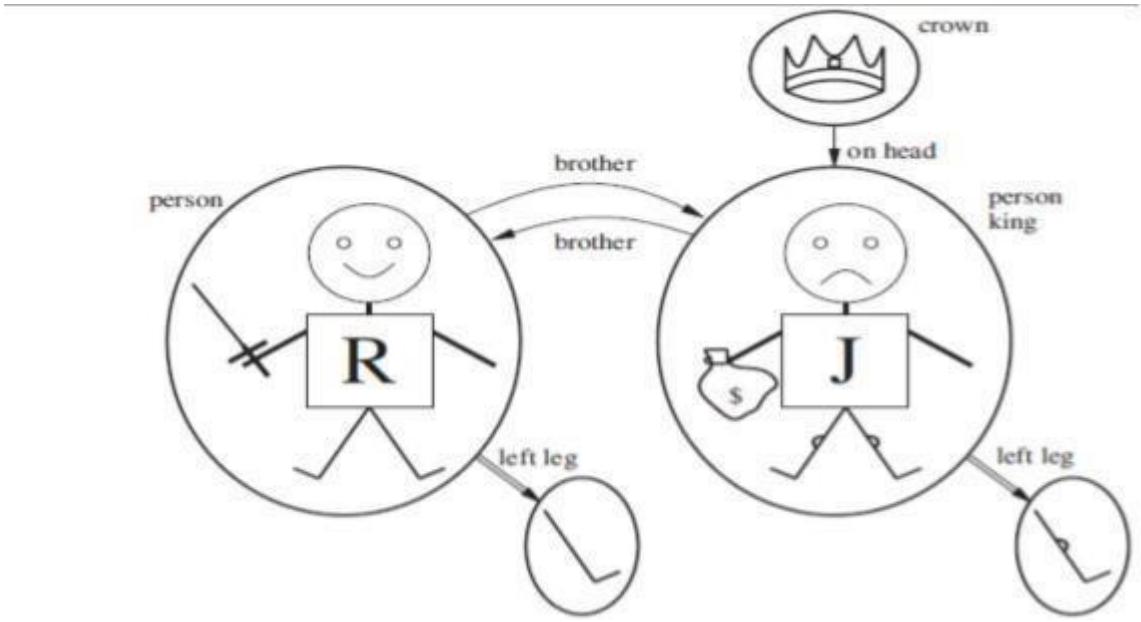
Models for first-order logic :

The models of a logical language are the formal structures that constitute the possible worlds under consideration. Each model links the vocabulary of the logical sentences to elements of the possible world, so that the truth of any sentence can be determined. Thus, models for propositional logic link proposition symbols to predefined truth values. Models for first-order logic have objects. The domain of a model is the set of objects or domain elements it contains. The domain is required to be nonempty—every possible world must contain at least one object.

A relation is just the set of tuples of objects that are related. Unary Relation: Relations relates to single Object Binary Relation: Relation Relates to multiple objects Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object.

For Example:

Richard the Lionheart, King of England from 1189 to 1199; His younger brother, the evil King John, who ruled from 1199 to 1215; the left legs of Richard and John; crown



Unary Relation : John is a king Binary Relation :crown is on head of john , Richard is brother ofjohn The unary "left leg" function includes the following mappings: (Richard the Lionheart) ->Richard's left leg (King John) ->Johns left Leg

Symbols and interpretations

Symbols are the basic syntactic elements of first-order logic. Symbols stand for objects, relations, and functions.

The symbols are of three kinds: □ Constant symbols which stand for objects; Example: John, Richard □Predicate symbols, which stand for relations; Example: OnHead, Person, King, and Crown Function symbols, which stand for functions. Example: left leg

Symbols will begin with uppercase letters.

Interpretation The semantics must relate sentences to models in order to determine truth. For this to happen, we need an interpretation that specifies exactly which objects, relations and functions are referred to by the constant, predicate, and function symbols.

For Example:

 Richard refers to Richard the Lionheart and John refers to the evil king John. Brother refers to the brotherhood relation OnHead refers to the "on head relation that holds between the crown and King John; Person, King, and Crown refer to the sets of objects that are persons, kings, and crowns. LeftLeg refers to the "left leg" function,

The truth of any sentence is determined by a model and an interpretation for the sentence's symbols. Therefore, entailment, validity, and so on are defined in terms of all possiblemodels and all possible interpretations. The number of domain elements in each model may be unbounded-for example, the domain elements may be integers or real numbers. Hence, the number of possible models is anbounded, as is the number of interpretations.

Term

A term is a logical expression that refers to an object. Constant symbols are therefore terms. Complex Terms A complex term is just a complicated kind of name. A complex term is formed by a function symbol followed by a parenthesized list of terms as arguments to the function symbol For example: "King John's left leg" Instead of using a constant symbol, we use LeftLeg(John). The formal semantics of terms :

Consider a term f (tl,. . . , t,). The function symbol frefers to some function in the model (F); the argument terms refer to objects in the domain (call them d1….dn); and the term as a whole refers to the object that is the value of the function Fapplied to dl, . . . , d,. For example,: the LeftLeg function symbol refers to the function " (King John) -+ John's left leg" and John refers to King John, then LeftLeg(John) refers to King John's left leg. In this way, the interpretation fixes the referent of every term.

Atomic sentences

An atomic sentence is formed from a predicate symbol followed by a parenthesized list of terms: For Example: Brother(Richard, John).

Atomic sentences can have complex terms as arguments. For Example: Married (Father(Richard), Mother(John)).

An atomic sentence is true in a given model, under a given interpretation, if the relation referred to by the predicate symbol holds among the objects referred to by the arguments

Complex sentences Complex sentences can be constructed using logical Connectives, just as in propositional calculus. For Example:

- ✓ ¬Brother (LeftLeg(Richard), John)
- ✓ Brother (Richard , John) ∧ Brother (John, Richard)
- ✓ King(Richard) ∨ King(John)
- ✓ ¬King(Richard) ⇒ King(John)

Quantifiers

Quantifiers express properties of entire collections of objects, instead of enumerating the objects by name.
First-order logic contains two standard quantifiers:

1. Universal Quantifier
2. Existential Quantifier

Universal Quantifier
Universal quantifier is defined as follows:
"Given a sentence ∀x P, where P is any logical expression, says that P is true for every object x."
More precisely, ∀x P is true in a given model if P is true in all possible **extended interpretations** constructed from the interpretation given in the model, where each extended interpretation specifies a domain element to which x refers.

For Example: "All kings are persons," is written in first-order logic as
∀x King(x) ⇒Person(x) .
∀ is usually pronounced "For all"

Thus, the sentence says, "For all x, if x is a king, then x is a person." The symbol x is called a variable. Variables are lowercase letters. A variable is a term all by itself, and can also serve as the argument of a function A term with no variables is called a ground term.

Assume we can extend the interpretation in different ways: x→ Richard the Lionheart, x→ King John, x→ Richard's left leg, x→ John's left leg, x→ the crown

The universally quantified sentence ∀x King(x) ⇒Person(x) is true in the original model if the sentence King(x) ⇒Person(x) is true under each of the five extended interpretations. That is, the universally quantified sentence is equivalent to asserting the following five sentences:

Richard the Lionheart is a king ⇒Richard the Lionheart is a person. King John is a king ⇒King John is a person. Richard's left leg is a king ⇒Richard's left leg is a person. John's left leg is a king ⇒John's left leg is a person. The crown is a king ⇒the crown is a person.

Existential quantification (∃)

Universal quantification makes statements about every object. Similarly, we can make a statement about some object in the universe without naming it, by using an existential quantifier.

"The sentence ∃x P says that P is true for at least one object x. More precisely, ∃x P is true in a given model if P is true in at least one extended interpretationthat assigns x to a domain element." ∃x is pronounced "There exists an x such that . . ." or "For some x . . .".

For example, that King John has a crown on his head, we write ∃xCrown(x) AOnHead(x, John)

Given assertions:

Richard the Lionheart is a crown ARichard the Lionheart is on John's head; King John is a crown AKing John is on John's head; Richard's left leg is a crown ARichard's left leg is on John's head; John's left leg is a crown AJohn's left leg is on John's head; The crown is a crown Athe crown is on John's head. The fifth assertion is true in the model, so the original existentially quantified sentence is true in the model. Just as ⇒appears to be the natural connective to use with ∀, Ais the natural connective to use with ∃.

Nested quantifiers

One can express more complex sentences using multiple quantifiers.

For example, "Brothers are siblings" can be written as ∀x∀y Brother (x, y) ⇒Sibling(x, y). Consecutive quantifiers of the same type can be written as one quantifier with several variables.

For example, to say that siblinghood is a symmetric relationship,

we can write∀x, y Sibling(x, y) ⇔Sibling(y, x).

In other cases we will have mixtures.

For example: 1. "Everybody loves somebody" means that for every person, there is someone that person loves: ∀x∃y Loves(x, y) . 2. On the other hand, to say "There is someone who is loved by everyone," we write ∃y∀x Loves(x, y) .

Connections between ∀and ∃

Universal and Existential quantifiers are actually intimately connected with each other, through negation.

Example assertions: 1. " Everyone dislikes medicine" is the same as asserting " there does not exist someone who likes medicine" , and vice versa: "∀x ¬ Likes(x, medicine)" is equivalent to "¬ ∃x Likes(x, medicine)". 2. "Everyone likes ice cream" means that " there is no one who does not like ice cream" : ∀xLikes(x, IceCream) is equivalent to ¬ ∃x ¬ Likes(x, IceCream) .

Because ∀is really a conjunction over the universe of objects and ∃is a disjunction that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

Because ∀is really a conjunction over the universe of objects and ∃is a disjunction that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\forall x\ \neg P \equiv \neg\exists x\ P \qquad \neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg\forall x\ P \equiv \exists x\ \neg P \qquad \neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\forall x\ P \equiv \neg\exists x\ \neg P \qquad P \wedge Q \equiv \neg(\neg P \vee \neg Q)$$
$$\exists x\ P \equiv \neg\forall x\ \neg P \qquad P \vee Q \equiv \neg(\neg P \wedge \neg Q)\ ..$$

Thus, Quantifiers are important in terms of readability.

