NOORUL ISLAM CENTRE FOR HIGHER EDUCATION, KUMARACOIL

AI32D1

FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE

COURSE MATERIAL

AI32D1 FUNDAMENTALS OF ARTIFICIAL INTELLIGENCE 3 0 0 3

OBJECTIVES

- Understand the basic concepts of intelligent agents.
- Develop general-purpose problem solving agents, logical reasoning agents, and agents that reason under uncertainty.
- Employ AI techniques to solve some of today's real world problems.

UNIT I: AI AGENTS 9

Introduction to AI – Agents and Environments – Concept of rationality – Nature of environments – Structure of agents – Problem solving agents – Search algorithms –Uninformed search strategies.

UNIT II: PROBLEM SOLVING 9

Heuristic search strategies – heuristic functions - Local search and optimization problems – Local search in continuous space – Search with non-deterministic actions – Search in partially observable environments – Online search agents and unknown environments.

UNIT III: THEORY OF GAME PLAYING 9

Game theory – Optimal decisions in games – Alpha-beta search – Monte-Carlo tree search – Stochastic games – Partially observable games.

Constraint Satisfaction Problems – Constraint propagation – Backtracking search for CSP – Local search for CSP– Structure of CSP.

UNIT IV: KNOWLEDGE BASED LOGICAL AGENTS 9

Knowledge-based agents – Propositional logic – Propositional theorem proving – Propositional model checking– Agents based on propositional logic.

First-order logic – Syntax and semantics – Knowledge representation and engineering – Inferences in first - order logic – Forward chaining– Backward chaining – Resolution.

UNIT V: KNOWLEDGE REPRESENTATION 9

Ontological engineering – Objects and categories – Events – Mental objects and Modal logic – Reasoning systems for categories – Reasoning with default information.

Classical planning – Algorithms for classical planning – Heuristics for planning – Hierarchical planning – Non-deterministic domains – Time, Schedule, and Resources – Analysis.

TOTAL: 45 PERIODS

TEXT BOOKS

1. Stuart Russel and Peter Norvig, "Artificial Intelligence: A Modern Approach", Fourth Edition, Pearson Education,2020.

REFERENCES

1. Dan W.Patterson,"Introduction to AI and DS",PearsonEducation,2007.
2. Kevin Night,Elaine Rich, and Nair B.,"Artificial Intelligence",McGrawHill,2008.
3. Patrick H.Winston, "Artificial Intelligence", Third edition, PearsonEdition,2006.
4. DeepakKhemani, "Artificial Intelligence", Tata McGraw Hill Education, 2013.
5. Artificial Intelligence by Example: Develop machine intelligence from scratch using real artificial intelligence use cases – by Dennis Rothman, 2018.

UNIT I

INTELLIGENT AGENTS

Introduction to AI – Agents and Environments – concept of rationality – nature of environments – structure of agents Problem solving agents – search algorithms – uninformed search strategies

1.1 Introduction to AI

What is artificial intelligence?

Artificial Intelligence is the branch of computer science concerned with making computers behave like humans.

Artificial intelligence is defined as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.

The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

Systems that think like humans "The exciting new effort to make computers think … machines with minds, in the full and literal sense."(Haugeland,1985)	**Systems that think rationally** "The study of mental faculties through the use of computer models." (Charniak and McDermont,1985)
Systems that act like humans The art of creating machines that performs functions that require intelligence when performed by people."(Kurzweil,1990)	**Systems that act rationally** **"**Computational intelligence is the study of the design of intelligent agents."(Poole et al.,1998)

The four approaches in more detail are as follows:

Acting humanly : The Turing Test approach

- Test proposed by Alan Turing in 1950
- The computer is asked questions by a human interrogator.

The computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or not. Programming a computer to pass ,the computer need to possess the following capabilities :

- **Natural language processing** to enable it to communicate successfully in English.

- **Knowledge representation** to store what it knows or hears
- **Automated reasoning** to use the stored information to answer questions and to draw new conclusions.
- **Machine learning** to adapt to new circumstances and to detect and extrapolate patterns

To pass the complete Turing Test, the computer will need

- **Computer vision to** perceive the objects, and
- **Robotics** to manipulate objects and move about.

Thinking humanly: The cognitive modeling approach

We need to get inside actual working of the human mind:

(a) Through introspection – trying to capture our own thoughts as they go by;

(b) Through psychological experiments

Thinking rationally : The "laws of thought approach"

The Greek philosopher Aristotle was one of the first to attempt to codify **"right thinking"**,that is irrefuatable reasoning processes. His **syllogism** provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example,**"Socrates is a man;all men are mortal;therefore Socrates is mortal.".**

These laws of thought were supposed to govern the operation of the mind; their study initiated a field called **logic.**

Acting rationally : The rational agent approach

An **agent** is something that acts. Computer agents are not mere programs ,but they are expected to have the following attributes also :

(a) operating under autonomous control,

(b) perceiving their environment,

(c) persisting over a prolonged time period,

(d) adapting to change.

A **rational agent** is one that acts so as to achieve the best outcome.

1.2. AGENTS AND ENVIRONMENTS

An **agent** is anything that can be viewed as perceiving its **environment** through **sensors** and acting upon that environment through **actuators.** This simple idea is illustrated in Figure 1.2.

- 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.

$$f : \mathcal{P}^* \rightarrow \mathcal{A}$$

Agent program

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.

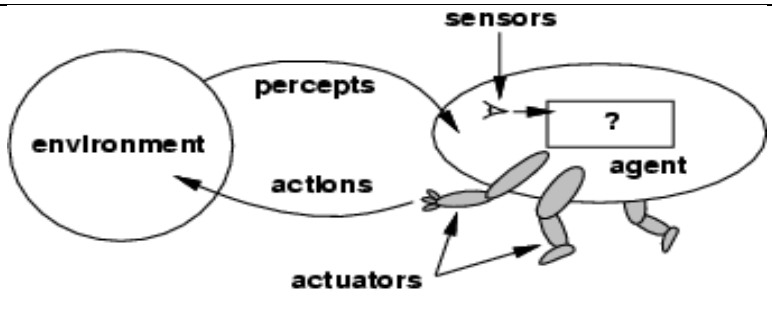


Figure 1.2 Agents interact with environments through sensors and actuators.

To illustrate these ideas, we use a very simple example-the vacuum-cleaner world. 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.

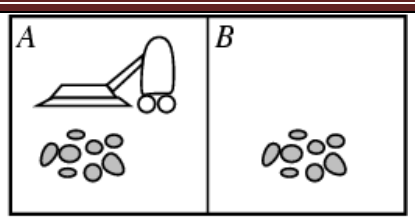


Figure 1.3 A vacuum-cleaner world with just two locations.

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
…	

Figure 1.4 Partial tabulation of a simple agent function

agent program

```
function REFLEX-VACUUM-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
```

1.3 Good Behavior - The Concept of Rationality

- The Concept of Rationality
- Performance measures
- Rationality
- Omniscience. learning. and autonomy

Rational Agent

A **rational agent** is one that does the right thing-conceptually speaking, every entry in the table for the agent function is filled out correctly.

Performance measures

A performance measure specifies the **criterion for success** of an agent's behavior.

When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.

This sequence of actions causes the environment to go through a sequence of states.

Rationality

Rationality depends on four things:

- The performance measure that defines the criterion of success.
- The agent's prior knowledge of the environment.
- The actions that the agent can perform.
- The agent's percept sequence to date.

Omniscience, learning, and autonomy

An **omniscient agent** knows the *actual* outcome of its actions and can act accordingly; but omniscience is impossible in reality. Doing actions in order to modify future percepts-sometimes called **information gathering**-is an important part of rationality.

A rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.

A rational agent should be **autonomous** it should learn what it can to compensate for partial or incorrect prior knowledge.

1.4 The Nature of Environments

- Specifying the task environment
- Properties of task environments

We must think about **task environments,** which are essentially the "**problems**" to which rational agents are the "**solutions**."

Specifying the task environment

The rationality of the simple vacuum-cleaner agent, needs specification of

- the performance measure
- the environment
- the agent's actuators and sensors.

PEAS

PEAS (Performance, Environment, Actuators, Sensors) description. In designing an agent, the first step must always be to specify the task environment as fully as possible.

Agent Type	Performance Measure	Environments	Actuators	Sensors
Taxi driver	Safe: fast, legal, comfortable trip, maximize profits	Roads,other traffic,pedestrians, Customers	Steering,accelerator, brake, Signal,horn,display	Cameras,sonar, Speedometer,GPS, Odometer,engine sensors,keyboards, accelerometer

Figure 1.5 PEAS description of the task environment for an automated taxi.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, minimize costs, lawsuits	Patient, hospital, staff	Display questions, tests, diagnoses, treatments, referrals	Keyboard entry of symptoms, findings, patient's answers
Satellite image analysis system	Correct image categorization	Downlink from orbiting satellite	Display categorization of scene	Color pixel arrays
Part-picking robot	Percentage of parts in correct bins	Conveyor belt with parts; bins	Jointed arm and hand	Camera, joint angle sensors
Refinery controller	Maximize purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Maximize student's score on test	Set of students, testing agency	Display exercises, suggestions, corrections	Keyboard entry

Figure 1.6 Examples of agent types and their PEAS descriptions.

Properties of task environments

- Fully observable vs. partially observable
- Deterministic vs. stochastic
- Episodic vs. sequential

- Static vs. dynamic
- Discrete vs. continuous
- Single agent vs. multiagent

Fully observable vs. **partially observable.**

If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable. A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;

An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simplly missing from the sensor data.

Deterministic vs. **stochastic.**

If the next state of the environment is completely determined by the current state and the action executed by the agent, then we say the environment is deterministic; otherwise, it is stochastic.

Episodic vs. **sequential**

In an **episodic task environment**, the agent's experience is divided into atomic episodes. Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.

For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;

In **sequential environments**, on the other hand, the current decision could affect all future decisions. Chess and taxi driving are sequential:

Discrete vs. **continuous.**

The discrete/continuous distinction can be applied to the *state* of the environment, to the way *time* is handled, and to the *percepts* and *actions* of the agent.

For example, a discrete-state environment such as a chess game has a finite number of distinct states. Chess also has a discrete set of percepts and actions. Taxi driving is a continuous-state and continuous-time problem: the speed and location of the taxi and of the other vehicles sweep through a range of continuous values and do so smoothly over time. Taxi-driving actions are also continuous (steering angles, etc.).

Single agent vs. **multiagent.**

An agent solving a crossword puzzle by itself is clearly in a single-agent environment, whereas an agent playing chess is in a two-agent environment.

Task Environment	Observable	Deterministic	Episodic	Static	Discrete	Agents
Crossword puzzle	Fully	Deterministic	Sequential	Static	Discrete	Single
Chess with a clock	Fully	Strategic	Sequential	Semi	Discrete	Multi
Poker	Partially	Stochastic	Sequential	Static	Discrete	Multi
Backgammon	Fully	Stochastic	Sequential	Static	Discrete	Multi
Taxi driving	Partially	Stochastic	Sequential	Dynamic	Continuous	Multi
Medical diagnosis	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Image-analysis	Fully	Deterministic	Episodic	Semi	Continuous	Single
Part-picking robot	Partially	Stochastic	Episodic	Dynamic	Continuous	Single
Refinery controller	Partially	Stochastic	Sequential	Dynamic	Continuous	Single
Interactive English tutor	Partially	Stochastic	Sequential	Dynamic	Discrete	Multi

Figure 1.7 Examples of task environments and their characteristics.

1.5 The Structure of Agents

- Agent programs
- Simple reflex agents
- Model-based reflex agents. .
- Goal-based agents
- Utility-based agents
- Learning agents

Agent programs

The agent programs take the current percept as input from the sensors and return an action to the actuators.

Function TABLE-DRIVEN_AGENT(*percept*) **returns** an action

static: *percepts*, a sequence initially empty

table, a table of actions, indexed by percept sequence

append *percept* to the end of *percepts*

action ← LOOKUP(*percepts*, *table*)

return *action*

The TABLE-DRIVEN-AGENT program is invoked for each new percept and returns **an** action each time.

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment

- **Problems**
  - Too big to generate and to store (Chess has about 10^120 states, for example)
  - No knowledge of non-perceptual parts of the current state
  - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
  - Looping: Can't make actions conditional
- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

Some Agent Types

- **Table-driven agents**
  - use a percept sequence/action table in memory to find the next action. They are implemented by a (large) **lookup table**.
- **Simple reflex agents**
  - are based on **condition-action rules**, implemented with an appropriate production system. They are stateless devices which do not have memory of past world states.
- **Agents with memory**
  - have **internal state**, which is used to keep track of past states of the world.
- **Agents with goals**
  - are agents that, in addition to state information, have **goal information** that describes desirable situations. Agents of this kind take future events into consideration.
- **Utility-based agents**
  - base their decisions on **classic axiomatic utility theory** in order to act rationally.

Simple Reflex Agent

The simplest kind of agent is the **simple reflex agent.** These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history. For example, the vacuum agent whose agent function is tabulated in Figure1.8 is a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.

- Select action on the basis of *only the current* percept.

  E.g. the vacuum-agent
- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules* If dirty then suck

Characteristics

- Only works if the environment is fully observable.
- Lacking history, easily get stuck in infinite loops
- One solution is to randomize actions

A Simple Reflex Agent: Schema

function SIMPLE-REFLEX-AGENT(percept) returns an action static: rules, a set of condition-action rules state ← INTERPRET-INPUT(percept) rule ← RULE-MATCH(state, rule) action ← RULE-ACTION[rule] return action
A simple reflex agent. It acts according to a rule whose condition matches the current state, as defined by the percept.

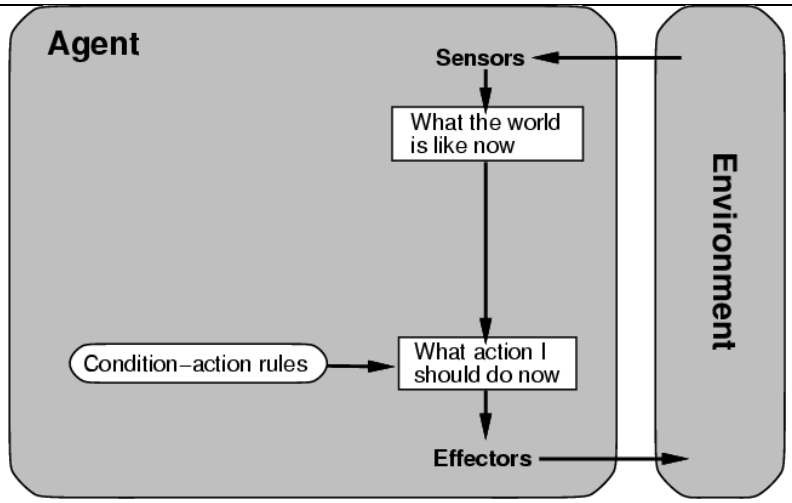



Figure1.8 Schematic diagram of a simple reflex agent.

function REFLEX-VACUUM-AGENT ([*location, status*]) return an action if *status == Dirty* then return *Suck* else if *location == A* then return *Right* else if *location == B* then return *Left*
The agent program for a simple reflex agent in the two-state vacuum environment.

Model-based reflex agents

The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now.*

Updating this internal state information requires two kinds of knowledge to be encoded in the agent program.

First, we need some information about how the world evolves independently of the agent-for example, that an overtaking car generally will be closer behind than it was a moment ago.

Second, we need some information about how the agent's own actions affect the world-for example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.

This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world. An agent that uses such a MODEL-BASED model is called a **model-based agent.**

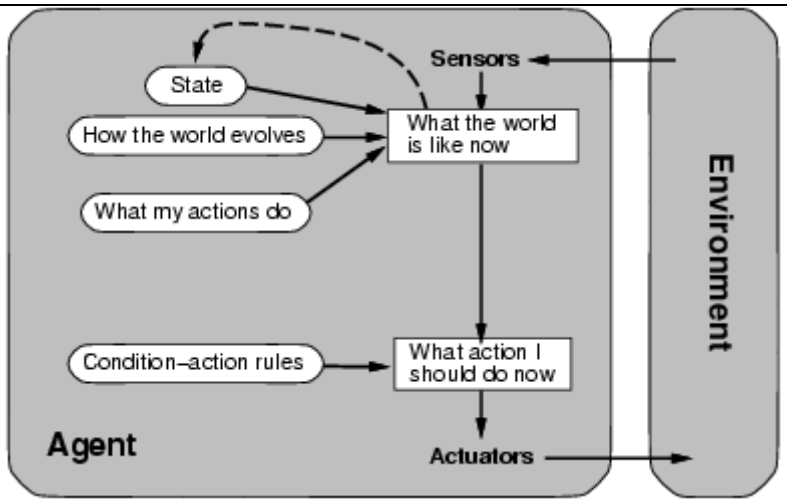


Figure1.9 A model based reflex agent

function REFLEX-AGENT-WITH-STATE(*percept*) **returns** an action

static: *rules*, a set of condition-action rules

state, a description of the current world state

action, the most recent action.

state ← UPDATE-STATE(*state*, *action*, *percept*)

rule ← RULE-MATCH(*state*, *rule*)

action ← RULE-ACTION[*rule*]

return *action*

Model based reflex agent.

Goal-based agents

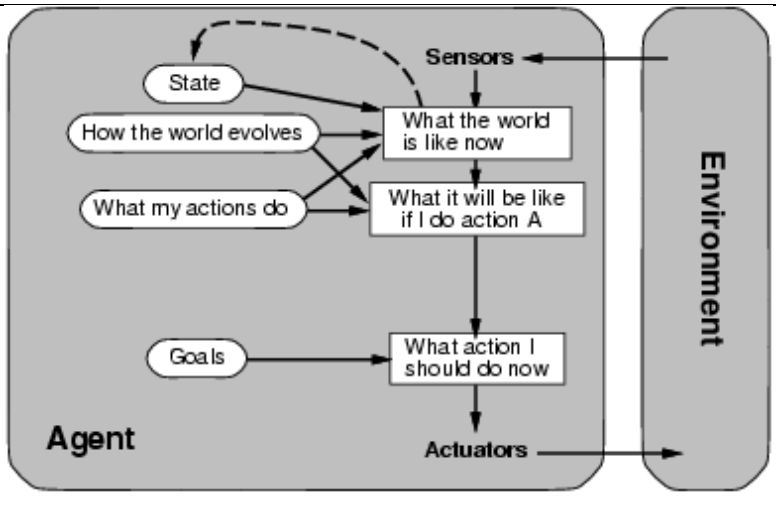


Figure1.10 A goal based agent

Knowing about the current state of the environment is not always enough to decide what to do. For example, at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to.

The agent needs some sort of **goal** information that describes situations that are desirable-for example, being at the passenger's destination. The agent program can combine this with information about the results of possible actions in order to choose actions that achieve the goal.

Utility-based agents

Goals alone are not really enough to generate high-quality behavior in most environments. For example, there are many action sequences that will get the taxi to its destination but some are quicker, safer, more reliable, or cheaper than others.

Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.

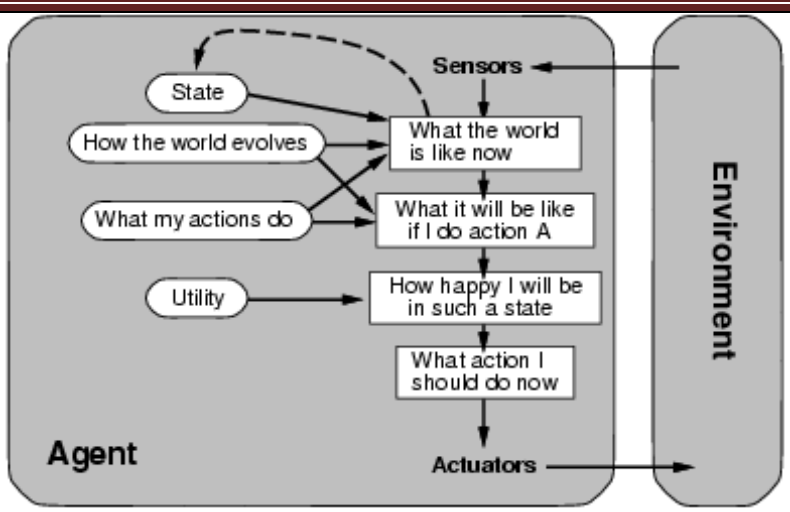


Figure1.11 A model-based, utility-based agent.

- Certain goals can be reached in different ways.
  - Some are better, have a higher utility.
- Utility function maps a (sequence of) state(s) onto a real number.
- Improves on goals:
  - Selecting between conflicting goals
  - Select appropriately between several goals based on likelihood of success.

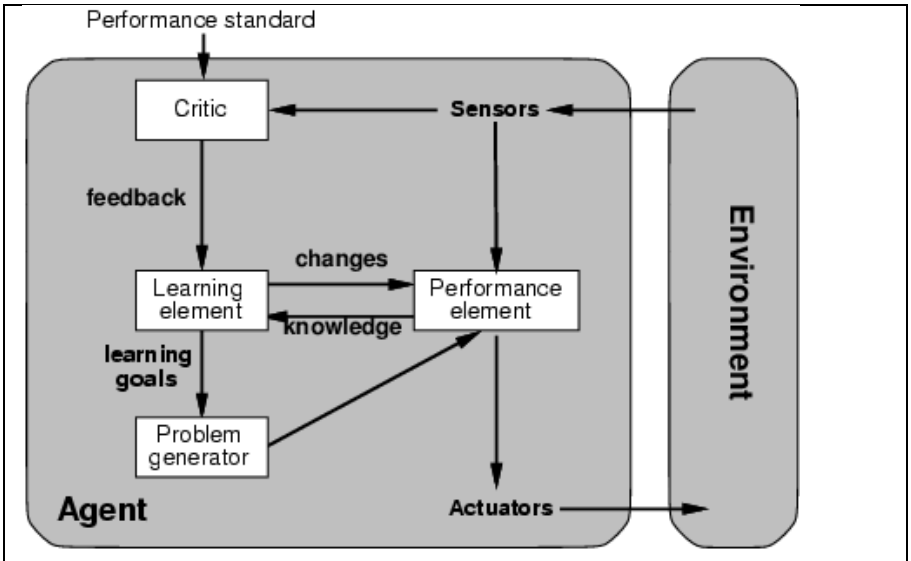


Figure 1.12 A general model of learning agents.

- All agents can improve their performance through **learning.**

A learning agent can be divided into four conceptual components, as shown in Figure 1.15 The most important distinction is between the **learning element**, which is responsible for making

improvements, and the **performance element**, which is responsible for selecting external actions. The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.

The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.

The last component of the learning agent is the **problem generator**. It is responsible for suggesting actions that will lead to new and **informative experiences**. But if the agent is willing to explore a little, it might discover much better actions for the long run. The problem generator's job is to suggest these **exploratory actions**. This is what scientists do when they carry out experiments.

1.6 Problem solving agents

An important aspect of intelligence is ***goal-based*** problem solving. The **solution** of many **problems** can be described by finding a **sequence of actions** that lead to a desirable **goal.** Each action changes the ***state*** and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.

A well-defined problem can be described by:

- **Initial state**
- **Operator or successor function** - for any state x returns s(x), the set of states reachable from x with one action
- **State space** - all states reachable from initial by any sequence of actions
- **Path** - sequence through state space
- **Path cost** - function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
- **Goal test** - test to determine if at goal state

What is Search?

Search is the systematic examination of **states** to find path from the **start/root state** to the **goal state.** The set of possible states, together with *operators* defining their connectivity constitute the *search space*. The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

Problem-solving agents

A Problem solving agent is a **goal-based** agent. It decide what to do by finding sequence of actions that lead to desirable states. The agent can adopt a goal and aim at satisfying it.

To illustrate the agent's behavior, let us take an example where our agent is in the city of Arad, which is in Romania. The agent has to adopt a **goal** of getting to Bucharest. **Goal formulation**, based on the current situation and the agent's performance measure, is the first step in problem solving.

The agent's task is to find out which sequence of actions will get to a goal state.

Problem formulation is the process of deciding what actions and states to consider given a goal.

Example: Route finding problem
Referring to figure On holiday in Romania : currently in Arad. Flight leaves tomorrow from Bucharest **Formulate goal**: be in Bucharest **Formulate problem**: **states**: various cities **actions**: drive between cities **Find solution**: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
Problem formulation
A **problem** is defined by four items: **initial state** e.g., "at Arad" **successor function** S(x) = set of action-state pairs e.g., S(Arad) = {[Arad -> Zerind;Zerind],….} **goal test**, can be explicit, e.g., x = at Bucharest" implicit, e.g., NoDirt(x) **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.
Goal formulation and problem formulation

1.7 Search

An agent with several immediate options of unknown value can decide what to do by examining different possible sequences of actions that leads to the states of known value, and then choosing the best sequence. The process of looking for sequences actions from the current state to reach the goal state is called **search.**

The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.** Once a solution is found, the **execution phase** consists of carrying out the recommended action..

Figure 1.18 shows a simple "formulate, search, execute" design for the agent. Once solution has been executed, the agent will formulate a new goal.

```
function SIMPLE-PROBLEM-SOLVING-AGENT( percept) returns an action
inputs : percept, a percept
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation
state UPDATE-STATE(state, percept)
if seq is empty then do
      goal ← FORMULATE-GOAL(state)
      problem ← FORMULATE-PROBLEM(state, goal)
      seq ← SEARCH( problem)
action ← FIRST(seq);
 seq ←REST(seq)
return action
```

A Simple problem solving agent. It first formulates a **goal** and a **problem**,searches for a sequence of actions that would solve a problem,and executes the actions one at a time.

- The agent design assumes the Environment is
  - **Static** : The entire process carried out without paying attention to changes that might be occurring in the environment.
  - **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
  - **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken

- **Deterministic :** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

Well-defined problems and solutions

A **problem** can be formally defined by **four components**:

- The **initial state** that the agent starts in . The initial state for our agent of example problem is described by *In(Arad)*
- A **Successor Function** returns the possible **actions** available to the agent. Given a state x,SUCCESSOR-FN(x) returns a set of {action,successor} ordered pairs where each action is one of the legal actions in state x,and each successor is a state that can be reached from x by applying the action.

For example,from the state In(Arad),the successor function for the Romania problem would return

{ [Go(Sibiu),In(Sibiu)],[Go(Timisoara),In(Timisoara)],[Go(Zerind),In(Zerind)] }

- **State Space** : The set of all states reachable from the initial state. The state space forms a graph in which the nodes are states and the arcs between nodes are actions.
- A **path** in the state space is a sequence of states connected by a sequence of actions.
- Thr **goal test** determines whether the given state is a goal state.
- A **path cost** function assigns numeric cost to each action. For the Romania problem the cost of path might be its length in kilometers.
- The **step cost** of taking action a to go from state x to state y is denoted by c(x,a,y). The step cost for Romania are shown in figure 1.18. It is assumed that the step costs are non negative.
- A **solution** to the problem is a path from the initial state to a goal state.
- An **optimal solution** has the lowest path cost among all solutions.

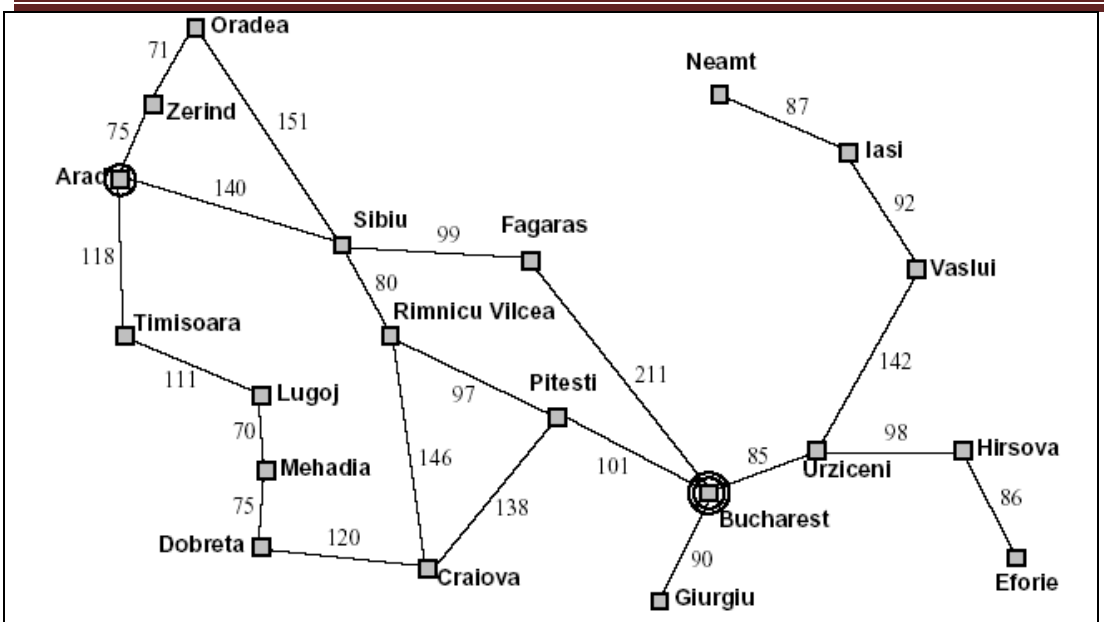


Figure A simplified Road Map of part of Romania

EXAMPLE PROBLEMS

The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below. They are distinguished as toy or real-world problems

A **toy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.

A **real world problem** is one whose solutions people actually care about.

TOY PROBLEMS

Vacuum World Example

- **States**: The agent is in one of two locations.,each of which might or might not contain dirt. Thus there are 2 x 2^2 = 8 possible world states.
- **Initial state**: Any state can be designated as initial state.
- **Successor function** : This generates the legal states that results from trying the three actions (left, right, suck). The complete state space is shown in figure 2.3
- **Goal Test** : This tests whether all the squares are clean.
- **Path test** : Each step costs one ,so that the the path cost is the number of steps in the path.

Vacuum World State Space

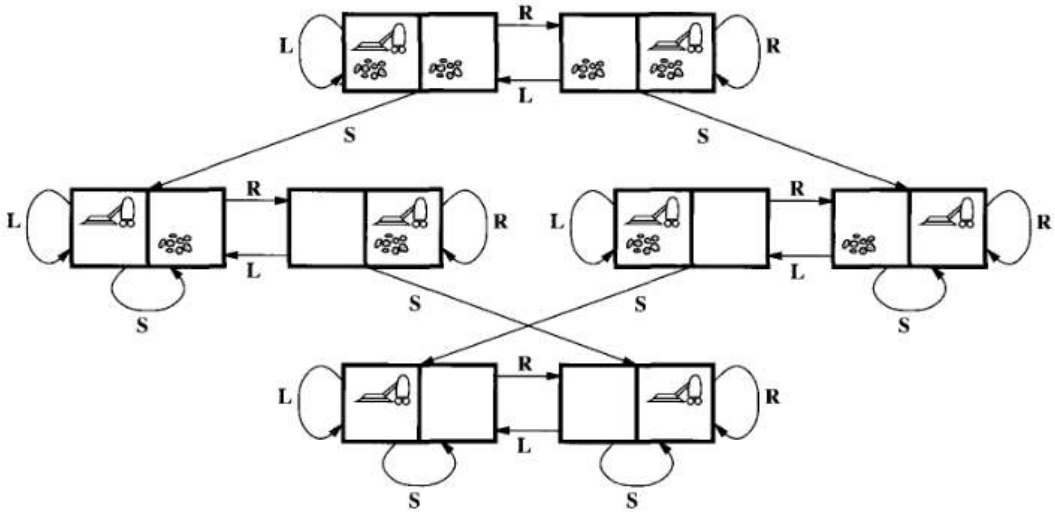


Figure 1.20 The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

The 8-puzzle

An 8-puzzle consists of a 3x3 board with eight numbered tiles and a blank space. A tile adjacent to the balank space can slide into the space. The object is to reach the goal state ,as shown in figure 2.4

Example: The 8-puzzle

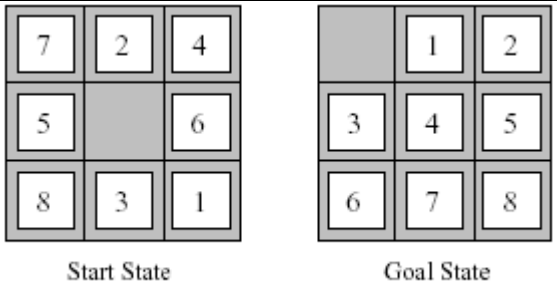


Figure 1.21 A typical instance of 8-puzzle.

The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions(blank moves Left,Right,Up or down).

- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost** : Each step costs 1,so the path cost is the number of steps in the path.

The 8-puzzle belongs to the family **of sliding-block puzzles**,which are often used as test problems for new search algorithms in AI. This general class is known as NP-complete.

The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.

The **15 puzzle** (4 x 4 board) has around 1.3 trillion states,an the random instances can be solved optimally in few milli seconds by the best search algorithms.

The **24-puzzle** (on a 5 x 5 board) has around 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-queens problem

The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other. Figure 2.5 shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.

An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state. A **complete-state formulation** starts with all 8 queens on the board and move them around.

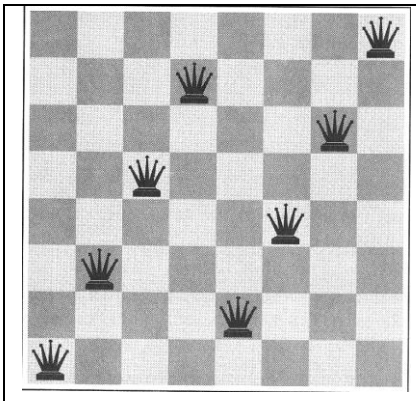


Figure 1.22 8-queens problem

The first incremental formulation one might try is the following :

- **States** : Any arrangement of 0 to 8 queens on board is a state.
- **Initial state** : No queen on the board.
- **Successor function** : Add a queen to any empty square.

- **Goal Test** : 8 queens are on the board,none attacked.

In this formulation,we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.

A better formulation would prohibit placing a queen in any square that is already attacked.

- **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
- **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.

SEARCHING FOR SOLUTIONS

SEARCH TREE

A **search tree** is generated by the **initial state** and the **successor function** that together define the **state space**. In general, we may have a *search graph* rather than a *search tree*,when the same state can be reached from multiple paths.

Figure shows some of the expansions in the search tree for finding a route from Arad to Bucharest.

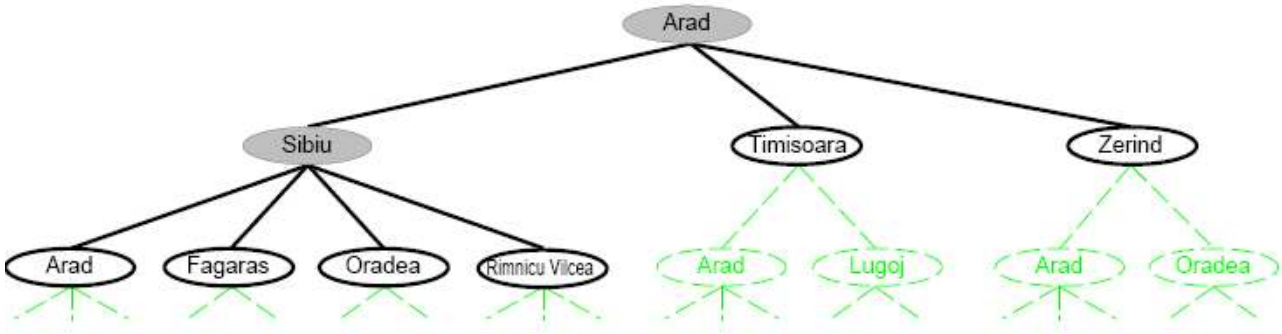


Figure Partial search trees for finding a route from Arad to Bucharest. Nodes that have been expanded are shaded.; nodes that have been generated but not yet expanded are outlined in bold;nodes that have not yet been generated are shown in faint dashed line

The root of the search tree is a **search node** corresponding to the initial **state**,In(Arad). The first step is to test whether this is a **goal state**. The current state is expanded by applying the successor function to the current state,thereby generating a new set of states.