Equality

First-order logic includes one more way to make atomic sentences, other than using a predicateand terms .We can use the equality symbol to signify that two terms refer to the same object.

For example,

"Father(John) =Henry" says that the object referred to by Father (John) and the object referred to by Henry are the same.

Because an interpretation fixes the referent of any term, determining the truth of an equality sentence is simply a matter of seeing that the referents of the two terms are the same object.The equality symbol can be used to state facts about a given function.It can also be used with negation to insist that two terms are not the same object.

For example,

"Richard has at least two brothers" can be written as, ∃x, y Brother (x,Richard) ABrother (y,Richard) A¬(x=y) .

The sentence

∃x, y Brother (x,Richard) ABrother (y,Richard) does not have the intended meaning. In particular, it is true only in the model where Richard has only one brother considering the extended interpretation in which both x and y are assigned to King John. The addition of ¬ (x=y) rules out such models.

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*Predicate* | *Predicate*(*Term*, ...) | *Term* = *Term*
ComplexSentence	→	(*Sentence*) | [*Sentence*]
	|	¬ *Sentence*
	|	*Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence*
	|	*Sentence* ⇒ *Sentence*
	|	*Sentence* ⇔ *Sentence*
	|	*Quantifier Variable*, ... *Sentence*
Term	→	*Function*(*Term*, ...)
	|	*Constant*
	|	*Variable*
Quantifier	→	∀ | ∃
Constant	→	*A* | X_1 | *John* | ···
Variable	→	*a* | *x* | *s* | ···
Predicate	→	*True* | *False* | *After* | *Loves* | *Raining* | ···
Function	→	*Mother* | *LeftLeg* | ···
OPERATOR PRECEDENCE	:	¬, =, ∧, ∨, ⇒, ⇔

Backus Naur Form for First Order Logic

USING FIRST ORDER LOGIC Assertions and queries in first-order logic

Assertions:

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such sentences are called assertions.

For example,

John is a king, TELL (KB, King (John)). Richard is a person. TELL (KB, Person (Richard)). All kings are persons: TELL (KB, ∀x King(x) ⇒Person(x)).

Asking Queries:

We can ask questions of the knowledge base using ASK. Questions asked with ASK are called queries or goals.

For example,

ASK (KB, King (John)) returns true.

Anyquery that is logically entailed by the knowledge base should be answered affirmatively.

Forexample, given the two preceding assertions, the query:

"ASK (KB, Person (John))" should also return true.

Substitution or binding list

We can ask quantified queries, such as ASK (KB, ∃x Person(x)) .

The answer is true, but this is perhaps not as helpful as we would like. It is rather like answering "Can you tell me the time?" with "Yes."

If we want to know what value of x makes the sentence true, we will need a different function, ASKVARS, which we call with ASKVARS (KB, Person(x)) and which yields a stream of answers.

In this case there will be two answers: {x/John} and {x/Richard}. Such an answer is called a substitution or binding list.

ASKVARS is usually reserved for knowledge bases consisting solely of Horn clauses, because in such knowledge bases every way of making the query true will bind the variables to specific values.

The kinship domain

The objects in Kinship domain are people.

We have two unary predicates, Male and Female.

Kinship relations—parenthood, brotherhood, marriage, and so on—are represented by binary predicates: Parent, Sibling, Brother,Sister,Child, Daughter, Son, Spouse, Wife, Husband, Grandparent,Grandchild, Cousin, Aunt, and Uncle.

We use functions for Mother and Father, because every person has exactly one of each of these.

We can represent each function and predicate, writing down what we know in termsof the other symbols.

For example:- 1. one's mother is one's female parent: ∀m, c Mother (c)=m ⇔Female(m) AParent(m, c) .

2. One's husband is one's male spouse: ∀w, h Husband(h,w) ⇔Male(h) ASpouse(h,w) .

3. Male and female are disjoint categories: ∀xMale(x) ⇔¬Female(x) .

4. Parent and child are inverse relations: ∀p, c Parent(p, c) ⇔Child (c, p) .

5. A grandparent is a parent of one's parent: ∀g, c Grandparent (g, c) ⇔∃p Parent(g, p) AParent(p, c) .

6. A sibling is another child of one's parents: ∀x, y Sibling(x, y) ⇔x _= y A∃p Parent(p, x) AParent(p, y) .

Axioms:

Each of these sentences can be viewed as an axiom of the kinship domain. Axioms are commonly associated with purely mathematical domains. They provide the basic factual information from which useful conclusions can be derived.

Kinship axioms are also definitions; they have the form ∀x, y P(x, y) ⇔. . ..

The axioms define the Mother function, Husband, Male, Parent, Grandparent, and Sibling predicates in terms of other predicates.

Our definitions "bottom out" at a basic set of predicates (Child, Spouse, and Female) in terms of which the others are ultimately defined. This is a natural way in which to build up the representation of a domain, and it is analogous to the way in which software packages are built up by successive definitions of subroutines from primitive library functions.

Theorems:

Not all logical sentences about a domain are axioms. Some are theorems—that is, they are entailed by the axioms.

For example, consider the assertion that siblinghood is symmetric: ∀x, y Sibling(x, y) ⇔Sibling(y, x) .

It is a theorem that follows logically from the axiom that defines siblinghood. If we ASK the knowledge base this sentence, it should return true. From a purely logical point of view, a knowledge base need contain only axioms and no theorems, because the theorems do not increase the set of conclusions that follow from the knowledge base. From a practical point of view, theorems are essential to reduce the computational cost of deriving new sentences. Without them, a reasoning system has to start from first principles every time.

Axioms Axioms without Definition

Not all axioms are definitions. Some provide more general information about certain predicates without constituting a definition. Indeed, some predicates have no complete definition because we do not know enough to characterize them fully.

For example, there is no obvious definitive way to complete the sentence

$\forall x \text{Person}(x) \Leftrightarrow \ldots$

Fortunately, first-order logic allows us to make use of the Person predicate without completely defining it. Instead, we can write partial specifications of properties that every person has and properties that make something a person:

$\forall x \text{Person}(x) \Rightarrow \ldots \forall x \ldots \Rightarrow \text{Person}(x)$.

Axioms can also be "just plain facts," such as Male (Jim) and Spouse (Jim, Laura).Such facts form the descriptions of specific problem instances, enabling specific questions to be answered. The answers to these questions will then be theorems that follow from the axioms

Numbers, sets, and lists

Number theory

Numbers are perhaps the most vivid example of how a large theory can be built up from NATURAL NUMBERS a tiny kernel of axioms. We describe here the theory of natural numbers or non-negative integers. We need:

□ predicate NatNum that will be true of natural numbers;□one PEANO AXIOMS constant symbol, 0;□ One function symbol, S (successor). □ The Peano axioms define natural numbers and addition.

Natural numbers are defined recursively: NatNum(0) . $\forall n$ NatNum(n) $\Rightarrow$ NatNum(S(n)) .

That is, 0 is a natural number, and for every object n, if n is a natural number, then S(n) is a natural number.

So the natural numbers are 0, S(0), S(S(0)), and so on. We also need axioms to constrain the successor function: ∀n 0 != S(n) . ∀m, n m != n ⇒ S(m) != S(n) .

Now we can define addition in terms of the successor function: ∀m NatNum(m) ⇒ + (0, m) = m . ∀m, n NatNum(m) A NatNum(n) ⇒ + (S(m), n) = S(+(m, n))

The first of these axioms says that adding 0 to any natural number m gives m itself. Addition is represented using the binary function symbol "+" in the term + (m, 0);

To make our sentences about numbers easier to read, we allow the use of infix notation. We can also write S(n) as n + 1, so the second axiom becomes :

∀m, n NatNum (m) A NatNum(n) ⇒ (m + 1) + n = (m + n)+1 .

This axiom reduces addition to repeated application of the successor function. Once we have addition, it is straightforward to define multiplication as repeated addition, exponentiation as repeated multiplication, integer division and remainders, prime numbers, and so on. Thus, the whole of number theory (including cryptography) can be built up from one constant, one function, one predicate and four axioms.

Sets

The domain of sets is also fundamental to mathematics as well as to commonsense reasoning. Sets can be represented as individualsets, including empty sets.

Sets can be built up by: adding an element to a set or Taking the union or intersection of two sets.

Operations that can be performed on sets are: To know whether an element is a member of a set Distinguish sets from objects that are not sets.

Vocabulary of set theory:

The empty set is a constant written as { }. There is one unary predicate, Set, which is true of sets. The binary predicates are

 x∈ s (x is a member of set s) s1 ⊆ s2 (set s1 is a subset, not necessarily proper, of set s2).

The binary functions are

 s1 ∩ s2 (the intersection of two sets), s1 U s2 (the union of two sets), and {x|s} (the set resulting from adjoining element x to set s).