In this case,we get three new states: In(Sibiu),In(Timisoara),and In(Zerind). Now we must choose which of these three possibilities to consider further. This is the essense of search- following up one option now and putting the others aside for latter,in case the first choice does not lead to a solution.

Search strategy . The general tree-search algorithm is described informally in Figure below

Tree Search

The choice of which state to expand is determined by the **search strategy**. There are an infinite number paths in this state space ,so the search tree has an infinite number of **nodes.**

A **node** is a data structure with five components :

- STATE : a state in the state space to which the node corresponds;
- PARENT-NODE : the node in the search tree that generated this node;
- ACTION : the action that was applied to the parent to generate the node;
- PATH-COST :the cost,denoted by g(n),of the path from initial state to the node,as indicated by the parent pointers; and
- DEPTH : the number of steps along the path from the initial state.

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

Figure An informal description of the general tree-search algorithm

It is important to remember the distinction between nodes and states. A node is a book keeping data structure used to represent the search tree. A state corresponds to configuration of the world.

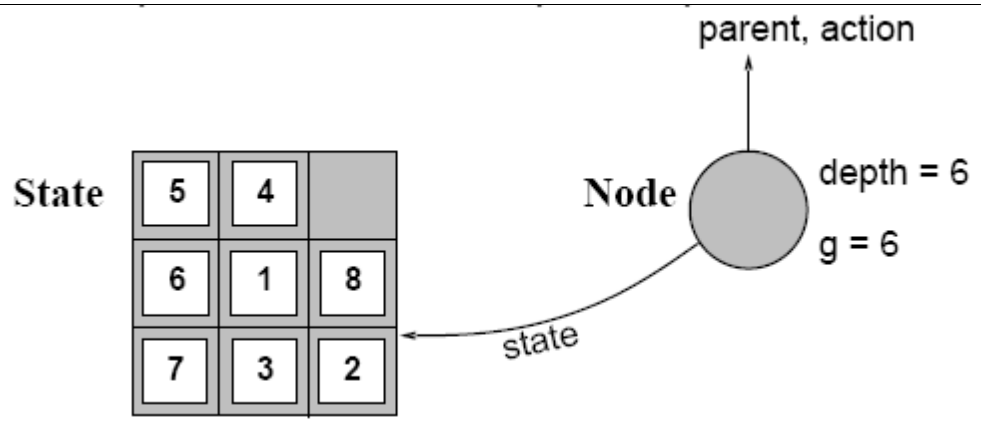


Figure Nodes are data structures from which the search tree is constructed. Each has a parent,a state, Arrows point from child to parent.

Fringe

Fringe is a collection of nodes that have been generated but not yet been expanded. Each element of the fringe is a leaf node,that is,a node with no successors in the tree. The fringe of each tree consists of those nodes with bold outlines.

The collection of these nodes is implemented as a **queue.**

The general tree search algorithm is shown in Figure below

The operations specified in Figure on a queue are as follows:

- **MAKE-QUEUE(element,…)** creates a queue with the given element(s).
- **EMPTY?(queue)** returns true only if there are no more elements in the queue.
- **FIRST(queue)** returns FIRST(queue) and removes it from the queue.
- **INSERT(element,queue)** inserts an element into the queue and returns the resulting queue.
- **INSERT-ALL(elements,queue)** inserts a set of elements into the queue and returns the resulting queue.

```
function TREE-SEARCH(problem, fringe) returns a solution, or failure

  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
     if EMPTY?(fringe) then return failure
     node ← REMOVE-FIRST(fringe)
     if GOAL-TEST[problem] applied to STATE[node] succeeds
        then return SOLUTION(node)
     fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function EXPAND(node, problem) returns a set of nodes

  successors ← the empty set
  for each ⟨action, result⟩ in SUCCESSOR-FN[problem](STATE[node]) do
     s ← a new NODE
     STATE[s] ← result
     PARENT-NODE[s] ← node
     ACTION[s] ← action
     PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
     DEPTH[s] ← DEPTH[node] + 1
     add s to successors
  return successors
```

Figure The general Tree search algorithm

MEASURING PROBLEM-SOLVING PERFORMANCE

The output of problem-solving algorithm is either failure or a solution.(Some algorithms might struck in an infinite loop and never return an output.

The algorithm's performance can be measured in four ways :

- **Completeness** : Is the algorithm guaranteed to find a solution when there is one?
- **Optimality** : Does the strategy find the optimal solution
- **Time complexity** : How long does it take to find a solution?
- **Space complexity** : How much memory is needed to perform the search?

1.8 UNINFORMED SEARCH STRATGES

Uninformed Search Strategies have no additional information about states beyond that provided in the **problem definition**.

Strategies that know whether one non goal state is "more promising" than another are called **Informed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth-first search

Breadth-first search is a simple strategy in which the root node is expanded first,then all successors of the root node are expanded next,then their successors,and so on. In general,all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.

Breath-first-search is implemented by calling TREE-SEARCH with an empty fringe that is a first-in-first-out(FIFO) queue,assuring that the nodes that are visited first will be expanded first.In otherwards,calling TREE-SEARCH(problem,FIFO-QUEUE()) results in breadth-first-search.The FIFO queue puts all newly generated successors at the end of the queue,which means that Shallow nodes are expanded before deeper nodes.

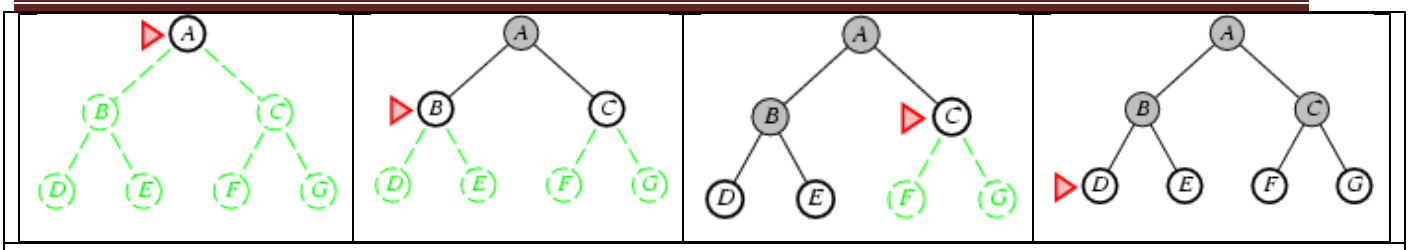


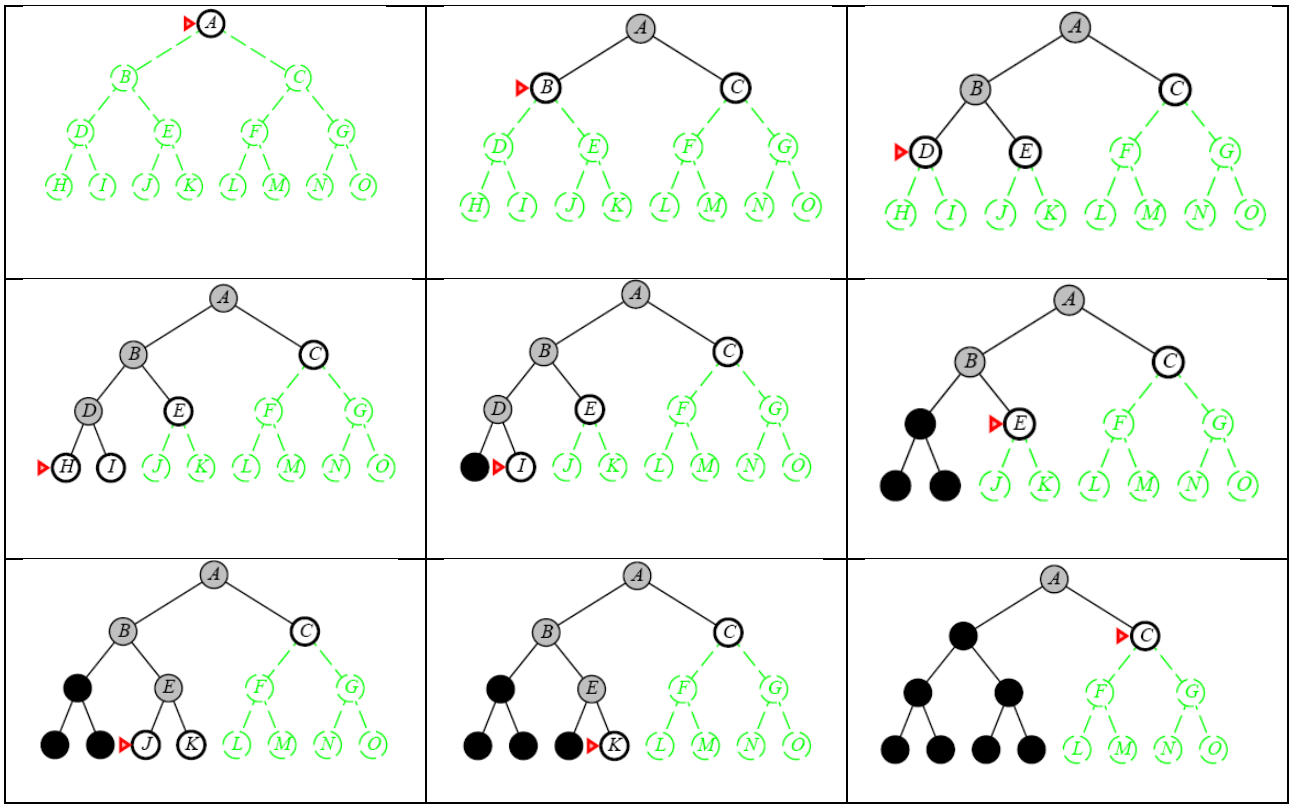
Figure Breadth-first search on a simple binary tree. At each stage ,the node to be expanded next is indicated by a marker.

UNIFORM-COST SEARCH

Instead of expanding the shallowest node,**uniform-cost search** expands the node n with the lowest path cost. uniform-cost search does not care about the number of steps a path has,but only about their total cost.

DEPTH-FIRST-SEARCH

Depth-first-search always expands the deepest node in the current fringe of the search tree. The progress of the search is illustrated in figure below. The search proceeds immediately to the deepest level of the search tree,where the nodes have no successors. As those nodes are expanded,they are dropped from the fringe,so then the search "backs up" to the next shallowest node that still has unexplored successors.



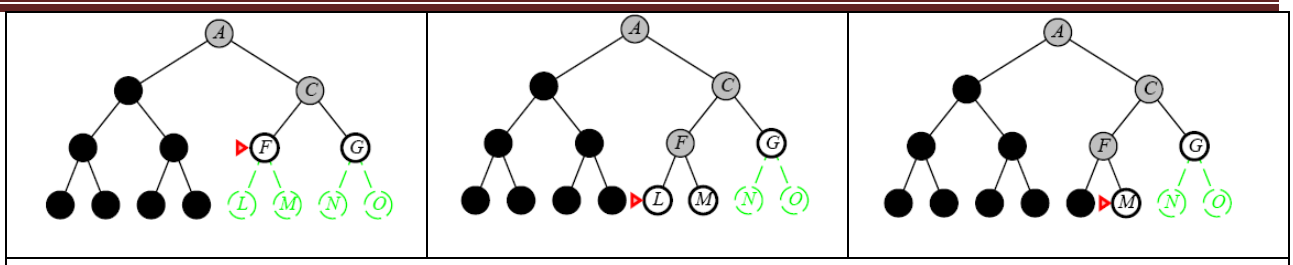


Figure Depth-first-search on a binary tree. Nodes that have been expanded and have no descendants in the fringe can be removed from the memory;these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

This strategy can be implemented by TREE-SEARCH with a last-in-first-out (LIFO) queue,also known as a stack.

It needs to store only a single path from the root to a leaf node,along with the remaining unexpanded sibling nodes for each node on the path. Once the node has been expanded,it can be removed from the memory,as soon as its descendants have been fully explored

For a state space with a branching factor b and maximum depth m, depth-first-search requires storage of only bm + 1 nodes.

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree. For example ,depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

A variant of depth-first search called backtracking search uses less memory and only one successor is generated at a time rather than all successors.; Only O(m) memory is needed rather than O(bm)

DEPTH-LIMITED-SEARCH

The problem of unbounded trees can be alleviated by supplying depth-first-search with a pre-determined depth limit l. That is, nodes at depth l are treated as if they have no successors. This approach is called **depth-limited-search**. The depth limit soves the infinite path problem.

Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo

Depth-limited-search can be implemented as a simple modification to the general tree-search algorithm or to the recursive depth-first-search algorithm. The pseudocode for recursive depth-limited-search is shown in Figure 1.32.

It can be noted that the above algorithm can terminate with two kinds of failure : the standard *failure* value indicates no solution; the *cutoff* value indicates no solution within the depth limit.

Depth-limited search = depth-first search with depth limit l,

returns cut off if any path is cut off by depth limit

```
function Depth-Limited-Search( problem, limit) returns a solution/fail/cutoff
return Recursive-DLS(Make-Node(Initial-State[problem]), problem, limit)
function Recursive-DLS(node, problem, limit) returns solution/fail/cutoff
cutoff-occurred? ← false
if Goal-Test(problem,State[node]) then return Solution(node)
else if Depth[node] = limit then return cutoff
else for each successor in Expand(node, problem) do
result  Recursive-DLS(successor, problem, limit)
if result = cutoff then cutoff_occurred? ← true
else if result not = failure then return result
if cutoff_occurred? then return cutoff else return failure
```

Figure Recursive implementation of Depth-limited-search:

ITERATIVE DEEPENING DEPTH-FIRST SEARCH

Iterative deepening search (or iterative-deepening-depth-first-search) is a general strategy often used in combination with depth-first-search,that finds the better depth limit. It does this by gradually increasing the limit – first 0,then 1,then 2, and so on – until a goal is found. This will occur when the depth limit reaches d,the depth of the shallowest goal node. The algorithm is shown in Figure 2.14.

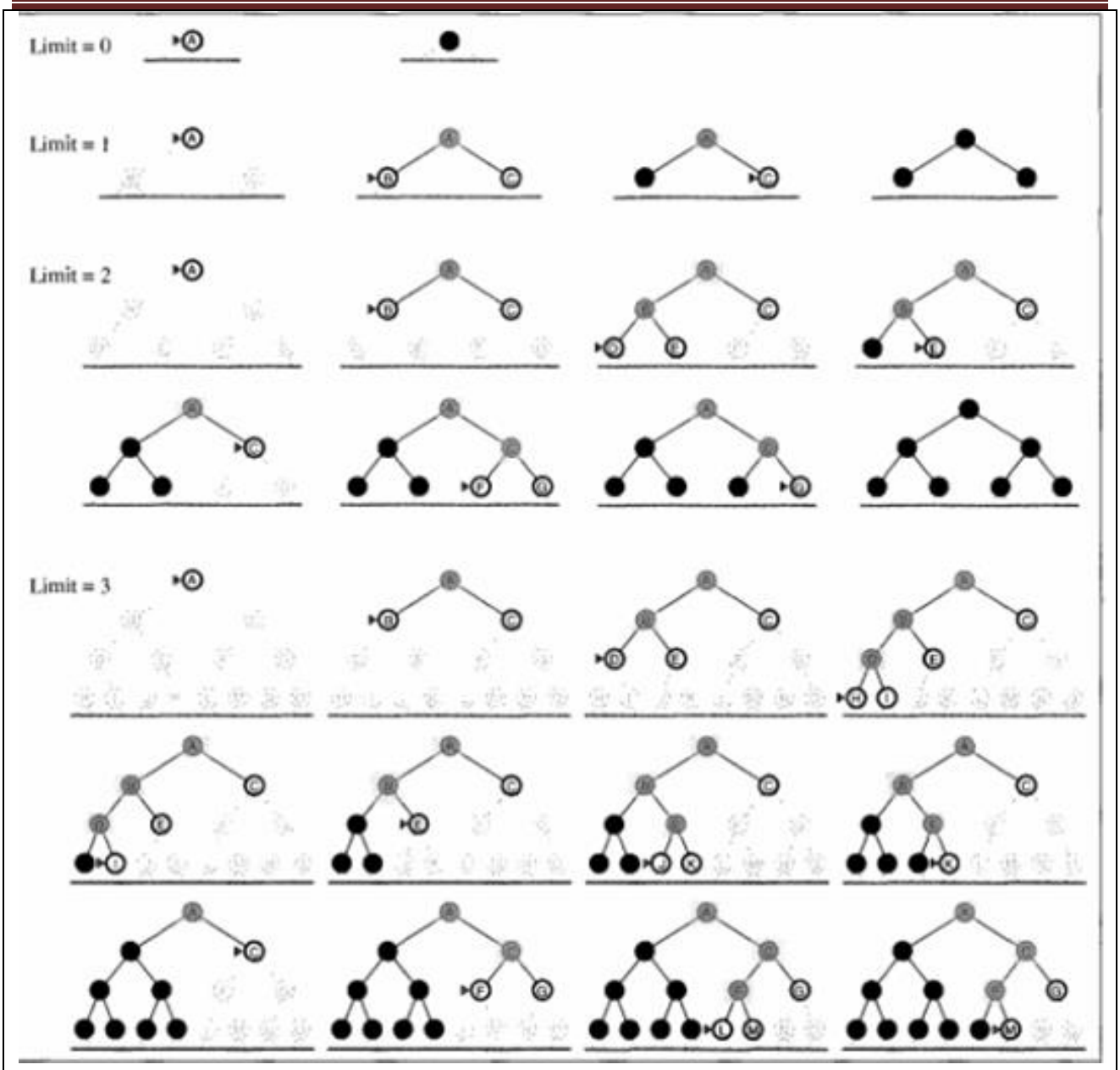


Figure Four iterations of iterative deepening search on a binary tree

Iterative deepening combines the benefits of depth-first and breadth-first-search

Like depth-first-search,its memory requirements are modest;O(bd) to be precise.