One possible set of axioms is as follows:

 The only sets are the empty set and those made by adjoining something to a set: ∀sSet(s) ⇔(s={}) ∨(∃x, s2 Set(s2) As={x|s2}) . The empty set has no elements adjoined into it. In other words, there is no way to decompose {} into a smaller set and an element: ¬∃x, s {x|s}={} . Adjoining an element already in the set has no effect: ∀x, s x∈s ⇔s={x|s} . The only members of a set are the elements that were adjoined into it. We express this recursively, saying that x is a member of s if and only if s is equal to some set s2 adjoined with some element y, where either y is the same as x or x is a member of s2: ∀x, s x∈s ⇔∃y, s2 (s={y|s2} A(x=y ∨x∈s2)) A set is a subset of another set if and only if all of the first set's members are members of the second set: ∀s1, s2 s1 ⊆s2 ⇔(∀x x∈s1 ⇒x∈s2) Two sets are equal if and only if each is a subset of the other: ∀s1, s2 (s1 =s2) ⇔(s1 ⊆s2 As2 ⊆s1)

 An object is in the intersection of two sets if and only if it is a member of both sets: ∀x, s1, s2 x∈(s1 ∩ s2) ⇔(x∈s1 Ax∈s2) An object is in the union of two sets if and only if it is a member of either set: ∀x, s1, s2 x∈(s1 Us2) ⇔(x∈s1 ∨x∈s2)

Lists : are similar to sets. The differences are that lists are ordered and the same element canappear more than once in a list. We can use the vocabulary of Lisp for lists:

 Nil is the constant list with no elements; Cons, Append, First, and Rest are functions; Find is the predicate that does for lists what Member does for sets. List? is a predicate that is true only of lists. The empty list is [ ]. The term Cons(x, y), where y is a nonempty list, is written [x|y]. The term Cons(x, Nil) (i.e., the list containing the element x) is written as [x]. A list of several elements, such as [A,B,C], corresponds to the nested term Cons(A, Cons(B, Cons(C, Nil))).

The wumpus world

Agents Percepts and Actions

The wumpus agent receives a percept vector with five elements. The corresponding first-order sentence stored in the knowledge base must include both the percept and the time at which it occurred; otherwise, the agent will get confused about when it saw what.We use integers for time steps. A typical percept sentence would be

Percept ([Stench, Breeze, Glitter,None, None], 5).

Here, Percept is a binary predicate, and Stench and so on are constants placed in a list. The actions in the wumpus world can be represented by logical terms:

Turn (Right), Turn (Left), Forward,Shoot,Grab, Climb.

To determine which is best, the agent program executes the query:

ASKVARS (∃a BestAction (a, 5)), which returns a binding list such as {a/Grab}.

The agent program can then return Grab as the action to take.

The raw percept data implies certain facts about the current state.

For example: ∀t, s, g, m, c Percept ([s, Breeze, g,m, c], t) ⇒Breeze(t) , ∀t, s, b, m, c Percept ([s, b, Glitter,m, c], t) ⇒Glitter (t) ,

UNIT III – Knowledge and Reasoning

These rules exhibit a trivial form of the reasoning process called perception.

Simple "reflex" behavior can also be implemented by quantified implication sentences.

For example, we have ∀tGlitter (t) ⇒BestAction(Grab, t) .

Given the percept and rules from the preceding paragraphs, this would yield the desired conclusion Best Action (Grab, 5)—that is, Grab is the right thing to do.

Environment Representation

Objects are squares, pits, and the wumpus. Each square could be named—Square1,2and so on—but then the fact that Square1,2and Square1,3 are adjacent would have to be an "extra"

fact, and this needs one suchfact for each pair of squares. It is better to use a complex term in which the row and columnappear as integers;

For example, we can simply use the list term [1, 2].

Adjacency of any two squares can be defined as:

∀x, y, a, b Adjacent ([x, y], [a, b]) ⇔ (x = a A(y = b − 1 ∨y = b + 1)) ∨(y = b A(x = a − 1 ∨x = a + 1)).

Each pit need not be distinguished with each other. The unary predicate Pit is true of squares containing pits.

Since there is exactly one wumpus, a constant Wumpus is just as good as a unary predicate. The agent's location changes over time, so we write At (Agent, s, t) to mean that theagent is at square s at time t.

To specify the Wumpus location (for example) at [2, 2] we can write ∀t At (Wumpus, [2, 2], t).

Objects can only be at one location at a time: ∀x, s1, s2, t At(x, s1, t) AAt(x, s2, t) ⇒s1 = s2 .

Given its current location, the agent can infer properties of the square from properties of its current percept.

For example, if the agent is at a square and perceives a breeze, then that square is breezy:

∀s, t At(Agent, s, t) ABreeze(t) ⇒Breezy(s) .

It is useful to know that a square is breezy because we know that the pits cannot move about.

Breezy has no time argument.

Having discovered which places are breezy (or smelly) and, very importantly, not breezy (or not smelly), the agent can deduce where the pits =e (and where the wumpus is).

There are two kinds of synchronic rules that could allow such deductions:

Diagnostic rules:

Diagnostic rules lead from observed effects to hidden causes. For finding pits, the obvious diagnostic rules say that if a square is breezy, some adjacent square must contain a pit, or

∀s Breezy(s) ⇒∃r Adjacent (r, s)APit(r) ,

and that if a square is not breezy, no adjacent square contains a pit: ∀s¬Breezy (s) ⇒¬∃r Adjacent (r, s) A Pit (,r) .Combining these two, we obtain the biconditional sentence ∀s Breezy (s)⇔∃r Adjacent(r, s) A Pit (r) .

Causal rules:

Causal rules reflect the assumed direction of causality in the world: some hidden property of the world causes certain percepts to be generated. For example, a pit causes all adjacent squares to be breezy:

and if all squares adjacent to a given square are pitless, the square will not be breezy: ∀s[∀r Adjacent (r, s) ⇒¬Pit (r)] ⇒¬Breezy (s) .

It is possible to show that these two sentences together are logically equivalent to the biconditional sentence " ∀s Breezy (s)⇔∃r Adjacent(r, s) A Pit (r)" .

The biconditional itself can also be thought of as causal, because it states how the truth value of Breezy is generated from the world state.

Systems that reason with causal rules are called model-based reasoning systems, because the causal rules form a model of how the environment operates.

Whichever kind of representation the agent uses, ifthe axioms correctly and completely describe the way the world works and the way that percepts are produced, then any complete logical inference procedure will infer the strongest possible description of the world state, given the available percepts. Thus, the agent designer can concentrate on getting the knowledgeright, without worrying too much about the processes of deduction.

Inference in First-Order Logic

Propositional Vs First Order Inference

Earlier inference in first order logic is performed with *Propositionalization* which is a process of converting the Knowledgebase present in First Order logic into Propositional logic and on that using any inference mechanisms of propositional logic are used to check inference.

Inference rules for quantifiers:

There are some Inference rules that can be applied to sentences with quantifiers to obtain sentences without quantifiers**.** These rules will lead us to make the conversion.

Universal Instantiation (UI):

The rule says that we can infer any sentence obtained by substituting a **ground term** (a term without variables) for the variable. Let SUBST (θ*)* denote the result of applying the substitution θ to the sentence *a.* Then the rule is written

$$\frac{\forall v \ \ a}{\text{SUBST}(\{v/g\}, \alpha)}$$

For any variable v and ground term *g.*

For example, there is a sentence in knowledge base stating that all greedy kings are Evils

$$\forall x \ \ King(x) \ A \ Greedy(x) \Rightarrow \ Evil(x) \ .$$

For the variable x, with the substitutions like {x/John},{x/Richard}the following sentences can be inferred.

$$King(John) \ A \ Greedy(John) \ \Rightarrow \ Evil(John).$$
$$King(Richard) \ A \ Greedy(Richard) \ \Rightarrow \ Evil(Richard) \ .$$

Thus a universally quantified sentence can be replaced by the set of *all* possible instantiations.

Existential Instantiation (EI)*:*

The existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object, that name must not already belong to another object. This new name is called a **Skolem constant.** Existential Instantiation is a special case of a more general process called *"skolemization".*

For any sentence *a,* variable v, and constant symbol *k* that does not appear elsewhere in the knowledge base,

$$\frac{\exists v \ \ \alpha}{\text{SUBST}(\{v/k\}, \alpha)}$$

For example, from the sentence

$$3x \ \ Crown(x) \ A \ OnHead(x, John)$$

So, we can infer the sentence

$$Crown(C_1) \ A \ OnHead(C_1, John)$$

As long as C_1 does not appear elsewhere in the knowledge base. Thus an existentially quantified sentence can be replaced by one instantiation

Elimination of Universal and Existential quantifiers should give new knowledge base which can be shown to be *inferentially equivalent* to old in the sense that it is satisfiable exactly when the original knowledge base is satisfiable.

Reduction to propositional inference:

Once we have rules for inferring non quantified sentences from quantified sentences, it becomes possible to reduce first-order inference to propositional inference. For example, suppose our knowledge base contains just the sentences

$$\forall x \ \ King(x) A \ Greedy(x) \ \Rightarrow \ \ Evil(x)$$
$$King(John)$$
$$Greedy(John)$$
$$Brother(Richard, John).$$

Then we apply UI to the first sentence using all possible ground term substitutions from the vocabulary of the knowledge base-in this case, *{xl John)* and *{x/Richard).* We obtain

$$King(John) A \ Greedy(John) \Rightarrow \ Evil(John),$$
$$King(Richard) A \ Greedy(Richard) \Rightarrow \ Evil(Richard)$$

We discard the universally quantified sentence. Now, the knowledge base is essentially propositional if we view the ground atomic sentences-King *(John), Greedy (John),* and Brother (Richard, *John*) as proposition symbols. Therefore, we can apply any of the complete propositional algorithms to obtain conclusions such as *Evil (John).*

Disadvantage:

If the knowledge base includes a function symbol, the set of possible ground term substitutions is infinite. Propositional algorithms will have difficulty with an infinitely large set of sentences.

NOTE:

Entailment for first-order logic is *semi decidable* which means algorithms exist that say yes to every entailed sentence, but no algorithm exists that also says no to every non entailed sentence

Unification and Lifting

Consider the above discussed example, if we add Siblings (Peter, Sharon) to the knowledge base then it will be

$\forall x\ King(x) \wedge Greedy(x) \Rightarrow Evil(x)$
$King(John)$
$Greedy(John)$
$Brother(Richard, John)$
$\mathbf{Siblings(Peter, Sharon)}$

Removing Universal Quantifier will add new sentences to the knowledge base which are not necessary for the query *Evil (John)?*

$King(John) \wedge Greedy(John) \Rightarrow Evil(John)$
$King(Richard) \wedge Greedy(Richard) \Rightarrow Evil(Richard)$
$King(Peter) \wedge Greedy(Peter) \Rightarrow Evil(Peter)$
$King(Sharon) \wedge Greedy(Sharon) \Rightarrow Evil(Sharon)$

Hence we need to teach the computer to make better inferences. For this purpose Inference rules were used.