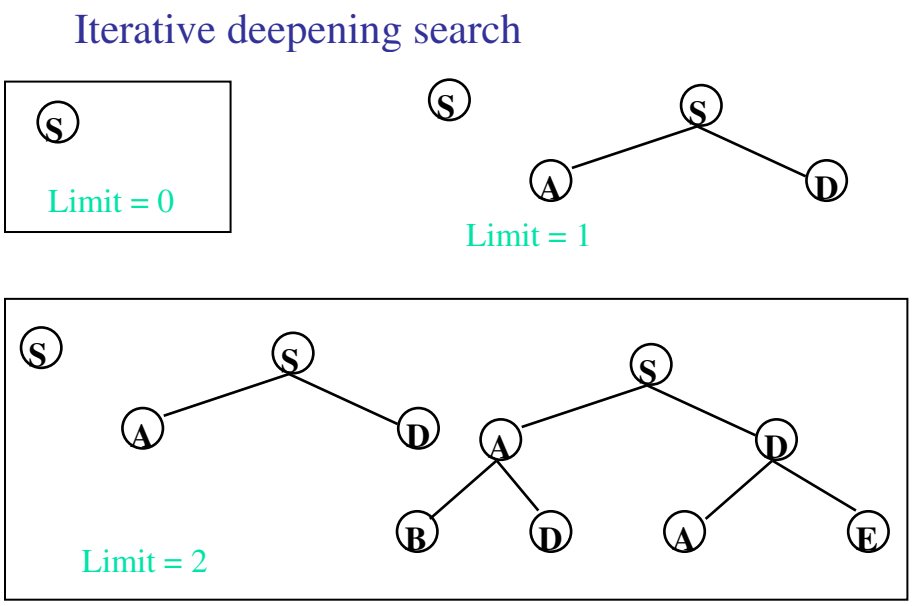
Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.

Figure shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
    end
```

Figure The **iterative deepening search algorithm** ,which repeatedly applies depth-limited-search with increasing limits. It terminates when a solution is found or if the depth limited search resturns *failure*,meaning that no solution exists.

Iterative search is not as wasteful as it might seem



Figure

Iterative search is not as wasteful as it might seem

1.3.4.6 Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches – one forward from the initial state and the other backward from the goal, stopping when the two searches meet in the middle

The motivation is that $b^{d/2} + b^{d/2}$ much less than ,or in the figure ,the area of the two small circles is less than the area of one big circle centered on the start and reaching to the goal.

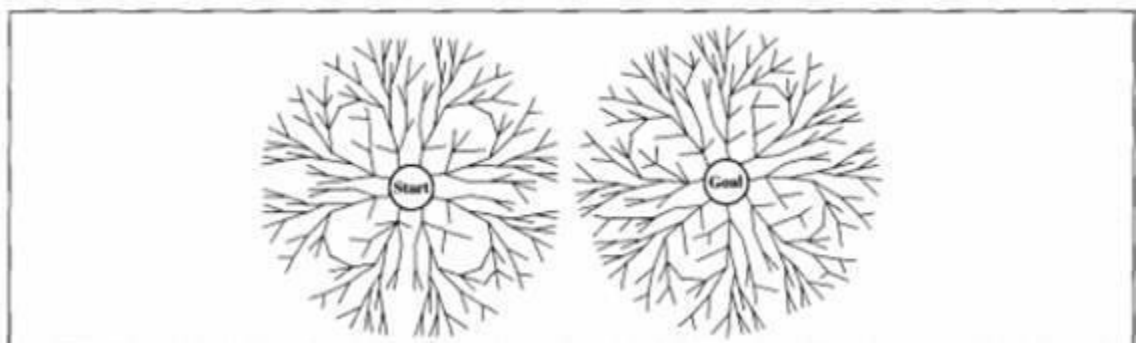


Figure A schematic view of a bidirectional search that is about to succeed,when a Branch from the Start node meets a Branch from the goal node.

Comparing Uninformed Search Strategies

Figure 1.38 compares search strategies in terms of the four evaluation criteria .

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^{d+1})$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 1.38 Evaluation of search strategies

b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit.

UNIT II

PROBLEM SOLVING

Heuristic search strategies – heuristic functions- Local search and optimization problems – local search in continuous space – search with non-deterministic actions – search in partially observable environments – online search agents and unknown environments

2.1 HEURISTIC SEARCH STRATEGIES

Informed(Heuristic) Search Strategies

Informed search strategy is one that uses problem-specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than uninformed strategy.

Best-first search

Best-first search is an instance of general TREE-SEARCH or GRAPH-SEARCH algorithm in which a node is selected for expansion based on an **evaluation function** f(n).

The node with lowest evaluation is selected for expansion,because the evaluation measures the distance to the goal.

This can be implemented using a priority-queue,a data structure that will maintain the fringe in ascending order of f-values.

Heuristic functions

A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.

The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

h(n) = extimated cost of the **cheapest path** from node n to a **goal node**.

For example,in Romania,one might estimate the cost of the cheapest path from Arad to Bucharest via a **straight-line distance** from Arad to Bucharest(Figure 2.1).

Heuristic function are the most common form in which additional knowledge is imparted to the search algorithm.

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 a solution quickly.

It evaluates the nodes by using 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 2.1. 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.

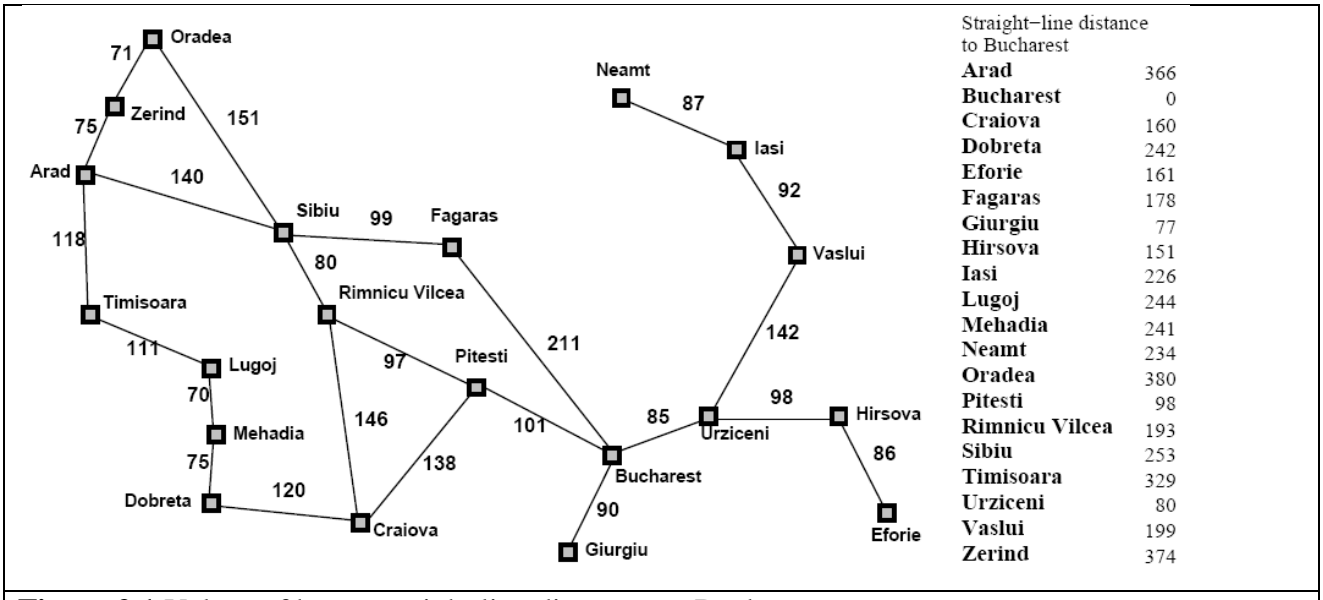


Figure 2.1 Values of h_{SLD} - straight line distances to Bucharest

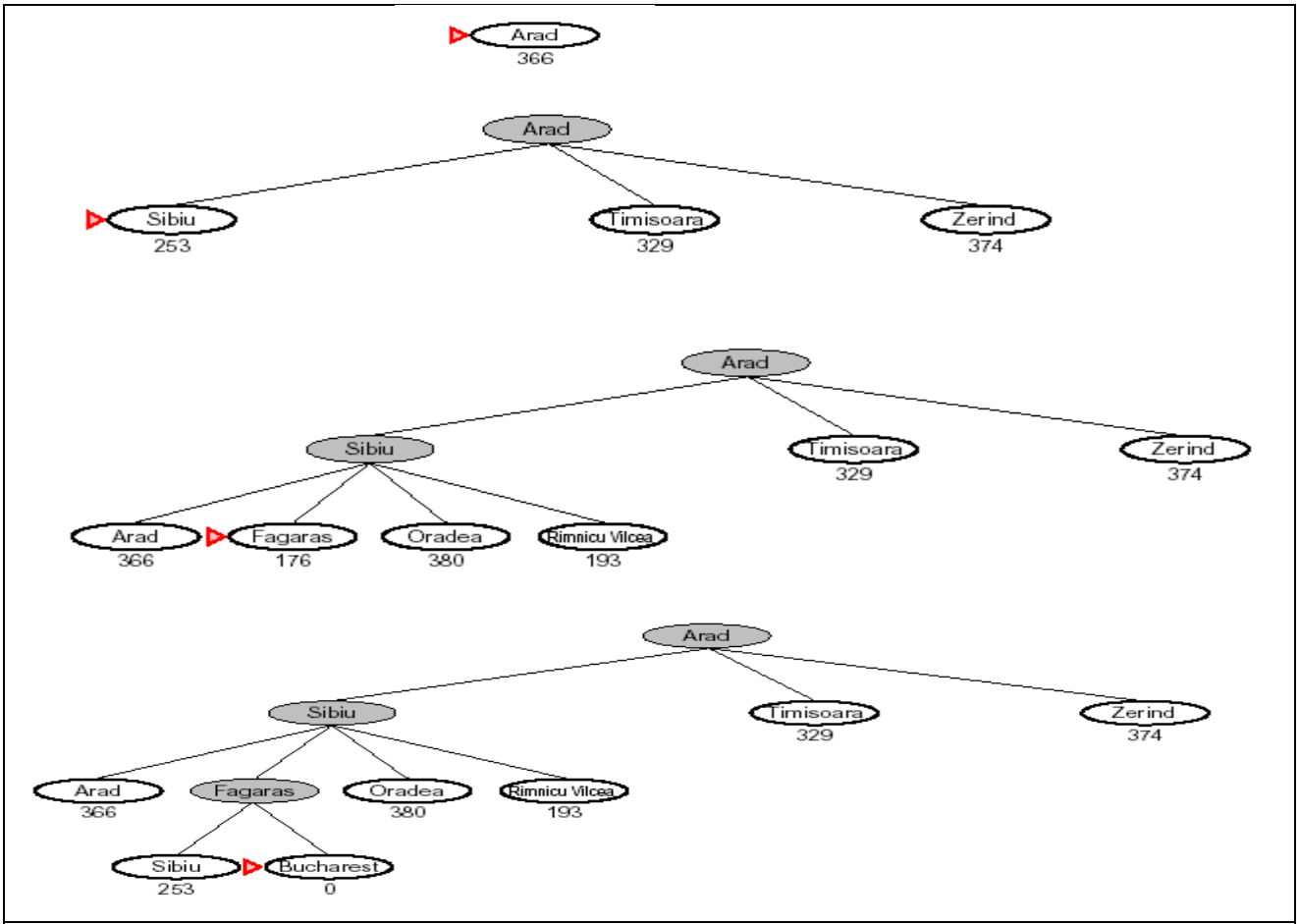


Figure 2.2 stages in greedy best-first search for Bucharest using straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Properties of greedy search

- **Complete??** No–can get stuck in loops, e.g.,

  Complete in finite space with repeated-state checking
- **Time??** O(bm), but a good heuristic can give dramatic improvement
- **Space??** 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.

Figure 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.

A* Search

A* Search is the most widely used form of best-first search. The evaluation function f(n) is obtained by combining

(1) **g(n) =** the cost to reach the node,and

(2) **h(n) =** the cost to get from the node to the **goal** :

f(n) = g(n) + h(n).

A* Search is both optimal and complete. A* is optimal if h(n) is an admissible heuristic. The obvious example of admissible heuristic is the straight-line distance h_{SLD}. It cannot be an overestimate.

A* Search is optimal if h(n) is an admissible heuristic – that is,provided that h(n) never overestimates the cost to reach the goal.

An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest. The progress of an A* tree search for Bucharest is shown in Figure. The values of ‘g ‘ are computed from the step costs shown in the Romania map

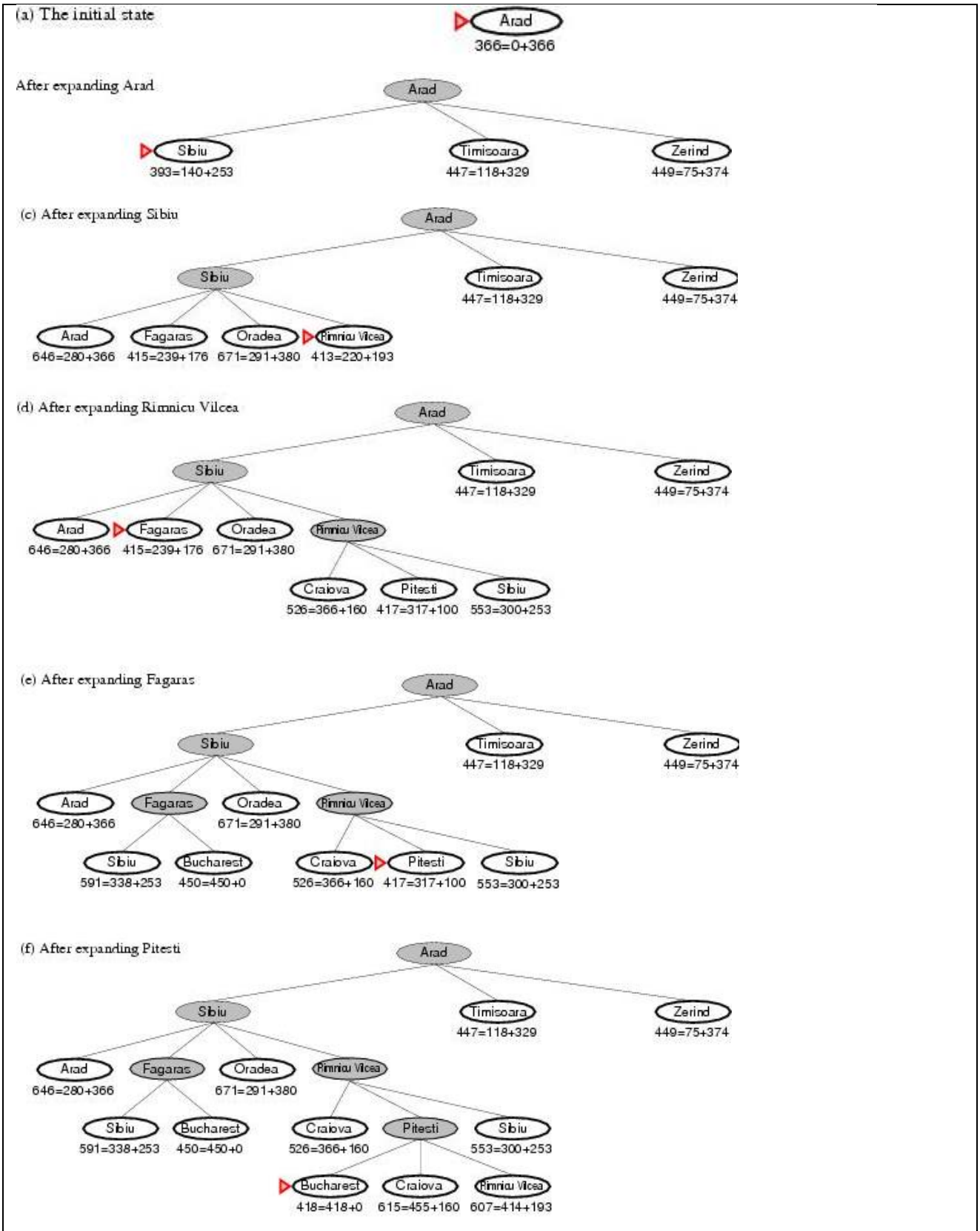


Figure 2.3 Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are

the straight-line distances to Bucharest taken from figure 2.1

Recursive Best-first Search(RBFS)

Recursive best-first search is a simple recursive algorithm that attempts to mimic the operation of standard best-first search,but using only linear space.

Figure 2.5 shows how RBFS reaches Bucharest.

The algorithm is shown in figure 2.4. Its structure is similar to that of recursive depth-first search,but rather than continuing indefinitely down the current path,it keeps track of the f-value of the best alternative path available from any ancestor of the current node. If the current node exceeds this limit,the recursion unwinds back to the alternative path. As the recursion unwinds,RBFS replaces the f-value of each node along the path with the best f-value of its children.

```
function RECURSIVE-BEST-FIRST-SEARCH(problem) return a solution or failure
    return RFBS(problem,MAKE-NODE(INITIAL-STATE[problem]),∞)

function RFBS( problem, node, f_limit) return a solution or failure and a new f-
    cost limit
    if GOAL-TEST[problem](STATE[node]) then return node
    successors ← EXPAND(node, problem)
    if successors is empty then return failure, ∞
    for each s in successors do
        f [s] ← max(g(s) + h(s), f [node])
    repeat
        best ← the lowest f-value node in successors
        if f [best] > f_limit then return failure, f [best]
        alternative ← the second lowest f-value among successors
        result, f [best] ← RBFS(problem, best, min(f_limit, alternative))
        if result ≠ failure then return result
```

Figure 2.4 The algorithm for recursive best-first search

RBFS **Evaluation** :

RBFS is a bit more efficient than IDA*

- Still excessive node generation (mind changes)

Like A*, optimal if *h(n)* is admissible

Space complexity is *O(bd)*.

- IDA* retains only one single number (the current f-cost limit)

Time complexity difficult to characterize

- Depends on accuracy if h(n) and how often best path changes.

IDA* en RBFS suffer from ***too little*** memory.

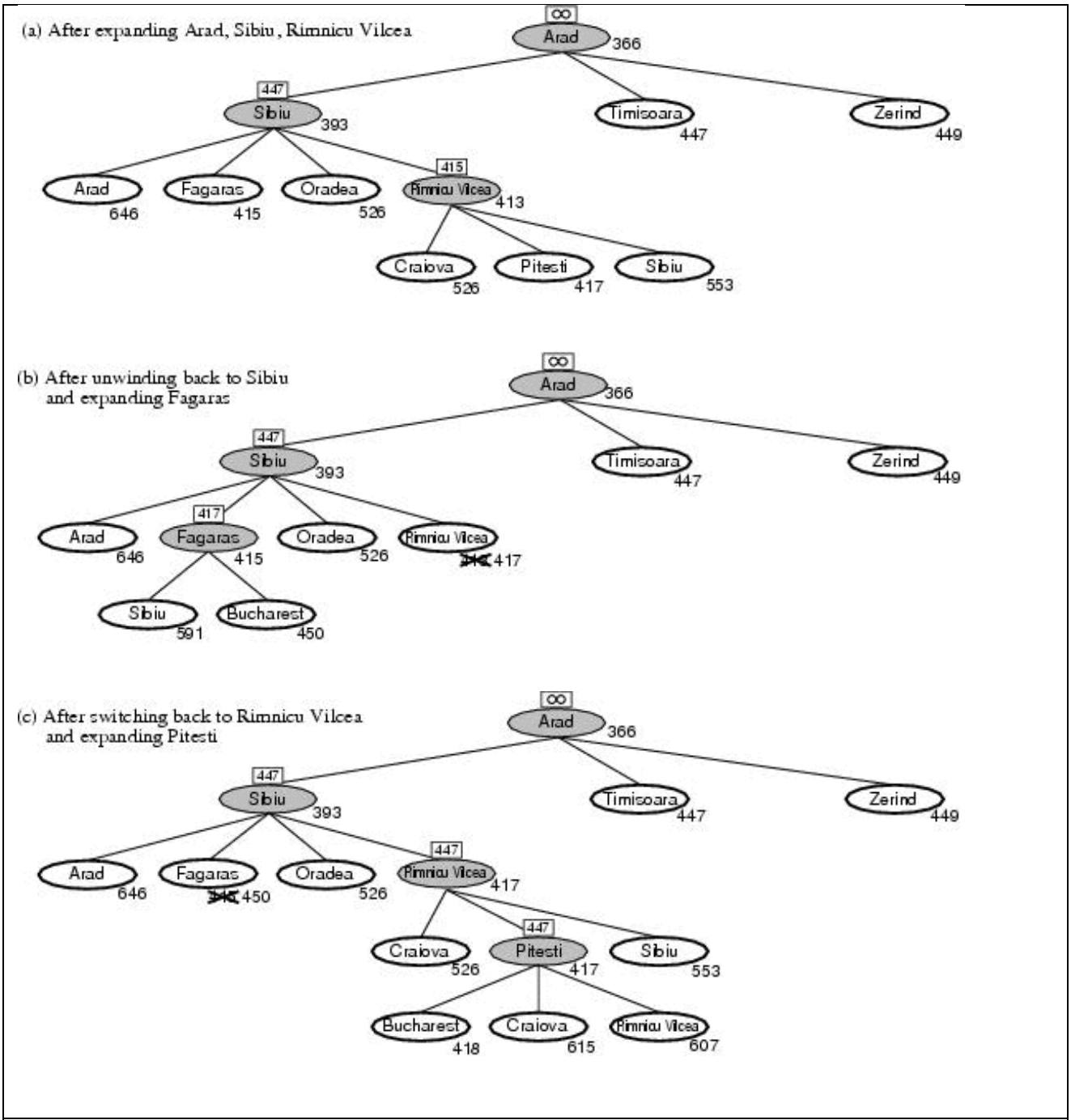


Figure 2.5 Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each

recursive call is shown on top of each current node.

(a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).

(b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.

(c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded. This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

2.2 Heuristic Functions

A **heuristic function** or simply a heuristic is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search

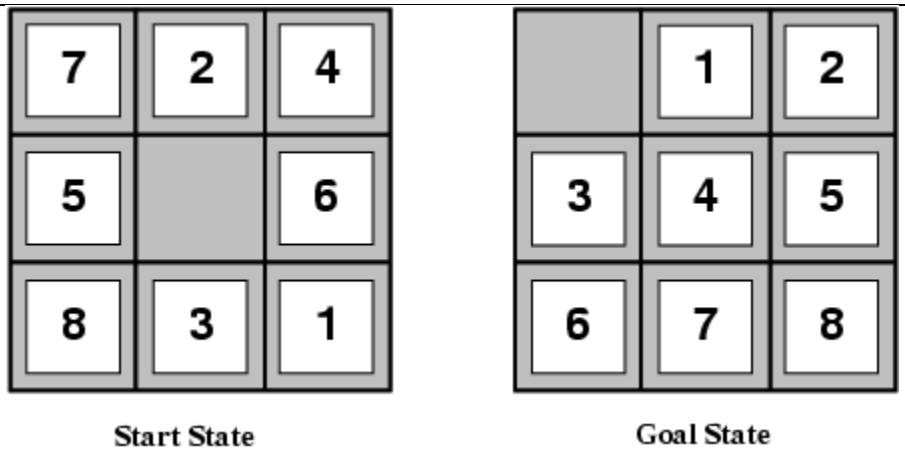


Figure 2.6 A typical instance of the 8-puzzle. The solution is 26 steps long.

The 8-puzzle

The 8-puzzle is an example of Heuristic search problem.

The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration (Figure 2.6)

The average cost for a randomly generated 8-puzzle instance is about 22 steps. The branching factor is about 3.

This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1 X 10^{10} states.By keeping track of repeated states,we could cut this down by a factor of about

170,000,because there are only 9!/2 = 181,440 distinct states that are reachable. This is a manageable number ,but the corresponding number for the 15-puzzle is roughly 10^{13}.

If we want to find the shortest solutions by using A^*,we need a heuristic function that never overestimates the number of steps to the goal.

The two commonly used heuristic functions for the 15-puzzle are :

(1) h_1 = the number of misplaced tiles.

For figure 2.6 ,all of the eight tiles are out of position, so the start state would have $h_1 = 8$. h_1 is an admissible heuristic.

(2) h_2 = the sum of the distances of the tiles from their goal positions. This is called **the city block distance** or **Manhattan distance.**

h_2 is admissible ,because all any move can do is move one tile one step closer to the goal.

Tiles 1 to 8 in start state give a Manhattan distance of

$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18.$

Neither of these overestimates the true solution cost ,which is 26.

2.3 LOCAL SEARCH ALGORITHMS AND OPTIMIZATION PROBLEMS

- In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
- For example,in the 8-queens problem,what matters is the final configuration of queens,not the order in which they are added.
- In such cases, we can **use local search algorithms.** They operate using a **single current state** and generally move only to neighbors of that state.
- The important applications of these class of problems are
  - (a) integrated-circuit design,
  - (b)Factory-floor layout,
  - (c)job-shop scheduling,
  - (d)automatic programming,
  - (e)telecommunications network optimization,
  - (f)Vehicle routing, and
  - (g) portfolio management.

Key advantages of Local Search Algorithms

(1) They use very little memory – usually a constant amount; and

(2) they can often find reasonable solutions in large or infinite(continuous) state spaces for which systematic algorithms are unsuitable.

OPTIMIZATION PROBLEMS

In addition to finding goals,local search algorithms are useful for solving pure **optimization problems**,in which the aim is to find the **best state** according to an **objective function**.

State Space Landscape

To understand local search,it is better explained using **state space landscape** as shown in figure 2.8.A landscape has both "**location**" (defined by the state) and "**elevation**"(defined by the value of the heuristic cost function or objective function).

If elevation corresponds to **cost**,then the aim is to find the **lowest valley** – a **global minimum**; if elevation corresponds to an **objective function**,then the aim is to find the **highest peak** – a **global maximum.**

Local search algorithms explore this landscape. A complete local search algorithm always finds a **goal** if one exists; an **optimal** algorithm always finds a **global minimum/maximum**.

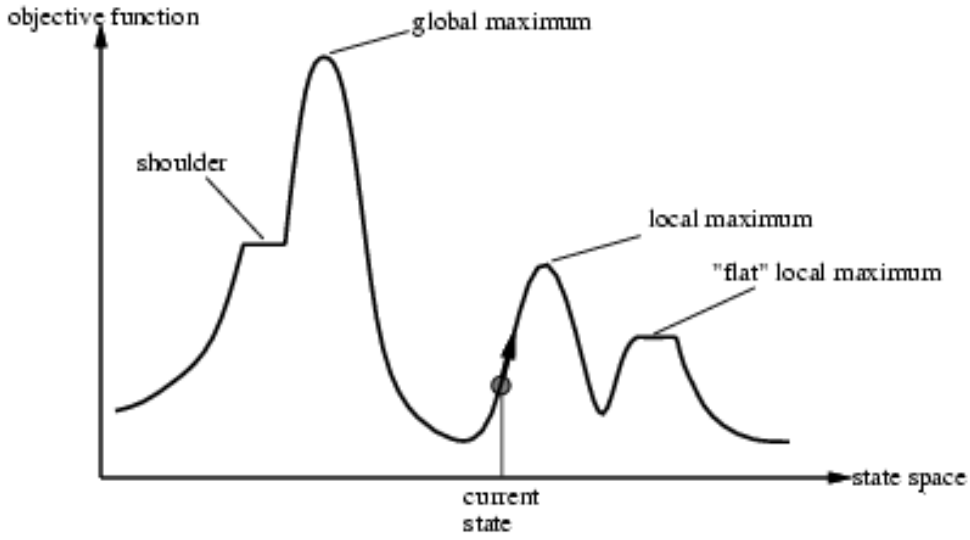


Figure 2.8 A one dimensional **state space landscape** in which elevation corresponds to the **objective function**. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it ,as shown by the arrow. The various topographic features are defined in the text

Hill-climbing search

The **hill-climbing** search algorithm as shown in figure 2.9, is simply a loop that continually moves in the direction of increasing value – that is,**uphill**. It terminates when it reaches a "**peak**" where no neighbor has a higher value.

```
function HILL-CLIMBING( problem) return a state that is a local maximum
    input: problem, a problem
    local variables: current, a node.   neighbor, a node.
    current ← MAKE-NODE(INITIAL-STATE[problem])
    loop do
        neighbor ← a highest valued successor of current
        if VALUE [neighbor] ≤ VALUE[current] then return STATE[current]
        current ← neighbor
```

Figure 2.9 The hill-climbing search algorithm (steepest ascent version),which is the most basic local search technique. At each step the current node is replaced by the best neighbor; the neighbor with the highest VALUE. If the heuristic cost estimate h is used,we could find the neighbor with the lowest h.

Hill-climbing is sometimes called greedy local search because it grabs a good neighbor state without thinking ahead about where to go next. Greedy algorithms often perform quite well.

Problems with hill-climbing

Hill-climbing often gets stuck for the following reasons :

- **Local maxima** : a local maximum is a peak that is higher than each of its neighboring states,but lower than the global maximum. Hill-climbing algorithms that reach the vicinity of a local maximum will be drawn upwards towards the peak,but will then be stuck with nowhere else to go
- **Ridges** : A ridge is shown in Figure 2.10. Ridges results in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
- **Plateaux** : A plateau is an area of the state space landscape where the evaluation function is flat. It can be a flat local maximum,from which no uphill exit exists,or a shoulder,from which it is possible to make progress.

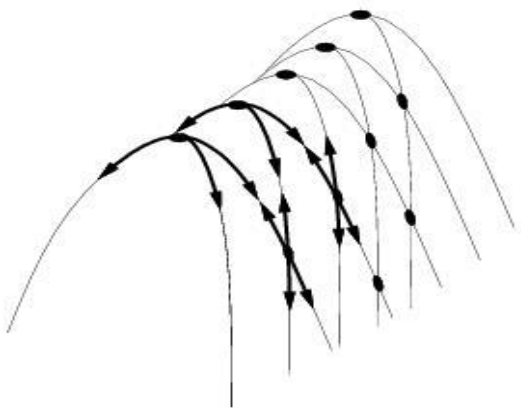


Figure 2.10 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right,creating a sequence of local maxima that are not directly connected to each other. From each local maximum,all th available options point downhill.

Hill-climbing variations

- **Stochastic hill-climbing**
  - Random selection among the uphill moves.
  - The selection probability can vary with the steepness of the uphill move.
- **First-choice hill-climbing**
  - cfr. stochastic hill climbing by generating successors randomly until a better one is found.
- **Random-restart hill-climbing**
  - Tries to avoid getting stuck in 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 choosen 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 someway that yields both efficiency and completeness. 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 E by which the evaluation is worsened.

```
function SIMULATED-ANNEALING(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 prob. 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^(Δ E/T)
```

Figure 2.11 The simulated annealing search algorithm,a version of stochastic hill climbing where some downhill moves are allowed.

Genetic algorithms

A Genetic algorithm(or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states,rather than by modifying a single state.

Like beam search,Gas begin with a set of k randomly generated states,called the population. Each state,or individual,is represented as a string over a finite alphabet – most commonly,a string of 0s and 1s. For example,an 8 8-quuens state must specify the positions of 8 queens,each in acolumn of 8 squares,and so requires 8 x $\log_2 8$ = 24 bits.

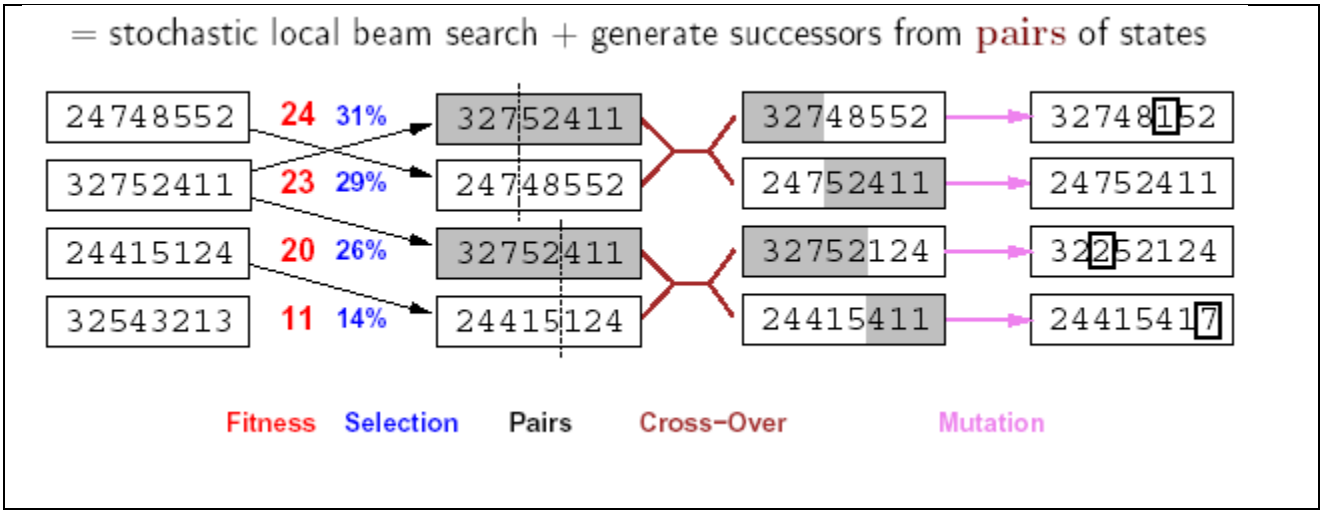


Figure 2.12 The genetic algorithm. The initial population in (a) is ranked by the fitness function in (b),resulting in pairs for mating in (c). They produce offspring in (d),which are subjected to mutation in (e).

Figure 2.12 shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure 2.12(b) to (e).

In (b) each state is rated by the evaluation function or the **fitness function**.

In (c),a random choice of two pairs is selected for reproduction,in accordance with the probabilities in (b).