First Order Inference Rule:

The key advantage of lifted inference rules over *propositionalization* is that they make only those substitutions which are required to allow particular inferences to proceed.

Generalized Modus Ponens:

If there is some substitution **θ** that makes the premise of the implication identical to sentences already in the knowledge base, then we can assert the conclusion of the implication, after applying **θ**. This inference process can be captured as a single inference rule called General ized Modus Ponens which is a ***lifted*** version of Modus Ponens-it raises Modus Ponens from propositional to first-order logic

For atomic sentences pi, pi ', and q, where there is a substitution θ such that SUBST(θ , pi) = SUBST(θ , pi '), for all i,

$$\mathbf{p1',\ p2',\ \ldots,\ pn',\ (p1 \boxtimes p2 \boxtimes \ldots \boxtimes pn \Rightarrow q)}$$

$$\mathbf{SUBST\ (\theta,\ q)}$$

There are N + 1 premises to this rule, N atomic sentences + one implication.

Applying SUBST (θ, q) yields the conclusion we seek. It is a sound inference rule.

Suppose that instead of knowing Greedy (John) in our example we know that everyone is greedy:

$$\forall \mathbf{y\ Greedy(y)}$$

We would conclude that Evil(John).

Applying the substitution {x/John, y / John) to the implication premises King (x) and Greedy (x) and the knowledge base sentences King(John) and Greedy(y) will make them identical. Thus, we can infer the conclusion of the implication.

For our example,

p_1' is $King(John)$ p_1 is $King(x)$
p_2' is $Greedy(y)$ p_2 is $Greedy(x)$
θ is $\{x/John, y/John)$ q is $Evil(x)$
SUBST(θ, q) is $Evil(John)$.

Unification:

It is the process used to find substitutions that make different logical expressions look identical. **Unification** is a key component of all first-order Inference algorithms.

UNIFY (p, q) = θ where SUBST (θ, p) = SUBST (θ, q) θ is our unifier value (if one exists).

Ex: "Who does John know?"

UNIFY (Knows (John, x), Knows (John, Jane)) = {x/ Jane}.
UNIFY (Knows (John, x), Knows (y, Bill)) = {x/Bill, y/ John}.
UNIFY (Knows (John, x), Knows (y, Mother(y))) = {x/Bill, y/ John}
UNIFY (Knows (John, x), Knows (x, Elizabeth)) = FAIL

- The last unification fails because both use the same variable, X. X can't equal both John and Elizabeth. To avoid this change the variable X to Y (or any other value) in Knows(X, Elizabeth)

Knows(X, Elizabeth) → Knows(Y, Elizabeth)

Still means the same. This is called **standardizing apart.**

- sometimes it is possible for more than one unifier returned:

UNIFY (Knows (John, x), Knows(y, z)) =???

This can return two possible unifications: {y/ John, x/ z} which means Knows (John, z) OR {y/ John, x/ John, z/ John}. For each unifiable pair of expressions there is a single **most general unifier (MGU)**, In this case it is *{y/ John, x/z)*.