```
function GENETIC_ALGORITHM( population, FITNESS-FN) return an individual
    input: population, a set of individuals
           FITNESS-FN, a function which determines the quality of the individual
    repeat
        new_population ← empty set
        loop for i from 1 to SIZE(population) do
            x ← RANDOM_SELECTION(population, FITNESS_FN)
        y ← RANDOM_SELECTION(population, FITNESS_FN)
            child ← REPRODUCE(x,y)
            if (small random probability) then child ← MUTATE(child )
            add child to new_population
        population ← new_population
    until some individual is fit enough or enough time has elapsed
    return the best individual
```

Figure 2.13 A genetic algorithm. The algorithm is same as the one diagrammed in Figure 2.12,with one variation:each mating of two parents produces only one offspring,not two.

2.4 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.

2.5 SEARCH WITH NON-DETERMINISTIC ACTIONS

In a partially observable environment, every percept helps narrow down the set of possible states the agent might be in, thus making it easier for the agent to achieve its goals.

When the environment is nondeterministic, percepts tell the agent which of the possible outcomes of its actions has actually occurred.

In both cases, the future percepts cannot be determined in advance and the agent's future actions will depend on those future percepts.

So the solution to a problem is not a sequence but a contingency plan that specifies what to do depending on what percepts are received.

2.5.1 The erratic vacuum world

As an example, we use the vacuum world, first and defined as a search problem. The state space has eight states, as shown in Figure 4.9.

There are three actions—Left, Right, and Suck—and the goal is to clean up all the dirt (states 7 and 8).

If the environment is observable, deterministic, and completely known, then the problem is trivially solvable by any of the algorithms and the solution is an action sequence.

For example, if the initial state is 1, then the action sequence [Suck,Right,Suck] will reach a goal state, 8.

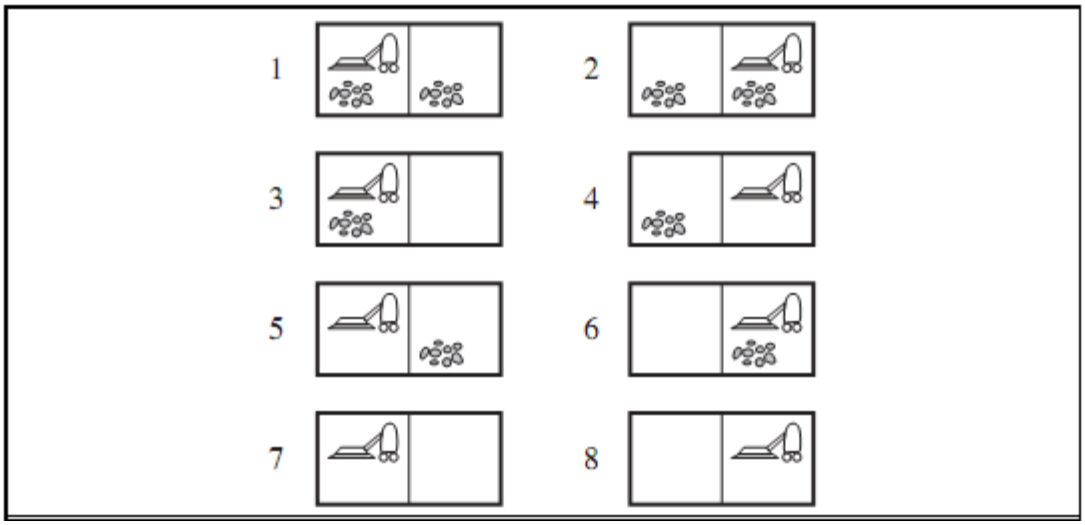


Figure 2.14 The eight possible states of the vacuum world; states 7 and 8 are goal states

In the **erratic vacuum world,** the Suck action works as follows:

• When applied to a dirty square the action cleans the square and sometimes cleans up dirt in an adjacent square, too.

• When applied to a clean square the action sometimes deposits dirt on the carpet.

To provide a precise formulation of this problem, we need to generalize the notion of a transition model.

Instead of defining the transition model by a RESULT function that returns a single state, we use a RESULTS function that returns a set of possible outcome states.

For example, in the erratic vacuum world, the Suck action in state 1 leads to a state in the set {5, 7}—the dirt in the right-hand square may or may not be vacuumed up.

[Suck, if State =5 then [Right, Suck] else []] . (4.3)

Thus, solutions for nondeterministic problems can contain nested if–then–else statements; this means that they are trees rather than sequences.

2.5.2 AND–OR search trees

An extension of a search tree introduced in deterministic environments.

We call these nodes OR nodes. In the vacuum world, for example, at an OR node the agents own choice in each state chooses Left or Right or Suck.

In a non-deterministic environment, branching is also introduced by the environment's choice of outcome for each action. We call these nodes AND nodes. For example, the Suck action

in state 1 leads to a state in the set {5, 7},so the agent would need to find a plan for state 5 and for state 7.

These two kinds of nodes alternate, leading to an AND–OR tree as illustrated in Figure 2.14.

A solution for an AND–OR search problem is a subtree that

(1) has a goal node at every leaf,

(2) specifies one action at each of its OR nodes, and

(3) includes every outcome branch at each of its AND nodes.

The solution is shown in bold lines in the figure; it corresponds to the plan given in Equation.

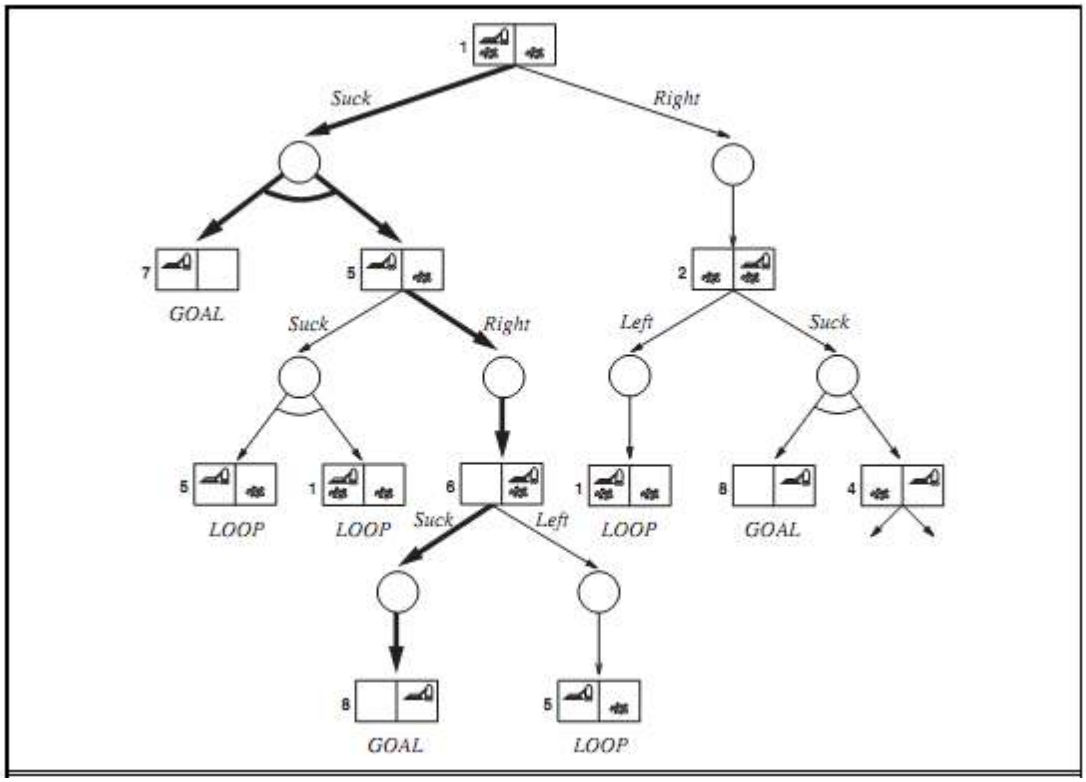


Figure 2.14 The first two levels of the search tree for the erratic vacuum world. State nodes are OR nodes where some action must be chosen. At the AND nodes, shown as circles, every outcome must be handled, as indicated by the arc linking the outgoing branches. The solution found is shown in bold lines

```
function AND-OR-GRAPH-SEARCH(problem) returns a conditional plan, or failure
  OR-SEARCH(problem.INITIAL-STATE, problem, [])

function OR-SEARCH(state, problem, path) returns a conditional plan, or failure
  if problem.GOAL-TEST(state) then return the empty plan
  if state is on path then return failure
  for each action in problem.ACTIONS(state) do
      plan ← AND-SEARCH(RESULTS(state, action), problem, [state | path])
      if plan ≠ failure then return [action | plan]
  return failure

function AND-SEARCH(states, problem, path) returns a conditional plan, or failure
  for each s_i in states do
      plan_i ← OR-SEARCH(s_i, problem, path)
      if plan_i = failure then return failure
  return [if s_1 then plan_1 else if s_2 then plan_2 else ... if s_{n-1} then plan_{n-1} else plan_n]
```

Figure 2.15 An algorithm for searching AND–OR graphs generated by nondeterministic environments. It returns a conditional plan that reaches a goal state in all circumstances. (The notation [x | l] refers to the list formed by adding object x to the front of list l.)

The concept of a heuristic function must be modified to estimate the cost of a contingent solution rather than a sequence, but the notion of admissibility carries over and there is an analog of the A algorithm for finding optimal solutions

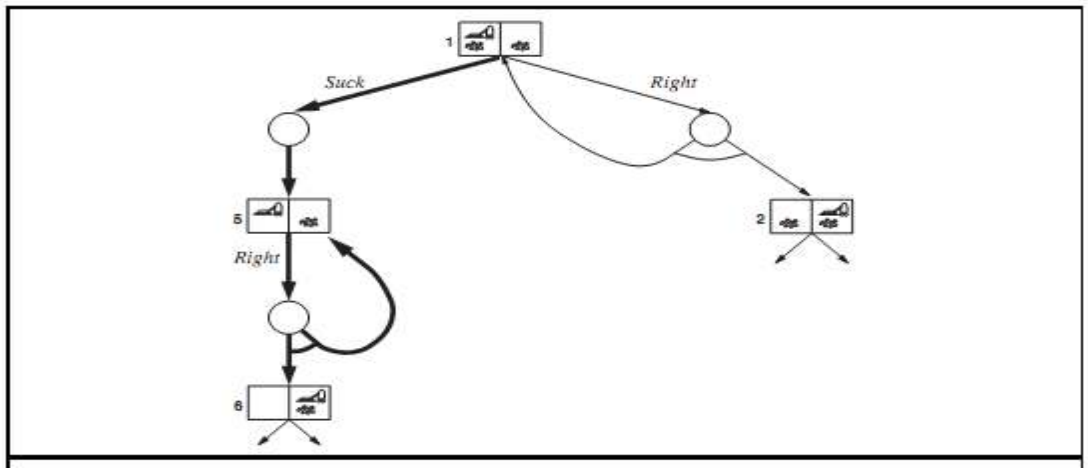


Figure 2.16 Part of the search graph for the slippery vacuum world, where we have shown (some) cycles explicitly. All solutions for this problem are cyclic plans because there is no way to move reliably

2.6 SEARCH IN PARTIALLY OBSERVABLE ENVIRONMENTS

The key concept required for solving partially observable problems is the belief state, representing the agent's current belief about the possible physical states it might be in, given the sequence of actions and percepts up to that point.

The simplest scenario for studying belief states, which is when the agent has no sensors at all; then we add in partial sensing as well as nondeterministic actions.

Searching with no observation

When the agent's percepts provide no information at all, we have what is called a sensorless problem or sometimes a conformant problem.

At first, one might think the sensorless agent has no hope of solving a problem if it has no idea what state it's in; in fact, sensorless problems are quite often solvable.

Sensorless agents can be surprisingly useful, primarily because

they don't rely on sensors working properly.

The high cost of sensing is another reason to avoid it:

for example, doctors often prescribe a broad spectrum antibiotic rather than using the contingent plan of doing an expensive blood test, then waiting for the results to come back, and then prescribing a more specific antibiotic and perhaps hospitalization because the infection has progressed too far.

The belief-state search problem is constructed. Suppose the underlying physical problem P is defined by $ACTIONS_P$,$RESULT_P$,GOAL-TESTp,and STEP-COSTp. Then we can define the corresponding sensorless problem as follows:

• **Belief states:** The entire belief-state space contains every possible set of physical states. If P has N states, then the sensorless problem has up to 2N P states, although many may be unreachable from the initial state.

• **Initial state:** The set of all states in P, although in some cases the agent will have more knowledge than this.

• **Actions:** This is slightly tricky. Suppose the agent is in belief state b ={s1,s2},but $ACTIONS_P$ (s1) ≠$ACTIONS_p$(s2); then the agent is unsure of which actions are legal.

If we assume that illegal actions have no effect on the environment, then it is safe to take the union of all the actions in any of the physical states in the current belief state b:

$$\text{ACTIONS}(b) = \bigcup_{s \in b} \text{ACTIONS}_P(s) .$$

.

If an illegal action might be the end of the world, it is safer to allow only the intersection, that is, the set of actions legal in all the states.

• **Transition model:** The process of generating the new belief state after the action is called the prediction step

For deterministic actions, b

$$b' = \text{RESULT}(b, a) = \{s' : s' = \text{RESULT}_P(s, a) \text{ and } s \in b\} .$$

is never larger than b. For nondeterminism, we have

$$\begin{aligned} b' = \text{RESULT}(b, a) &= \{s' : s' \in \text{RESULTS}_P(s, a) \text{ and } s \in b\} \\ &= \bigcup_{s \in b} \text{RESULTS}_P(s, a) , \end{aligned}$$

which may be larger than b,

Goal test: A belief state P satisfies the goal only if all the physical states in it satisfy GOAL-TEST$_p$.

• **Path cost:** Assume that the cost of an action is the same in all states and so can be transferred directly from the underlying physical problem.

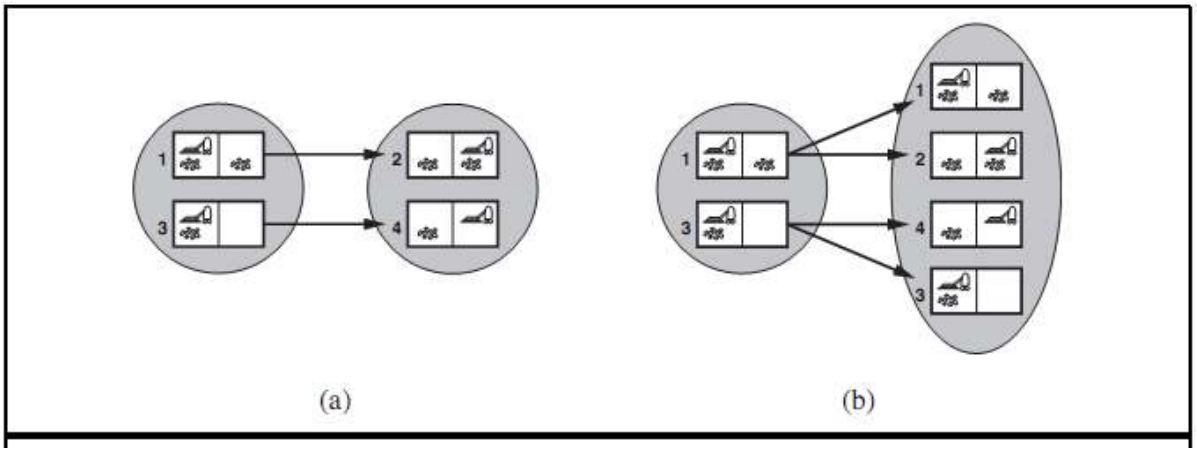


Figure 2.17 a) Predicting the next belief state for the sensorless vacuum world with a deterministic action, Right . (b) Prediction for the same belief state and action in the slippery version of the sensorless vacuum world.

Figure 2.18 shows the reachable belief-state space for the deterministic, sensorless vacuum world. There are only 12 reachable belief states out of 2^8=256 possible belief states.

This works for belief states, too; for example, in Figure 2.18, the action sequence [Suck,Left,Suck] starting at the initial state reaches the same belief state as [Right,Left,Suck], namely, {5, 7}.

Now, consider the belief state reached by [Left], namely, {1, 3, 5, 7}.

Obviously, this is not identical to {5, 7}, but it is a superset.

It is easy to prove that if an action sequence is a solution for a belief state b, it is also a solution for any subset of b.

Hence, we can discard a path reaching {1, 3, 5, 7} if {5, 7} has already been generated.

Conversely, if {1, 3, 5, 7} has already been generated and found to be solvable, then any subset,suchas{5, 7}, is guaranteed to be solvable.

This extra level of pruning may dramatically improve the efficiency of sensorless problem solving.

Even with this improvement, however, sensorless problem-solving as we have described it is seldom feasible in practice.

The difficulty is not so much the vastness of the belief-state space—even though it is exponentially larger than the underlying physical state space.

The real difficulty lies with the size of each belief state. For example, the initial belief state for the 10 ×10 vacuum world contains 100 $\times 2^{100}$ or around 10 physical states—far too many if we use the atomic representation, which is an explicit list of states.

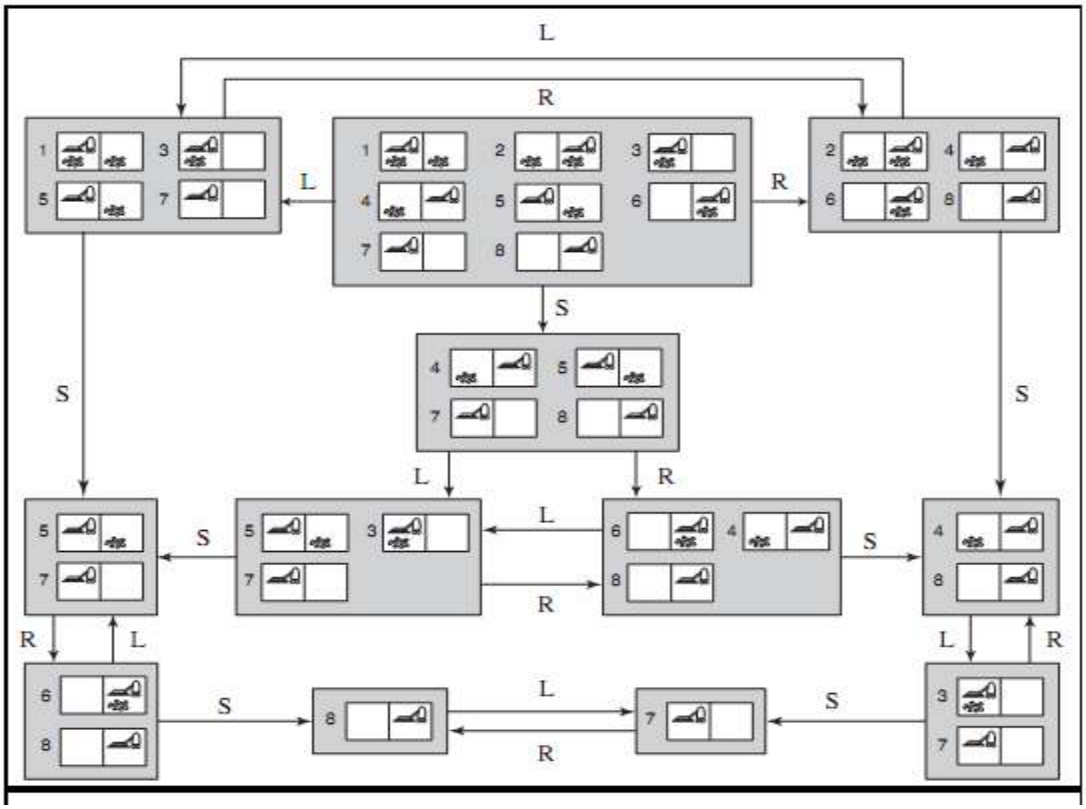


Figure 2.18 The reachable portion of the belief-state space for the deterministic, sensorless vacuum world. Each shaded box corresponds to a single belief state. At any given point, the agent is in a particular belief state but does not know which physical state it is in. The initial belief state (complete ignorance) is the top center box. Actions are represented by labeled links. Self-loops are omitted for clarity.

Searching with observations

For a general partially observable problem, we have to specify how the environment generates percepts for the agent.

For example, we might define the local-sensing vacuum world to be one in which the agent has a position sensor and a local dirt sensor but has no sensor capable of detecting dirt in other squares.

The formal problem specification includes a PERCEPT(s) function that returns the percept received in a given state.

For example, in the local-sensing vacuum world, the PERCEPT in state 1 is [A, Dirty].

Fully observable problems are a special case in which PERCEPT(s)=s for every state s, while sensorless problems are a special case in which PERCEPT(s)=null .

When observations are partial, it will usually be the case that several states could have produced any given percept.

For example, the percept [A, Dirty] is produced by state 3 as well as by state 1. Hence, given this as the initial percept, the initial belief state for the local-sensing vacuum world will be {1, 3}.

The ACTIONS,STEP-COST,andGOAL-TEST are constructed from the underlying physical problem just as for sensorless problems, but the transition model is a bit more complicated.

• The **prediction** stage is the same as for sensorless problems: given the action a in belief state b, the predicted belief state is b =PREDICT(b, a).

• The **observation prediction** stage determines the set of percepts o that could be observed in the predicted belief state:

$$\text{POSSIBLE-PERCEPTS}(\hat{b}) = \{o : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\} .$$

• The **update stage** determines, for each possible percept, the belief state that would result from the percept. The new belief state b have produced the percept:

$$b_o = \text{UPDATE}(\hat{b}, o) = \{s : o = \text{PERCEPT}(s) \text{ and } s \in \hat{b}\} .$$

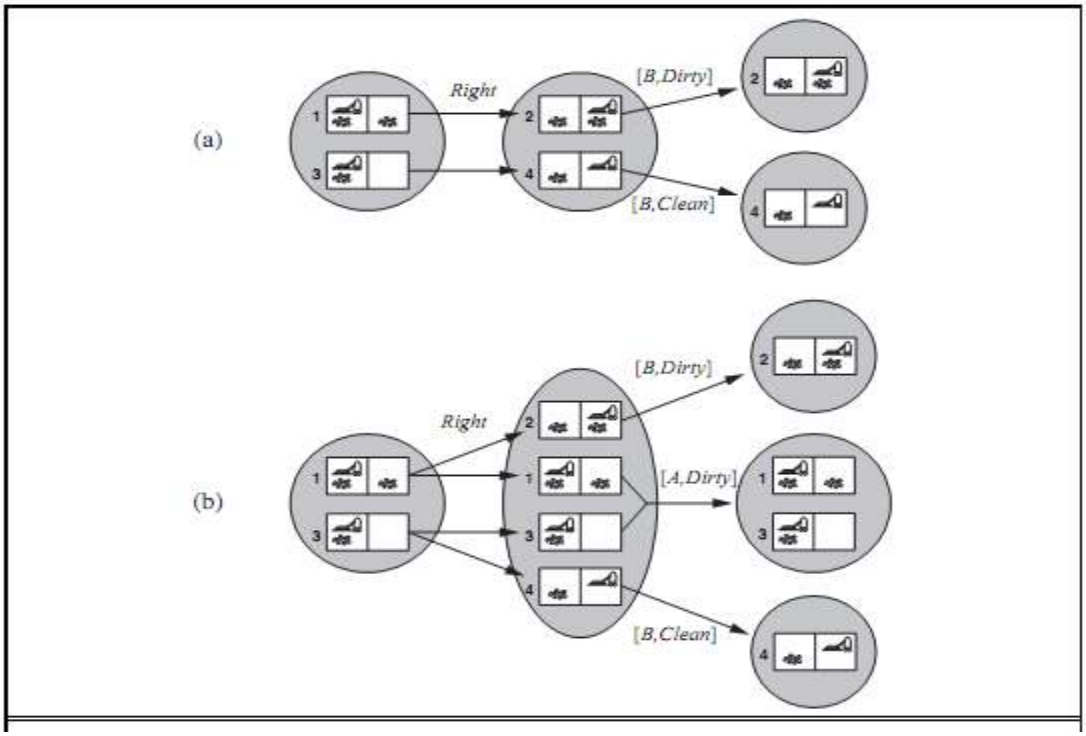


Figure 2.19 Two example of transitions in local-sensing vacuum worlds.

(a) In the deterministic world, Right is applied in the initial belief state, resulting in a new belief state with two possible physical states; for those states, the possible percepts are [B,Dirty] and [B,Clean], leading to two belief states, each of which is a singleton.

(b) In the slippery world, Right is applied in the initial belief state, giving a new belief state with four physical states; for those states, the possible percepts are [A, Dirty], [B,Dirty],and[B,Clean], leading to three belief states as shown.

Putting these three stages together, we obtain the possible belief states resulting from a given action and the subsequent possible percepts:

$$\text{RESULTS}(b, a) = \{b_o : b_o = \text{UPDATE}(\text{PREDICT}(b, a), o) \text{ and } o \in \text{POSSIBLE-PERCEPTS}(\text{PREDICT}(b, a))\} . \quad 2.5$$

Solving partially observable problems

Applying AND–OR search algorithm on the constructed belief state space

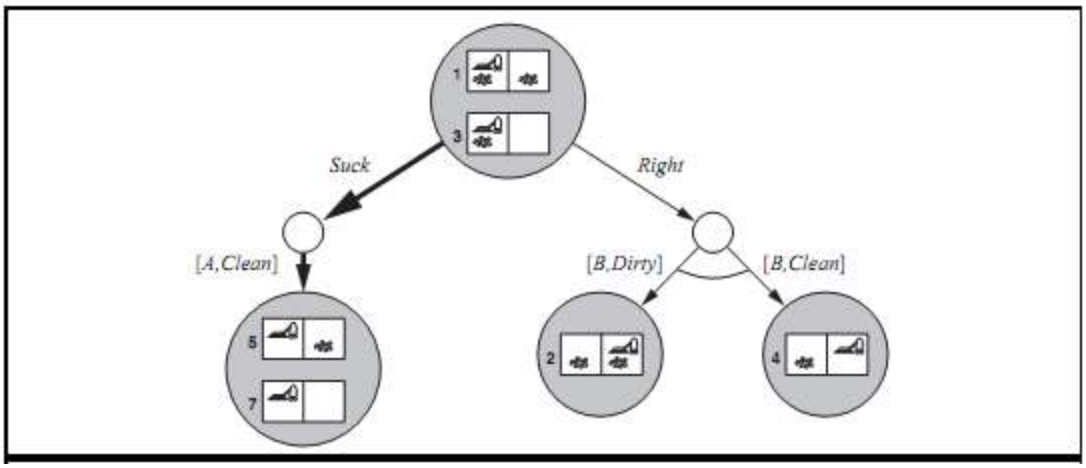


Figure 2.20 The first level of the AND–OR search tree for a problem in the local-sensing vacuum world; Suck is the first step of the solution.

Figure 2.20 shows part of the search tree for the local-sensing vacuum world, assuming an initial percept [A, Dirty].

The solution is the conditional plan

[Suck, Right, if Bstate={6} then Suck else []] .

Notice that, because we supplied a belief-state problem to the AND–OR search algorithm, it returned a conditional plan that tests the belief state rather than the actual state.

An agent for partially observable environments

The design of a problem-solving agent for partially observable environments is quite similar to the simple problem-solving agent : the agent formulates a problem, calls a search algorithm to solve it, and executes the solution.

There are two main differences.

First, the solution to a problem will be a conditional plan rather than a sequence; if the first step is an if–then–else expression, the agent will need to test the condition in the if-part and execute the then-part or the else-part accordingly.

Second, the agent will need to maintain its belief state as it performs actions and receives percepts.

This process resembles the prediction–observation–update process in Equation (2.5) but is actually simpler because the percept is given by the environment rather than calculated by the

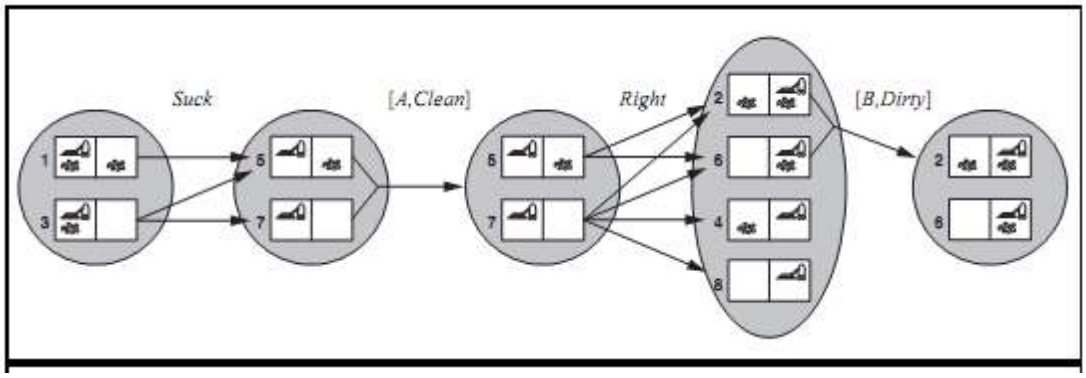


Figure 2.21 Two prediction–update cycles of belief-state maintenance in the kindergarten vacuum world with local sensing.

agent. Given an initial belief state b, an action a, and a percept o, the new belief state is:

b= UPDATE(PREDICT(b, a),o) . (2.6)

Figure 2.21 shows the belief state being maintained in the kindergarten vacuum world with _local sensing, wherein any square may become dirty at any time unless the agent is actively cleaning it at that moment.

In partially observable environments—which include the vast majority of real-world environments—maintaining one's belief state is a core function of any intelligent system.

This function goes under various names, including monitoring, filtering and state estimation.

Equation (2.6) is called a recursive state estimator because it computes the new belief state from the previous one rather than by examining the entire percept sequence.

If the agent is not to "fall behind," the computation has to happen as fast as percepts are coming in.

As the environment becomes more complex, the exact update computation becomes infeasible and the agent will have to compute an approximate belief state, perhaps focusing on the implications of the percept for the aspects of the environment that are of current interest

The example concerns a robot with the task of localization: working out where it is, given a map of the world and a sequence of percepts and actions.

Our robot is placed in the maze-like environment of Figure 2.22.

The robot is equipped with four sonar sensors that tell whether there is an obstacle—the outer wall or a black square in the figure—in each of the four compass directions.

We assume that the sensors give perfectly correct data, and that the robot has a correct map of the enviornment.

But unfortunately the robot's navigational system is broken, so when it executes a Move action, it moves randomly to one of the adjacent squares.

The robot's task is to determine its current location.

Suppose the robot has just been switched on, so it does not know where it is.

Thus its initial belief state consists of the set of all locations.

Th robot receives the percept

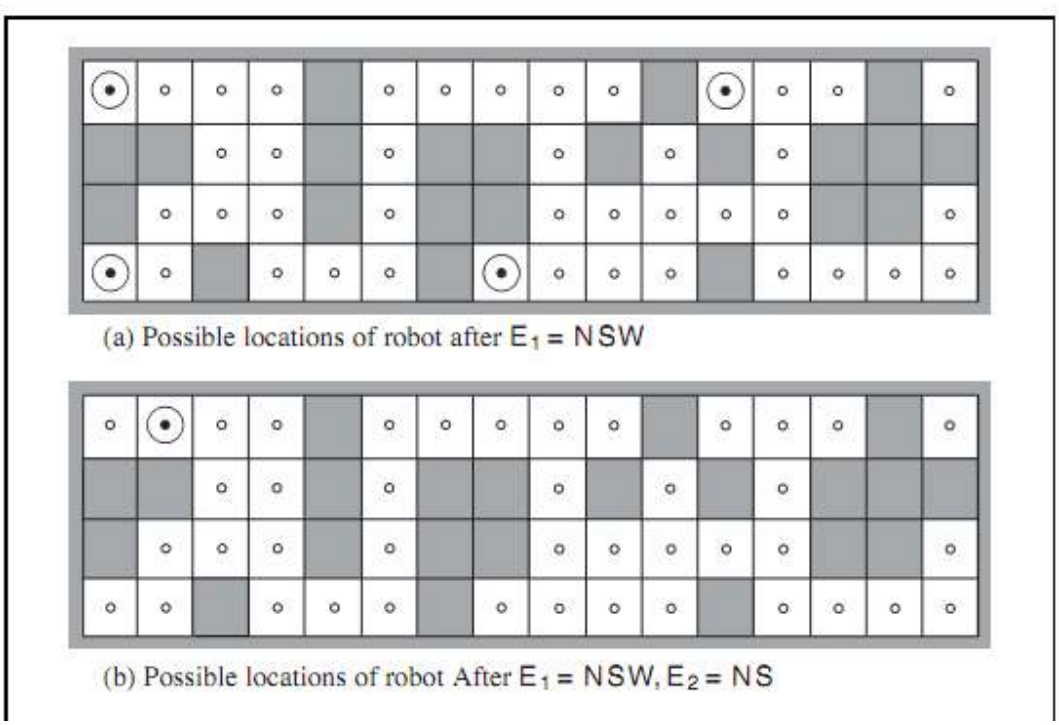


Figure 2.22 Possible positions of the robot, , (a) after one observation E =NSW and (b) after a second observation E

=NS. When sensors are noiseless and the transition model is accurate, there are no other possible locations for the robot consistent with this sequence of two observations.

NSW, meaning there are obstacles to the north, west, and south, and does an update using the equation b=UPDATE(b), yielding the 4 locations shown in Figure 2.22 (a).

can inspect the maze to see that those are the only four locations that yield the percept NWS.

Next the robot executes a Move action, but the result is nondeterministic.

The new belief state, b_a=PREDICT(b, Move), contains all the locations that are one step away from the locations in b_o.

When the second percept, NS, arrives, the robot does UPDATE(b, NS) and finds that the belief state has collapsed down to the single location shown in Figure 4.18(b).That's the only location that could be the result of

UPDATE(PREDICT(UPDATE(b, NSW), Move), NS) .

With nondetermnistic actions the PREDICT step grows the belief state, but the UPDATE step shrinks it back down—as long as the percepts provide some useful identifying information.

Sometimes the percepts don't help much for localization:

If there were one or more long east-west corridors, then a robot could receive a long sequence of NS percepts, but never know where in the corridor(s) it was.

2.7 ONLINE SEARCH AGENTS AND UNKNOWN ENVIRONMENTS

An online search agent interleaves computation and action: first it takes an action, then it observes the environment and computes the next action.

Online search is a good idea in

dynamic or semidynamic domains

nondeterministic domains because it allows the agent to focus its computational efforts on the contingencies that actually arise rather than those that might happen but probably won't.

unknown environments, where the agent does not know what states exist or what its actions do

Consider a newborn baby: it has many possible actions but knows the outcomes of none of them, and it has experienced only a few of the possible states that it can reach. The baby's gradual discovery of how the world works is, in part, an online search process.

Online search problems

An online search problem must be solved by an agent executing actions, rather than by pure computation. We assume a deterministic and fully observable environment but we stipulate that the agent knows only the following:

• ACTIONS(s), which returns a list of actions allowed in state s;

• The step-cost function c(s, a, s)—note that this cannot be used until the agent knows that s is the outcome; and

• GOAL-TEST(s).

the agent cannot determine RESULT(s, a) except by actually being in s and doing a.

For example, in the maze problem shown in Figure 4.19, the agent does not know that going Up from (1,1) leads to (1,2); nor, having done that, does it know that going Down will take it back to (1,1).

This degree of ignorance can be reduced in some applications—for example, a robot explorer might know how its movement actions work and be ignorant only of the locations of obstacles.

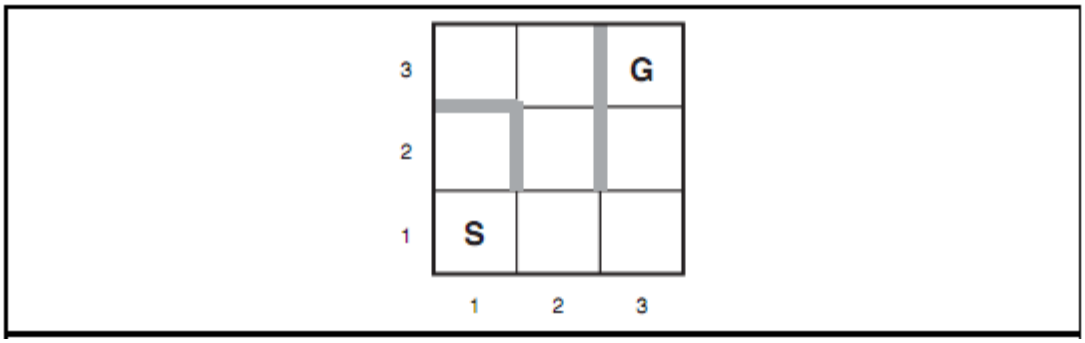


Figure 2.23 A simple maze problem. The agent starts at S and must reach G but knows nothing of the environment.

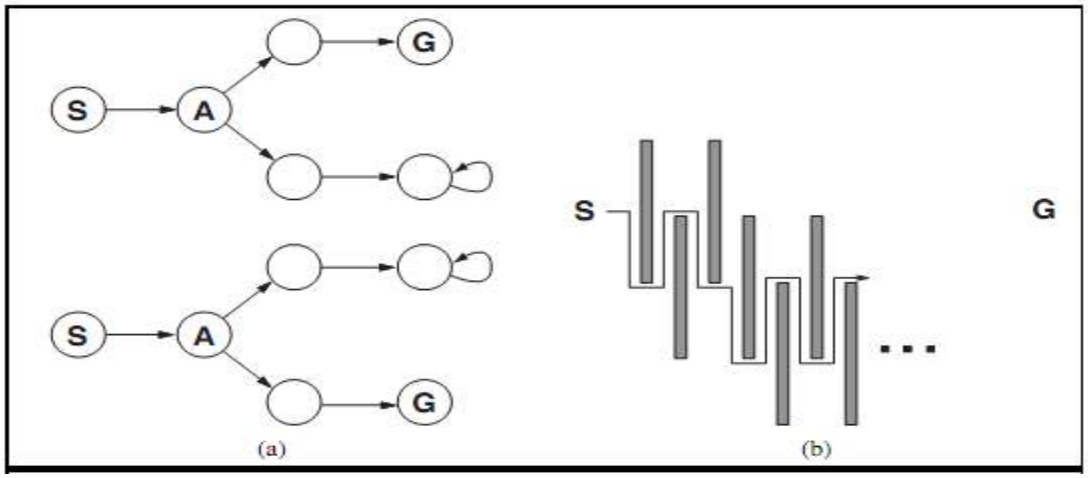


Figure 2.24 (a) Two state spaces that might lead an online search agent into a dead end. Any given agent will fail in at least one of these spaces. (b) A two-dimensional environment that can cause an online search agent to follow an arbitrarily inefficient route to the goal.

Finally, the agent might have access to an admissible heuristic function h(s) that estimates the distance from the current state to a goal state.

For example, in Figure 4.19, the agent might know the location of the goal and be able to use the Manhattan-distance heuristic.

Typically, the agent's objective is to reach a goal state while minimizing cost.

The cost is the total path cost of the path that the agent actually travels.

The state space is safely explorable—that is, some goal state is reachable from every reachable state.

State spaces with reversible actions, such as mazes and 8-puzzles, can be viewed as undirected graphs and are clearly safely explorable.

Even in safely explorable environments, no bounded competitive ratio can be guaranteed if there are paths of unbounded cost.

Online search agents

After each action, an online agent receives a percept telling it what state it has reached; from this information, it can augment its map of the environment.

The current map is used to decide where to go next.

This interleaving of planning and action means that online search algorithms are quite different from the offline search algorithms we have seen previously.

For example, offline algorithms can expand a node in one part of the space and then immediately expand a node in another part of the space, because node expansion involves simulated rather than real actions.

An online algorithm, on the other hand, can discover successors only for a node that it physically occupies. To avoid traveling all the way across the tree to expand the next node, it seems better to expand nodes in a local order.

Depth-first search has exactly this property because (except when backtracking) the next node expanded is a child of the previous node expanded.