An algorithm for computing most general unifiers is shown below.

```
function UNIFY(x, y, θ) returns a substitution to make x and y identical
  inputs: x, a variable, constant, list, or compound
          y, a variable, constant, list, or compound
          θ, the substitution built up so far (optional, defaults to empty)

  if θ = failure then return failure
  else if x = y then return θ
  else if VARIABLE?(x) then return UNIFY-VAR(x, y, θ)
  else if VARIABLE?(y) then return UNIFY-VAR(y, x, θ)
  else if COMPOUND?(x) and COMPOUND?(y) then
      return UNIFY(ARGS[x], ARGS[y], UNIFY(OP[x], OP[y], θ))
  else if LIST?(x) and LIST?(y) then
      return UNIFY(REST[x], REST[y], UNIFY(FIRST[x], FIRST[y], θ))
  else return failure
```

```
function UNIFY-VAR(var, x, θ) returns a substitution
  inputs: var, a variable
          x, any expression
          θ, the substitution built up so far

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

Figure 2.1 *The unification algorithm. The algorithm works by comparing the structures of the inputs, element by element. The substitution 0 that is the argument to UNIFY is built up along the way and is used to make sure that later comparisons are consistent with bindings that were established earlier. In a compound expression, such as F (A, B), the function OP picks out the function symbol F and the function ARCS picks out the argument list (A, B).*

The process is very simple: recursively explore the two expressions simultaneously "side by side," building up a unifier along the way, but failing if two corresponding points in the structures do not match. **Occur check** step makes sure same variable isn't used twice.

Storage and retrieval

- STORE(s) stores a sentence *s* into the knowledge base
- FETCH(s) returns all unifiers such that the query q unifies with some sentence in the knowledge base.

Easy way to implement these functions is Store all sentences in a long list, browse list one sentence at a time with UNIFY on an ASK query. But this is inefficient.

To make FETCH more efficient by ensuring that unifications are attempted only with sentences that have *some* chance of unifying. (i.e. Knows(John, x) vs. Brother(Richard, John) are not compatible for unification)

- To avoid this, a simple scheme called ***predicate indexing*** puts all the *Knows* facts in one bucket and all the *Brother* facts in another.
- The buckets can be stored in a hash table for efficient access. Predicate indexing is useful when there are many predicate symbols but only a few clauses for each symbol.

But if we have many clauses for a given predicate symbol, facts can be stored under multiple index keys.

For the fact *Employs (AIMA.org, Richard)*, the queries are
Employs (A IMA. org, Richard) Does AIMA.org employ Richard?
Employs (x, Richard) who employs Richard?
Employs (AIMA.org, y) whom does AIMA.org employ?
Employs Y(x), who employs whom?

We can arrange this into a **subsumption lattice,** as shown below**.**

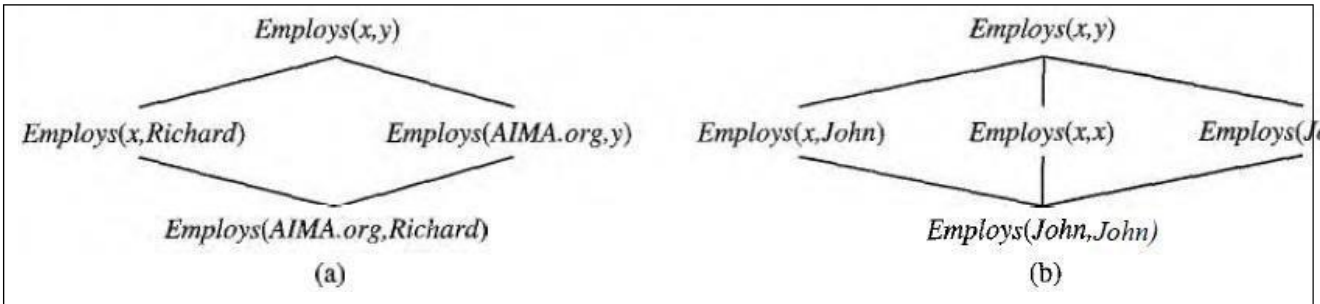


Figure 2.2 *(a) The subsumption lattice whose lowest node is the sentence Employs (AIMA.org, Richard). (b) The subsumption lattice for the sentence Employs (John, John).*

A subsumption lattice has the following properties:

- ✓ child of any node obtained from its parents by one substitution

- ✓ the "highest" common descendant of any two nodes is the result of applying their most general unifier
- ✓ predicate with n arguments contains O(2n) nodes (in our example, we have two arguments, so our lattice has four nodes)
- ✓ Repeated constants = slightly different lattice.

Forward Chaining

First-Order Definite Clauses:

A definite clause either is atomic or is an implication whose antecedent is a conjunction of positive literals and whose consequent is a single positive literal. The following are first-order definite clauses:

$$King(x) \wedge Greedy(x) \Rightarrow Evil(x) .$$
$$King(John) .$$
$$Greedy(y) .$$

Unlike propositional literals, first-order literals can include variables, in which case those variables are assumed to be universally quantified.

Consider the following problem;

"The law says that it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles, and all of its missiles were sold to it by Colonel West, who is American."

We will represent the facts as first-order definite clauses

". . . It is a crime for an American to sell weapons to hostile nations":

$$American(x) \wedge Weapon(y) \wedge Sells(x, y, z) \wedge Hostile(z) \Rightarrow Criminal(x)$$ --------- (1)

"Nono . . . has some missiles." The sentence 3 x *Owns (Nono,* .rc) *A Missile* (x) is transformed into two definite clauses by Existential Elimination, introducing a new constant *M1:*

Owns (Nono, M1)---------- (2)

Missile (Ml) ---------- (3*)*

"All of its missiles were sold to it by Colonel West":

Missile (x) A Owns (Nono, x) =>Sells (West, z, Nono) ----------(4)

We will also need to know that missiles are weapons:

Missile (x) => Weapon (x) ---------- (5)

We must know that an enemy of America counts as "hostile":

***Enemy (x, America) =>Hostile(x)*----------(6)**

"West, who is American":

***American (West)*----------(7)**

"The country Nono, an enemy of America ":

Enemy (Nono, America) ---------- (8)

A simple forward-chaining algorithm:

- Starting from the known facts, it triggers all the rules whose premises are satisfied, adding their conclusions lo the known facts
- The process repeats until the query is answered or no new facts are added. Notice that a fact is not "new" if it is just *renaming* of a known fact.

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (4), (5), and (6). Two iterations are required:

- ✓ On the first iteration, rule (1) has unsatisfied premises.

Rule (4) is satisfied with *{x/Ml),* and *Sells (West,* M1, *Nono)* is added.

Rule (5) is satisfied with *{x/M1) an*d *Weapon (M1)* is added.

Rule (6) is satisfied with *{x/Nono},* and *Hostile (Nono)* is added.

- ✓ On the second iteration, rule (1) is satisfied with *{x/West, Y/MI, z /Nono),* and *Criminal* (*West)* is added.

It is ***sound,*** because every inference is just an application of Generalized Modus Ponens, it is ***complete*** for definite clause knowledge bases; that is, it answers every query whose answers are entailed by any knowledge base of definite clauses

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

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

Figure 3.1 A *conceptually straightforward, but very inefficient, forward-chaining algorithm. On each iteration, it adds to KB all the atomic sentences that can be inferred in one step from the implication sentences and the atomic sentences already in KB.*

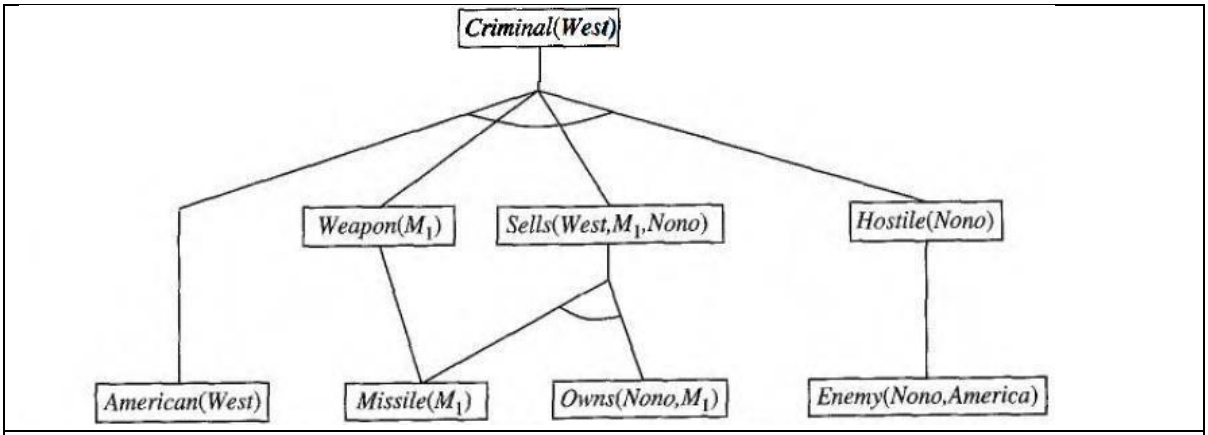


Figure 3.2 *The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.*

Efficient forward chaining:

The above given forward chaining algorithm was lack with efficiency due to the the three sources of complexities:

- ✓ Pattern Matching
- ✓ Rechecking of every rule on every iteration even a few additions are made to rules
- ✓ Irrelevant facts

1. ***Matching rules against known facts:***

For example, consider this rule,

Missile(x) A Owns (Nono, x) => Sells (West, x, Nono).

The algorithm will check all the objects owned by Nono in and then for each object, it could check whether it is a missile. This is the ***conjunct ordering problem:***

"Find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized". The **most constrained variable** heuristic used for CSPs would suggest ordering the conjuncts to look for missiles first if there are fewer missiles than objects that are owned by Nono.

The connection between pattern matching and constraint satisfaction is actually very close. We can view each conjunct as a constraint on the variables that it contains-for example, Missile(x) is a unary constraint on x. Extending this idea, we can express every finite-domain CSP as a single definite clause together with some associated ground facts. Matching a definite clause against a set of facts is NP-hard

2. Incremental forward chaining:

On the second iteration, the rule ***Missile (x) => Weapon (x)***

Matches against Missile (M1) (again), and of course the conclusion Weapon(x/M1) is already known so nothing happens. Such redundant rule matching can be avoided if we make the following observation:

"Every new fact inferred on iteration t must be derived from at least one new fact inferred on iteration t – 1".

This observation leads naturally to an incremental forward chaining algorithm where, at iteration t, we check a rule only if its premise includes a conjunct p, that unifies with a fact p: newly inferred at iteration t - 1. The rule matching step then fixes p, to match with p', but allows the other conjuncts of the rule to match with facts from any previous iteration.

3. Irrelevant facts:

- One way to avoid drawing irrelevant conclusions is to use backward chaining.
- Another solution is to restrict forward chaining to a selected subset of rules

- A third approach, is to rewrite the rule set, using information from the goal.so that only relevant variable bindings-those belonging to a so-called **magic** set-are considered during forward inference.

For example, if the goal is Criminal (West), the rule that concludes Criminal (x) will be rewritten to include an extra conjunct that constrains the value of x:

Magic(x) A American(z) A Weapon(y)A Sells(x, y, z) A Hostile(z) =>Criminal(x)

The fact *Magic (West)* is also added to the KB. In this way, even if the knowledge base contains facts about millions of Americans, only Colonel West will be considered during the forward inference process.

4. Backward Chaining

This algorithm work backward from the goal, chaining through rules to find known facts that support the proof. It is called with a list of goals containing the original query, and returns the set of all substitutions satisfying the query. The algorithm takes the first goal in the list and finds every clause in the knowledge base whose **head,** unifies with the goal. Each such clause creates a new recursive call in which **body,** of the clause is added to the goal stack .Remember that facts are clauses with a head but no body, so when a goal unifies with a known fact, no new sub goals are added to the stack and the goal is solved. The algorithm for backward chaining and proof tree for finding criminal (West) using backward chaining are given below.

function FOL-BC-ASK(KB, *goals*, θ) **returns** a set of substitutions
  inputs: KB, a knowledge base
    goals, a list of conjuncts forming a query (θ already applied)
    θ, the current substitution, initially the empty substitution $\{\}$
  local variables: *answers*, a set of substitutions, initially empty

  if *goals* is empty **then return** $\{\theta\}$
  $q' \leftarrow$ SUBST(θ, FIRST(*goals*))
  for each sentence r **in** KB where STANDARDIZE-APART(r) $= (p_1 \wedge \ldots \wedge p_n \Rightarrow q)$
    and $\theta' \leftarrow$ UNIFY(q, q') succeeds
    new-goals $\leftarrow [p_1, \ldots, p_n | \text{REST}(goals)]$
    answers $\leftarrow$ FOL-BC-ASK(KB, *new-goals*, COMPOSE(θ', θ)) $\cup$ *answers*
  return *answers*

Figure 4.1 *A simple backward-chaining algorithm.*

.

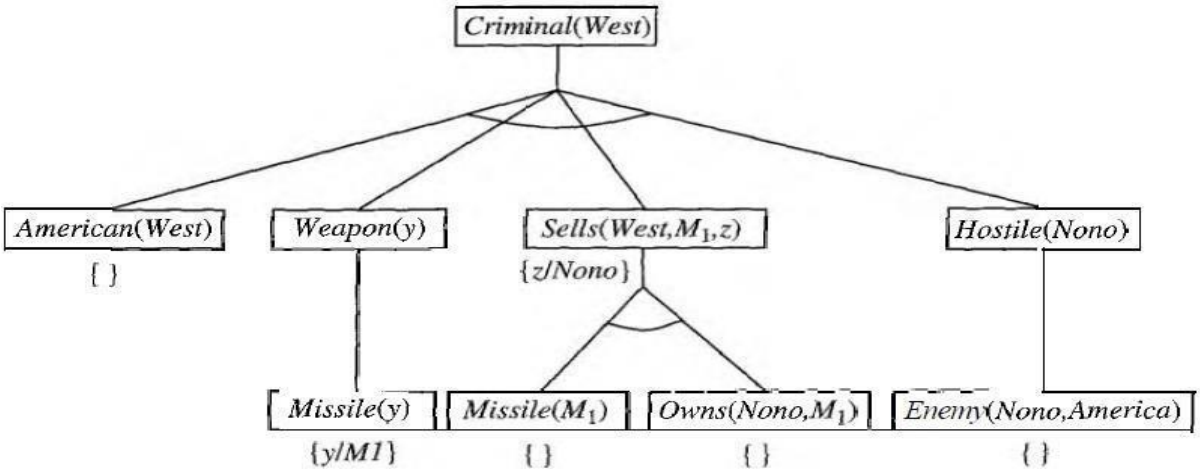


Figure 4.2 *Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. Bindings for each successful unification are shown next to the corresponding sub goal. Note that once one sub goal in a conjunction succeeds, its substitution is applied to subsequent sub goals.*

Logic programming:

- **Prolog** is by far the most widely used logic programming language.
- Prolog programs are sets of definite clauses written in a notation different from standard first-order logic.
- Prolog uses uppercase letters for variables and lowercase for constants.
- Clauses are written with the head preceding the body; " : -" is used for left implication, commas separate literals in the body, and a period marks the end of a sentence

```
criminal(X) :- american(X), weapon(Y), sells(X,Y,Z), hostile(Z)
```

Prolog includes "syntactic sugar" for list notation and arithmetic. Prolog program for append (X, Y, Z), which succeeds if list Z is the result of appending lists x and Y

```
append([],Y,Y).
append([A|X],Y,[A|Z]) :- append(X,Y,Z)
```

For example, we can ask the query append (A, B, [1, 2]): what two lists can be appended to give [1, 2]? We get back the solutions

```
A=[]     B=[1,2]
A=[1]    B=[2]
A=[1,2]  B=[]
```

- The execution of Prolog programs is done via depth-first backward chaining
- Prolog allows a form of negation called **negation as failure.** A negated goal not P is considered proved if the system fails to prove p. Thus, the sentence

Alive (X) : - not dead(X) can be read as "Everyone is alive if not provably dead."

- Prolog has an equality operator, =, but it lacks the full power of logical equality. An equality goal succeeds if the two terms are *unifiable* and fails otherwise. So X+Y=2+3 succeeds with x bound to *2* and Y bound to 3, but Morningstar=evening star fails.
- The occur check is omitted from Prolog's unification algorithm.

Efficient implementation of logic programs:

The execution of a Prolog program can happen in two modes: interpreted and compiled.

- Interpretation essentially amounts to running the FOL-BC-ASK algorithm, with the program as the knowledge base. These are designed to maximize speed.

  First, instead of constructing the list of all possible answers for each sub goal before continuing to the next, Prolog interpreters generate one answer and a "promise" to generate the rest when the current answer has been fully explored. This promise is called a **choice point.** FOL-BC-ASK spends a good deal of time in generating and composing substitutions when a path in search fails. Prolog will backup to previous choice point and unbind some variables. This is called "TRAIL". So, new variable is bound by UNIFY-VAR and it is pushed on to trail.

- Prolog Compilers compile into an intermediate language i.e., Warren Abstract Machine or WAM named after David. H. D. Warren who is one of the implementers of first prolog compiler. So, WAM is an abstract instruction set that is suitable for prolog and can be either translated or interpreted into machine language.

Continuations are used to implement choice point's continuation as packaging up a procedure and a list of arguments that together define what should be done next whenever the current goal succeeds.

- Parallelization can also provide substantial speedup. There are two principal sources of parallelism

1. The first, called **OR-parallelism,** comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel.

a

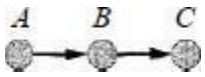
2. The second, called **AND-parallelism,** comes from the possibility of solving each conjunct in the body of an implication in parallel. AND-parallelism is more difficult to achieve, because solutions for the whole conjunction require consistent bindings for all the variables.

Redundant inference and infinite loops:

Consider the following logic program that decides if a path exists between two points on a directed graph.

```
path(X,Z) :- link(X,Z).
path(X,Z) :- path(X,Y), link(Y,Z)
```

A simple three-node graph, described by the facts link (a, b) and link (b, c)



It generates the query path (a, c)

Hence each node is connected to two random successors in the next layer.

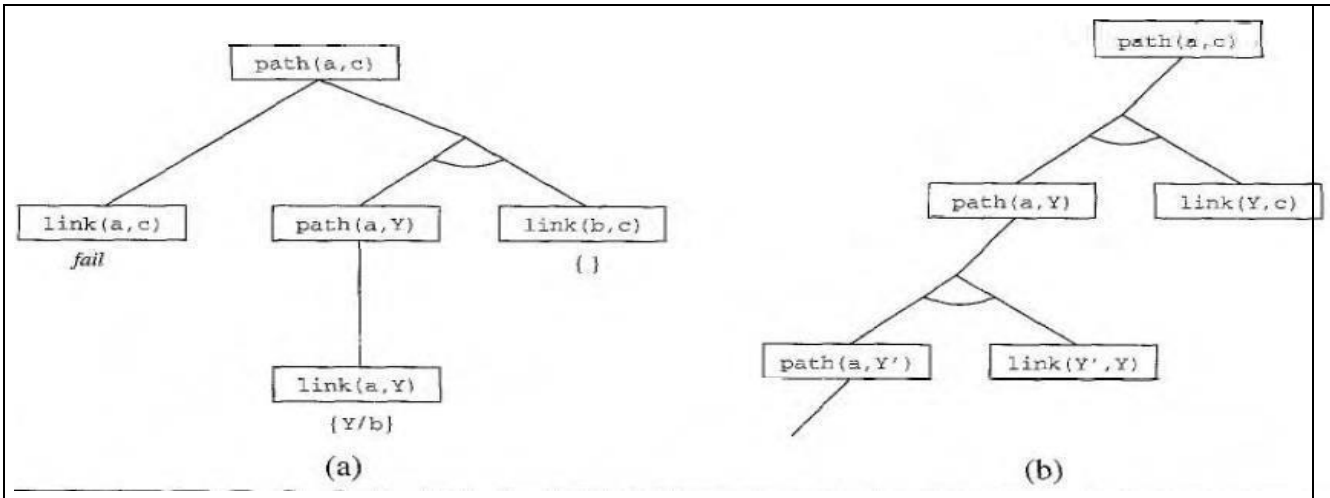


Figure 4.3 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

Constraint logic programming:

The Constraint Satisfaction problem can be solved in prolog as same like backtracking algorithm. Because it works only for finite domain CSP's in prolog terms there must be finite number of solutions for any goal with unbound variables.

```
triangle(X,Y,Z) :-
    X>=0, Y>=0, Z>=0, X+Y>=Z, Y+Z>=X, X+Z>=Y.
```

- If we have a query, triangle (3, 4, and 5) works fine but the query like, triangle (3, 4, Z) no solution.
- The difficulty is variable in prolog can be in one of two states i.e., Unbound or bound.
- Binding a variable to a particular term can be viewed as an extreme form of constraint namely "equality".CLP allows variables to be constrained rather than bound.

The solution to triangle (3, 4, Z) is Constraint 7>=Z>=1.

5. Resolution

As in the propositional case, first-order resolution requires that sentences be in **conjunctive normal form** (CNF) that is, a conjunction of clauses, where each clause is a disjunction of literals.

Literals can contain variables, which are assumed to be universally quantified. Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence. We will illustrate the procedure by translating the sentence

"Everyone who loves all animals is loved by someone," or

$$\forall x \ [\forall y \ Animal(y) \Rightarrow Loves(x, y)] \Rightarrow [\exists y \ Loves(y, x)]$$

The steps are as follows:

- Eliminate implications:

$$ti\, x \ [\neg\forall y \ \neg Animal(y) \vee Loves(x, y)] \vee [\exists y \ Loves(y, x)]$$

- Move Negation inwards: In addition to the usual rules for negated connectives, we need rules for negated quantifiers. Thus, we have

$$\neg\forall x \ p \quad \text{becomes} \quad 3\, x \ \neg p$$
$$\neg\exists x \ p \quad \text{becomes} \quad \forall x \ \neg p$$

Our sentence goes through the following transformations:

$$\forall x \ [\exists y \ \neg(\neg Animal(y) \vee Loves(x,y))] \vee [\exists y \ Loves(y,x)] .$$
$$\forall x \ [\exists y \ \neg\neg Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y \ Loves(y,x)].$$
$$\forall x \ [\exists y \ Animal(y) \wedge \neg Loves(x,y)] \vee [\exists y \ Loves(y,x)] .$$

- Standardize variables: For sentences like $(\forall x \ P(x)) \vee (\exists x \ Q(x))$ which use the same variable name twice, change the name of one of the variables. This avoids confusion later when we drop the quantifiers. Thus, we have

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

- Skolemize: Skolemization is the process of removing existential quantifiers by elimination. Translate *3 x P(x)* into *P(A)*, where *A* is a new constant. If we apply this rule to our sample sentence, however, we obtain

$$\forall x \ [Animal(A) \wedge \neg Loves(x,A)] \vee Loves(B,x)$$

Which has the wrong meaning entirely: it says that everyone either fails to love a particular animal *A* or is loved by some particular entity *B.* In fact, our original sentence allows each person to fail to love a different animal or to be loved by a different person.

Thus, we want the Skolem entities to depend on *x:*

$$\forall x \ [Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee Loves(G(x),x)$$

Here *F* and *G* are Skolem functions. The general rule is that the arguments of the Skolem function are all the universally quantified variables in whose scope the existential quantifier appears.

- Drop universal quantifiers: At this point, all remaining variables must be universally quantified. Moreover, the sentence is equivalent to one in which all the universal quantifiers have been moved to the left. We can therefore drop the universal quantifiers

$$[Animal(F(x)) \wedge \neg Loves(x,F(x))] \vee Loves(G(x),x)$$

- Distribute V over A

$$[Animal(F(x)) \vee Loves(G(x),x)] \wedge [\neg Loves(x,F(x)) \vee Loves(G(x),x)].$$

This is the CNF form of given sentence.

The resolution inference rule:

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Propositional literals are complementary if one is the negation of the other; first-order literals are complementary if one ***unifies with*** the negation of the other. Thus we have

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n)}$$

Where UNIFY (l_i, m_j) == θ.

For example, we can resolve the two clauses

$$[Animal(F(x)) \vee Loves(G(x), x)] \quad \text{and} \quad [\neg Loves(u, v) \vee \neg Kills(u, v)]$$

By eliminating the complementary literals *Loves (G(x), x)* and ¬*Loves (u, v),* with unifier θ = *{u/G(x), v/x),* to produce the resolvent clause

$$[Animal(F(x)) \vee \neg Kills(G(x), x)] .$$

Example proofs:

Resolution proves that KB /= *a* by proving *KB* A *la* unsatisfiable, i.e., by deriving the empty clause.

The sentences in CNF are

$$\neg American(x) \vee \neg Weapon(y) \vee \neg Sells(x, y, z) \vee \neg Hostile(z) \vee Criminal(x)$$
$$\neg Missile(x) \vee \neg Owns(Nono, x) \vee Sells(West, x, Nono) .$$
$$\neg Enemy(x, America) \vee Hostile(x) .$$
$$\neg Missile(x) \vee Weapon(x) .$$
$$Owns(Nono, M_1) . \qquad Missile(M_1) .$$
$$American(West) . \qquad Enemy(Nono, America) .$$

The resolution proof is shown in below figure;

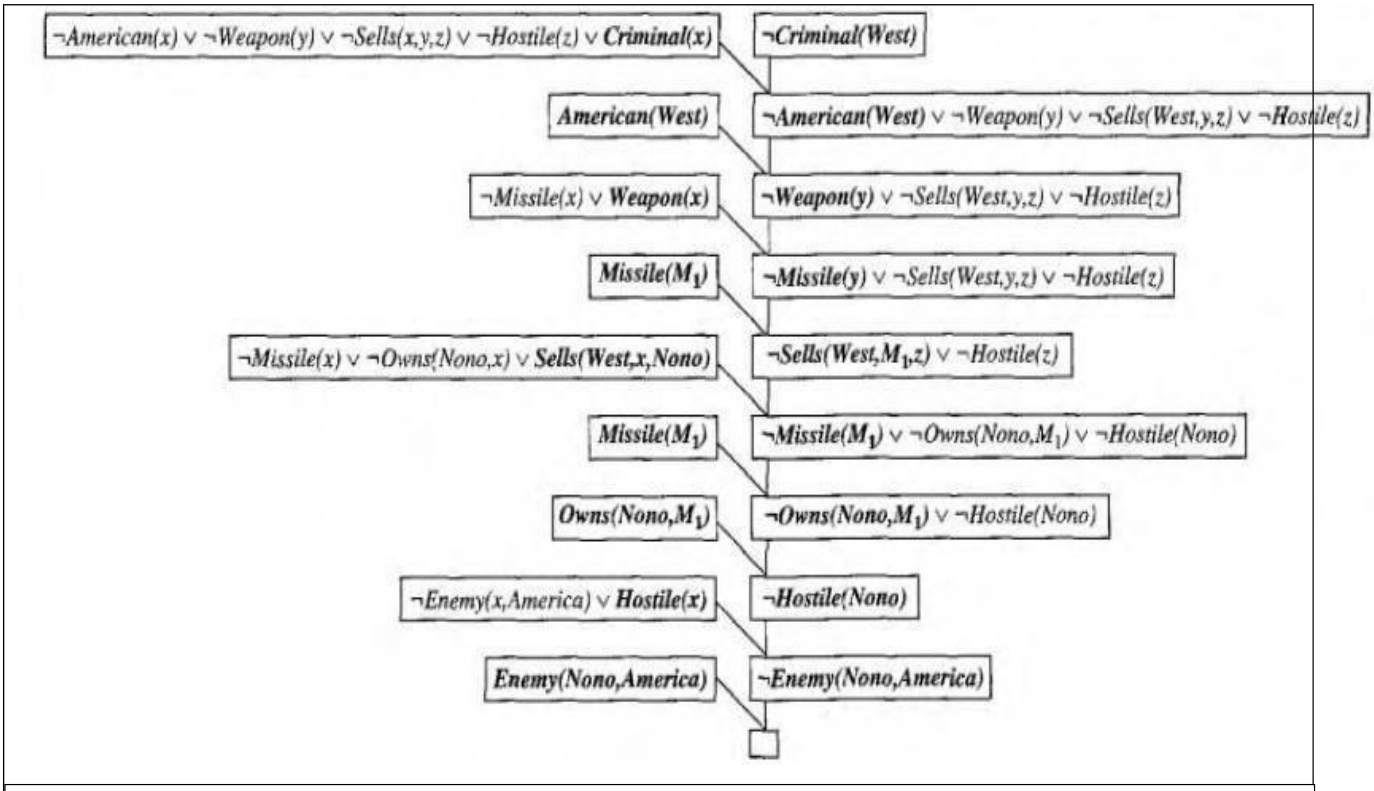


Figure 5.1 A resolution proof that West is a criminal.

Notice the structure: single "spine" beginning with the goal clause, resolving against clauses from the knowledge base until the empty clause is generated. Backward chaining is really just a special case of resolution with a particular control strategy to decide which resolution to perform next.

UNIT IV

DEFINITION OF CLASSIC PLANNING:

The problem-solving agent of Chapter 3 can find sequences of actions that result in a goal state. But it deals with atomic representations of states and thus needs good domain-specific heuristics to perform well. The hybrid propositional logical agent of Chapter 7 can find plans without domain-specific heuristics because it uses domain-independent heuristics based on the logical structure of the problem. But it relies on ground (variable-free) propositional inference, which means that it may be swamped when there are many actions and states. For example, in the wumpus world, the simple action of moving a step forward had to be repeated for all four agent orientations, T time steps, and n^2 current locations.

In response to this, planning researchers have settled on a **factored representation**— one in which a state of the world is represented by a collection of variables. We use a language called **PDDL**, the Planning Domain Definition Language, that allows us to express all $4T\ n^2$ actions with one action schema. There have been several versions of PDDL; we select a simple version and alter its syntax to be consistent with the rest of the book.[1] We now show how PDDL describes the four things we need to define a search problem: the initial state, the actions that are available in a state, the result of applying an action, and the goal test.

Each **state** is represented as a conjunction of fluents that are ground, functionless atoms. For example, Poor ∧ Unknown might represent the state of a hapless agent, and a state in a package delivery problem might be $At(Truck_1, Melbourne) \wedge At(Truck_2, Sydney)$. **Database semantics** is used: the closed-world assumption means that any fluents that are not mentioned are false, and the unique names assumption means that $Truck_1$ and $Truck_2$ are distinct. The following fluents are *not* allowed in a state: At(x, y) (because it is non-ground),

¬Poor (because it is a negation), and At(Father(Fred), Sydney) (because it uses a function symbol). The representation of states is carefully designed so that a state can be treated either as a conjunction of fluents, which can be manipulated by logical inference, or as a *set* of fluents, which can be manipulated with set operations. The **set semantics** is sometimes easier to deal with.

Actions are described by a set of action schemas that implicitly define the ACTIONS(s) and RESULT(s, a) functions needed to do a problem-solving search. We saw in Chapter 7 that any system for action description needs to solve the frame problem—to say what changes and what stays the same as the result of the action. Classical planning concentrates on problems where most actions leave most things unchanged. Think of a world consisting of a bunch of objects on a flat surface. The action of nudging an object causes that object to change its lo- cation by a vector Δ. A concise description of the action should mention only Δ; it shouldn't have to mention all the objects that stay in place. PDDL does that by specifying the result of an action in terms of what changes; everything that stays the same is left unmentioned.

A set of ground (variable-free) actions can be represented by a single **action schema**. The schema is a **lifted** representation—it lifts the level of reasoning from propositional logic to a restricted subset of first-order logic. For example, here is an action schema for flying a plane from one location to another:

Action(Fly(p, from, to),

PRECOND:At(p, from) ∧ Plane(p) ∧ Airport(from) ∧ Airport(to)

EFFECT:¬At(p, from) ∧ At(p, to))

The schema consists of the action name, a list of all the variables used in the schema, a **precondition** and an **effect**.

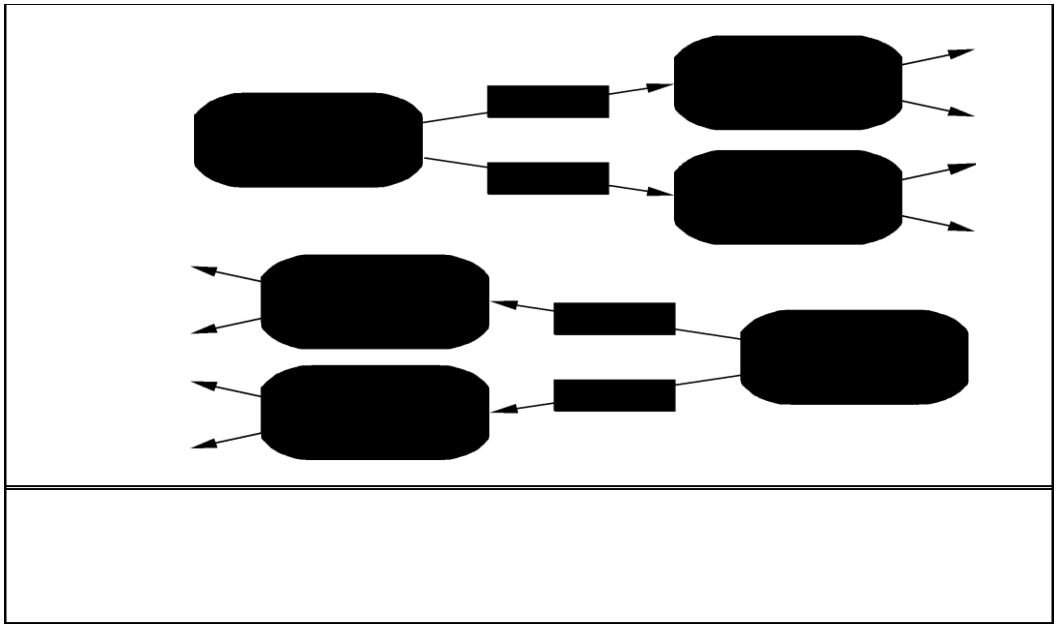
ALGORITHMS FOR PLANNING AS STATE-SPACE SEARCH

Now we turn our attention to planning algorithms. We saw how the description of a planning problem defines a search problem: we can search from the initial state through the space of states, looking for a goal. One of the nice advantages of the declarative representation of action schemas is that we can also search backward from the goal, looking for the initial state. Figure 10.5 compares forward and backward searches.

10.2.1 Forward (progression) state-space search

Now that we have shown how a planning problem maps into a search problem, we can solve planning problems with any of the heuristic search algorithms from Chapter 3 or a local search algorithm from Chapter 4 (provided we keep track of the actions used to reach the goal). From the earliest days of planning research (around 1961) until around 1998 it was assumed that forward state-space search was too inefficient to be practical. It is not hard to come up with reasons why.

First, forward search is prone to exploring irrelevant actions. Consider the noble task of buying a copy of *AI: A Modern Approach* from an online bookseller. Suppose there is an



1. **Figure 10.5** Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting at the set of states representing the goal and using the inverse of the actions to search backward for the initial state.

action schema Buy(isbn) with effect Own(isbn). ISBNs are 10 digits, so this action schema represents 10 billion ground actions. An uninformed forward-search algorithm would have to start enumerating these 10 billion actions to find one that leads to the goal.

Second, planning problems often have large state spaces. Consider an air cargo problem with 10 airports, where each airport has 5 planes and 20 pieces of cargo. The goal is to move all the cargo at airport A to airport B. There is a simple solution to the problem: load the 20 pieces of cargo into one of the planes at A, fly the plane to B, and unload the cargo. Finding the solution can be difficult because the average branching factor is huge: each of the 50 planes can fly to 9 other airports, and each of the 200 packages can be either unloaded (if it is loaded) or loaded into any plane at its airport (if it is unloaded). So in any state there is a minimum of 450 actions (when all the packages are at airports with no planes) and a maximum of 10,450 (when all packages and planes are at the same airport). On average, let's say

there are about 2000 possible actions per state, so the search graph up to the depth of the obvious solution has about 2000^{41} nodes.

Clearly, even this relatively small problem instance is hopeless without an accurate heuristic. Although many real-world applications of planning have relied on domain-specific heuristics, it turns out (as we see in Section 10.2.3) that strong domain-independent heuristics can be derived automatically; that is what makes forward search feasible.

RELEVANT-STATES

Backward (regression) relevant-states search

In regression search we start at the goal and apply the actions backward until we find a sequence of steps that reaches the initial state. It is called **relevant-states** search because we only consider actions that are relevant to the goal (or current state). As in belief-state search (Section 4.4), there is a *set* of relevant states to consider at each step, not just a single state.

We start with the goal, which is a conjunction of literals forming a description of a set of states—for example, the goal ¬Poor A Famous describes those states in which Poor is false, Famous is true, and any other fluent can have any value. If there are n ground fluents in a domain, then there are 2^n ground states (each fluent can be true or false), but 3^n descriptions of sets of goal states (each fluent can be positive, negative, or not mentioned).

In general, backward search works only when we know how to regress from a state description to the predecessor state description. For example, it is hard to search backwards for a solution to the n-queens problem because there is no easy way to describe the states that are one move away from the goal. Happily, the PDDL representation was designed to make it easy to regress actions—if a domain can be expressed in PDDL, then we can do regression search on it. Given a ground goal description g and a ground action a, the regression from g over a gives us a state description g defined by

.

That is, the effects that were added by the action need not have been true before, and also the preconditions must have held before, or else the action could not have been executed. Note that DEL(a) does not appear in the formula; that's because while we know the fluents in DEL(a) are no longer true after the action, we don't know whether or not they were true before.

Unit 5

RELEVANCE

To get the full advantage of backward search, we need to deal with partially uninstanti- ated actions and states, not just ground ones. For example, suppose the goal is to deliver a spe- cific piece of cargo to SFO: At(C2, SFO). That suggests the action Unload(C2, p , SFO):

Action(Unload(C2, p , SFO),
PRECOND:In(C2, p) A At(p , SFO) A Cargo(C2) A Plane(p) A Airport(SFO)
EFFECT:At(C2, SFO) A ¬In(C2, p) .

(Note that we have **standardized** variable names (changing p to p in this case) so that there will be no confusion between variable names if we happen to use the same action schema twice in a plan. The same approach was used in Chapter 9 for first-order logical inference.) This represents unloading the package from an *unspecified* plane at SFO; any plane will do, but we need not say which one now. We can take advantage of the power of first-order representations: a single description summarizes the possibility of using *any* of the planes by implicitly quantifying over p . The regressed state description is

g = In(C2, p) A At(p , SFO) A Cargo(C2) A Plane(p) A Airport(SFO) .

The final issue is deciding which actions are candidates to regress over. In the forward direc- tion we chose actions that were **applicable**—those actions that could be the next step in the plan. In backward search we want actions that are **relevant**—those actions that could be the *last* step in a plan leading up to the current goal state.

For an action to be relevant to a goal it obviously must contribute to the goal: at least one of the action's effects (either positive or negative) must unify with an element of the goal. What is less obvious is that the action must not have any effect (positive or negative) that negates an element of the goal. Now, if the goal is A A B A C and an action has the effect AABA¬C then there is a colloquial sense in which that action is very relevant to the goal—it gets us two-thirds of the way there. But it is not relevant in the technical sense defined here, because this action could not be the *final* step of a solution—we would always need at least one more step to achieve C.

Given the goal At(C2, SFO), several instantiations of Unload are relevant: we could chose any specific plane to unload from, or we could leave the plane unspecified by using the action Unload(C2, p , SFO). We can reduce the branching factor without ruling out any solutions by always using the action formed by substituting the most general unifier into the (standardized) action schema.

As another example, consider the goal Own(0136042597), given an initial state with

10 billion ISBNs, and the single action schema

A = Action(Buy(i), PRECOND:ISBN (i), EFFECT:Own(i)) .

As we mentioned before, forward search without a heuristic would have to start enumerating the 10 billion ground Buy actions. But with backward search, we would unify the goal Own(0136042597) with the (standardized) effect Own(i), yielding the substitution θ = {i /0136042597}. Then we would regress over the action Subst(θ, A) to yield the predecessor state description ISBN (0136042597). This is part of, and thus entailed by, the initial state, so we are done.

We can make this more formal. Assume a goal description g which contains a goal literal gi and an action schema A that is standardized to produce A . If A has an effect literal

e

=j SUBST(θ, A) and if there is no effect in a that is the negation of a literal in g, then a is a relevant action towards g.

w h e r e U n i f y (g i T, his section shows how a special data structure called a **planning graph** can be u e sed to give better heuristic estimates. These heuristics can be applied to any of the se j arch techniques we have seen so far. Alternatively, we can search for a solution o) ver the space formed by the planning graph, using an algorithm called GRAPHPLAN.

Backward search keeps the branching factor lower than forward search, for most prob- lem domains. However, the fact that backward search uses state sets rather than individual states makes it harder to come up with good heuristics. That is the main reason why the majority of current systems favor forward search.

PLANNING GRAPHS

All of the heuristics we have suggested can suffer from inaccuracies.

= θ a n d w h e r e d e f i n e d a

A planning problem asks if we can reach a goal state from the initial state. Suppose we are given a tree of all possible actions from the initial state to successor states, and their suc- cessors, and so on. If we indexed this tree appropriately, we could answer the planning ques- tion "can we reach state G from state S0" immediately, just by looking it up. Of course, the tree is of exponential size, so this approach is impractical. A planning graph is polynomial- size approximation to this tree that can be constructed quickly. The planning graph can't answer definitively whether G is reachable from S0, but it can *estimate* how many steps it takes to reach G. The estimate is always correct when it reports the goal is not reachable, and it never overestimates the number of steps, so it is an admissible heuristic.

A planning graph is a directed graph organized into **levels**: first a level S0 for the initial state, consisting of nodes representing each fluent that holds in S0; then a level A0 consisting of nodes for each ground action that might be applicable in S0; then alternating levels Si followed by Ai; until we reach a termination condition (to be discussed later).

Roughly speaking, Si contains all the literals that *could* hold at time i, depending on the actions executed at preceding time steps. If it is possible that either P or ¬P could hold, then both will be represented in Si. Also roughly speaking, Ai contains all the actions that *could* have their preconditions satisfied at time i. We say "roughly speaking" because the planning graph records only a restricted subset of the possible negative interactions among actions; therefore, a literal might show up at level Sj when actually it could not be true until a later level, if at all. (A literal will never show up too late.) Despite the possible error, the level j at which a literal first appears is a good estimate of how difficult it is to achieve the literal from the initial state.

))

Init(Have(Cake))

Action(Eat(Cake)

PRECOND: Have(Cake)

EFFECT: ¬ Have(Cake) A Eaten(Cake))

Action(Bake(Cake)

PRECOND: ¬ Have(Cake)

EFFECT: Have(Cake))

Figure 10.7 The "have cake and eat cake too" problem.

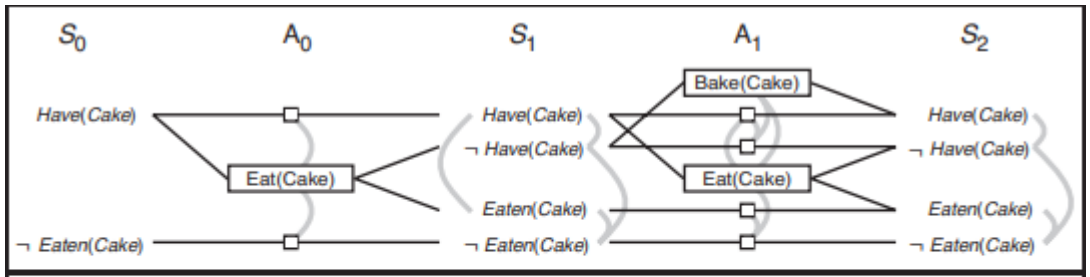


Figure 10.8 The planning graph for the "have cake and eat cake too" problem up to level S_2. Rectangles indicate actions (small squares indicate persistence actions), and straight lines indicate preconditions and effects. Mutex links are shown as curved gray lines. Not all mutex links are shown, because the graph would be too cluttered. In general, if two literals are mutex at S_i, then the persistence actions for those literals will be mutex at A_i and we need not draw that mutex link.

tion schemas. Despite the resulting increase in the size of the problem description, planning graphs have proved to be effective tools for solving hard planning problems.

Figure 10.7 shows a simple planning problem, and Figure 10.8 shows its planning graph. Each action at level A_i is connected to its preconditions at S_i and its effects at S_{i+1}. So a literal appears because an action caused it, but we also want to say that a literal can persist if no action negates it. This is represented by a **persistence action** (sometimes called a *no-op*). For every literal C, we add to the problem a persistence action with precondition C and effect C. Level A0 in Figure 10.8 shows one "real" action, Eat(Cake), along with two persistence actions drawn as small square boxes.

Level A_0 contains all the actions that *could* occur in state S_0, but just as important it records conflicts between actions that would prevent them from occurring together. The gray lines in Figure 10.8 indicate **mutual exclusion** (or **mutex**) links. For example, Eat(Cake) is

MUTEX mutually exclusive with the persistence of either Have(Cake) or ¬Eaten(Cake). We shall see shortly how mutex links are computed.

Level S_1 contains all the literals that could result from picking any subset of the actions in A_0, as well as mutex links (gray lines) indicating literals that could not appear together, regardless of the choice of actions. For example, Have(Cake) and Eaten(Cake) are m