```
function ONLINE-DFS-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table indexed by state and action, initially empty
              untried, a table that lists, for each state, the actions not yet tried
              unbacktracked, a table that lists, for each state, the backtracks not yet tried
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in untried) then untried[s'] ← ACTIONS(s')
  if s is not null then
      result[s, a] ← s'
      add s to the front of unbacktracked[s']
  if untried[s'] is empty then
      if unbacktracked[s'] is empty then return stop
      else a ← an action b such that result[s', b] = POP(unbacktracked[s'])
  else a ← POP(untried[s'])
  s ← s'
  return a
```

Figure 2.25 An online search agent that uses depth-first exploration. The agent is applicable only in state spaces in which every action can be "undone" by some other action.

Online local search

Hill climbing search has the property of locality in its node expansions.

In fact, because it keeps just one current state in memory, hill-climbing search is already an online search algorithm!

Unfortunately, it is not very useful in its simplest form because it leaves the agent sitting at local maxima with nowhere to go.

Moreover, random restarts cannot be used, because the agent cannot transport itself to a new state.

A random walk simply selects at random one of the available actions from the current state; preference can be given to actions that have not yet been tried.

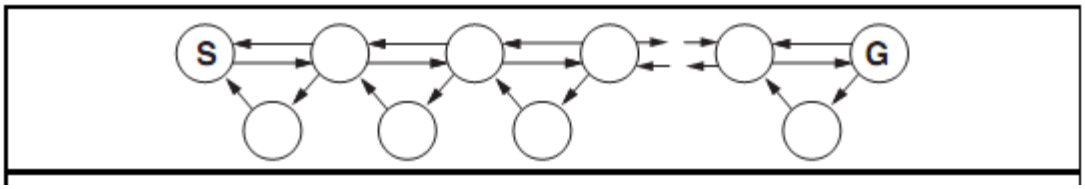


Figure 2.26 An environment in which a random walk will take exponentially many steps to find the goal.

Solutions:

It is easy to prove that a random walk will eventually find a goal or complete its exploration, provided that the space is finite.

Augmenting hill climbing with memory rather than randomness turns out to be a more effective approach.

The basic idea is to store a "current best estimate" H(s) of the cost to reach the goal from each state that has been visited.

H(s) starts out being just the heuristic estimate h(s) and is updated as the agent gains experience in the state space.

An agent implementing this scheme, which is called learning real-time as shown in Figure

Like ONLINE-DFS-AGENT, it builds a map of the environment in the result table.

This **optimism under uncertainty** encourages the agent to explore new, possibly promising paths.

An LRTA agent is guaranteed to find a goal in any finite, safely explorable environment.

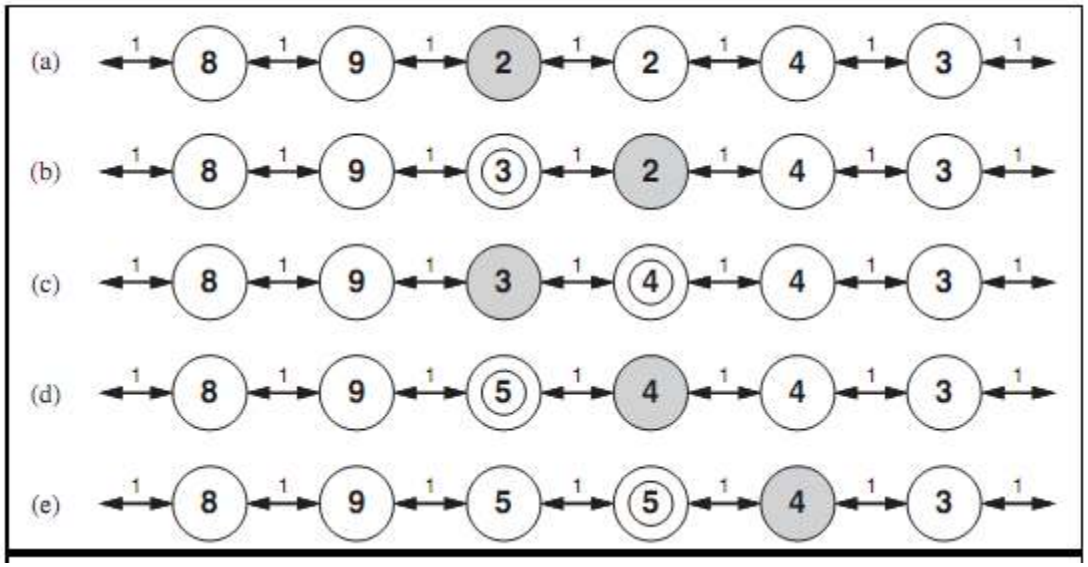


Figure 2.27 Five iterations of LRTA * on a one-dimensional state space. Each state is labeled with H(s), the current cost estimate to reach a goal, and each link is labeled with its step cost. The shaded state marks the location of the agent, and the updated cost estimates at each iteration are circled.

```
function LRTA*-AGENT(s') returns an action
  inputs: s', a percept that identifies the current state
  persistent: result, a table, indexed by state and action, initially empty
              H, a table of cost estimates indexed by state, initially empty
              s, a, the previous state and action, initially null

  if GOAL-TEST(s') then return stop
  if s' is a new state (not in H) then H[s'] ← h(s')
  if s is not null
      result[s, a] ← s'
      H[s] ← min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[s, b], H)
  a ← an action b in ACTIONS(s') that minimizes LRTA*-COST(s', b, result[s', b], H)
  s ← s'
  return a

function LRTA*-COST(s, a, s', H) returns a cost estimate
  if s' is undefined then return h(s)
  else return c(s, a, s') + H[s']
```

Figure 2.28 LRTA*-AGENT selects an action according to the values of neighboring states, which are updated as the agent moves about the state space.

The LRTA agent is just one of a large family of online agents that one can define by specifying the action selection rule and the update rule in different ways.

Learning in online search

The initial ignorance of online search agents provides several opportunities for learning. First, the agents learn a "map" of the environment—more precisely, the outcome of each action in each state—simply by recording each of their experiences.

Second, the local search agents acquire more accurate estimates of the cost of each state by using local updating rules, as in LRTA.

For example, after it has seen that the Up action goes from (1,1) to (1,2), the agent still has no idea that the Down action goes back to (1,1) or that the Up action also goes from (2,1) to (2,2), from (2,2) to (2,3), and so on.

In general, we would like the agent to learn that Up increases the y-coordinate unless there is a wall in the way, that Down reduces it, and so on.For this to happen, we need two things.

First, we need a formal and explicitly manipulate representation for these kinds of general rules; so far, we have hidden the information inside the black box called the RESULT function. Second, we need algorithms that can construct suitable general rules from the specific observations made by the agent.

UNIT III

GAME PLAYING AND CSP

Game theory – optimal decisions in games – alpha-beta search – monte-carlo tree search – stochastic games – partially observable games Constraint satisfaction problems – constraint propagation – backtracking search for CSP – local search for CSP – structure of CSP

3.1 Game theory

Adversarial Search

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

- But, there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.
- The environment with more than one agent is termed as **multi-agent environment**, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.
- So, **Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games**.
- Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

Types of Games in AI:

	Deterministic	**Chance Moves**
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

- **Perfect information:** A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.

- **Imperfect information:** If in a game agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
- **Deterministic games:** Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
- **Non-deterministic games:** Non-deterministic are those games which have various unpredictable events and have a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

Zero-Sum Game

- Zero-sum games are adversarial search which involves pure competition.
- In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent.
- One player of the game try to maximize one single value, while other player tries to minimize it.
- Each move by one player in the game is called as ply.
- Chess and tic-tac-toe are examples of a Zero-sum game.

Formalization of the problem:

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result(s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility(s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

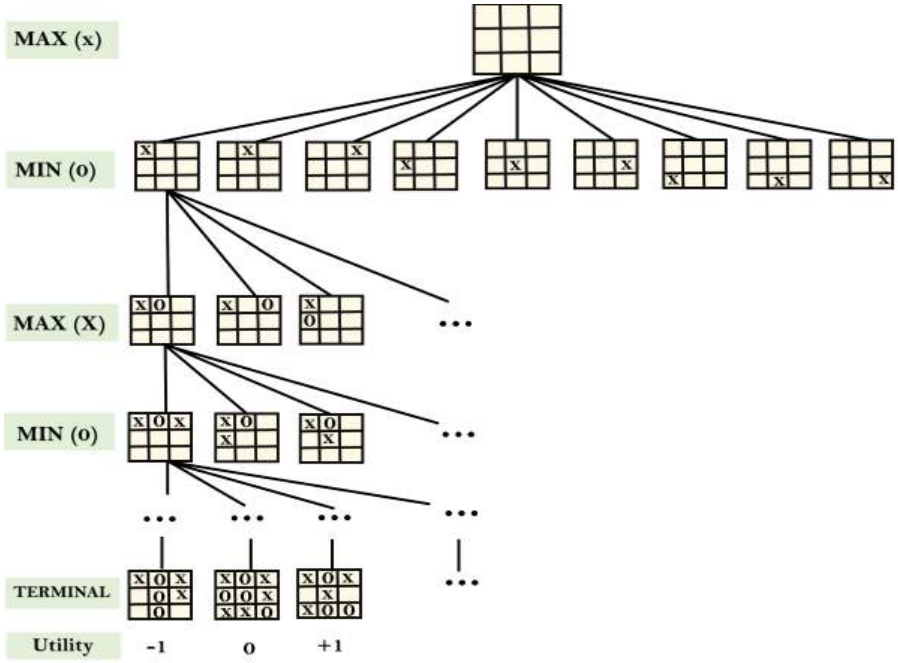
Game tree:

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.



3.2 Optimal decisions in games

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.

- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as **Ply**. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.
- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

Mini-Max Algorithm

- Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Mini-Max algorithm uses recursion to search through the game-tree.
- Min-Max algorithm is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.
- In this algorithm two players play the game, one is called MAX and other is called MIN.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value.

- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

MinMax Algorithm:

```
function MINIMAX-DECISION(state) returns an action
  return arg max_{a ∈ ACTIONS(s)} MIN-VALUE(RESULT(state, a))

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

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

Figure 5.3 An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves,

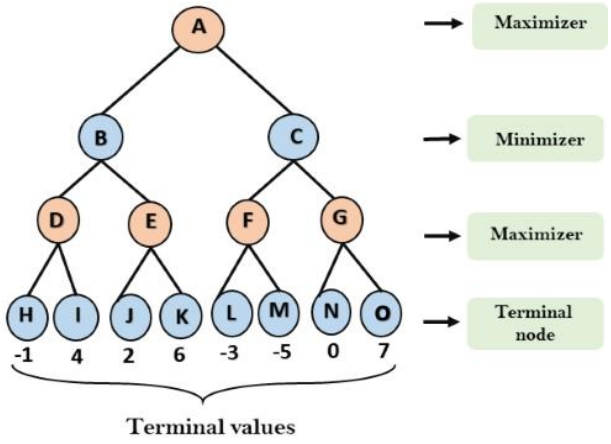
to determine the backed-up value of a state. The notation argmax a ∈ S f(a) computes the element a of set S that has the maximum value of f(a).

Working of Min-Max Algorithm:

- The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.

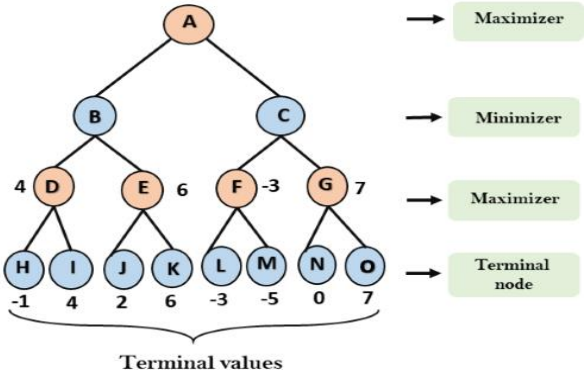
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

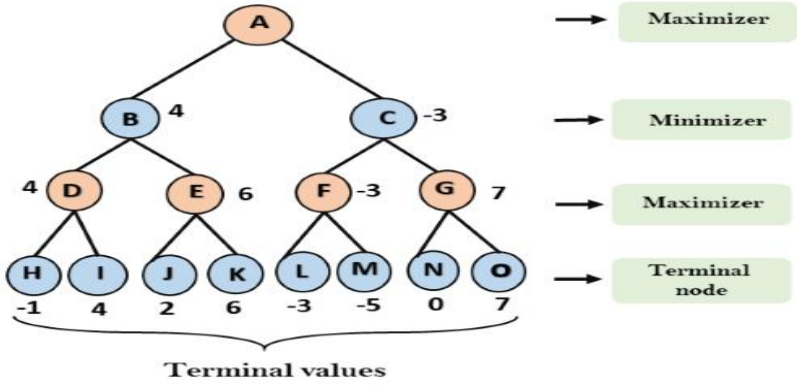
- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7



Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

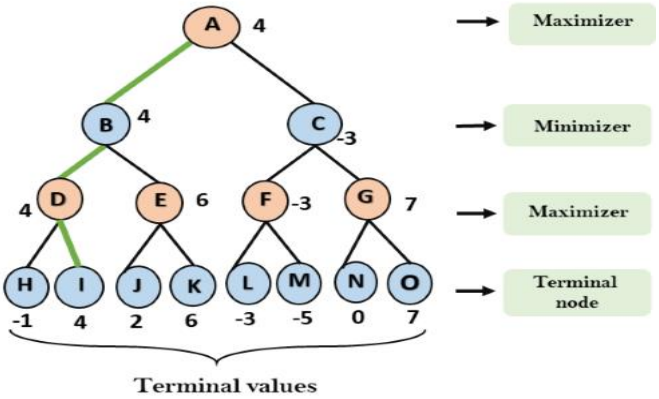
- For node B= min(4,6) = 4

- For node C= min (-3, 7) = -3



Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

- For node A max(4, -3)= 4



That was the complete workflow of the minimax two player game.

Properties of Mini-Max algorithm:

- **Complete-** Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimal-** Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity-** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is **$O(b^m)$**, where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity-** Space complexity of Mini-max algorithm is also similar to DFS which is **O(bm)**.

Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from **alpha-beta pruning** which we have discussed in the next topic.

3.3 Alpha-Beta Pruning

- Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm.
- As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent, but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called **pruning**. This involves two threshold parameter Alpha and beta for future expansion, so it is called **alpha-beta pruning**. It is also called as **Alpha-Beta Algorithm**.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.
- The two-parameter can be defined as:
    1. **Alpha:** The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is **-∞**.
    2. **Beta:** The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is **+∞**.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

The main condition which required for alpha-beta pruning is:

1. $\alpha >= \beta$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.

- We will only pass the alpha, beta values to the child nodes.

Pseudo-code for Alpha-beta Pruning:

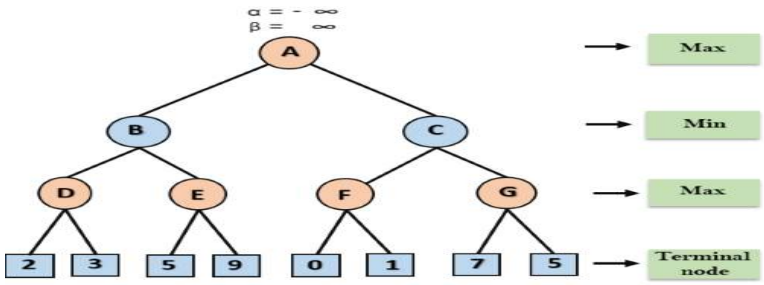
```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v

function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v

function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

Figure 5.7 The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β (and the bookkeeping to pass these parameters along).

Working of Alpha-Beta Pruning:

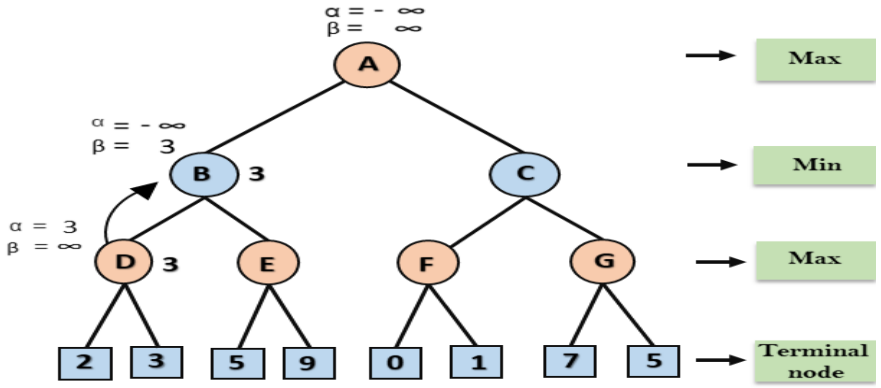
Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.



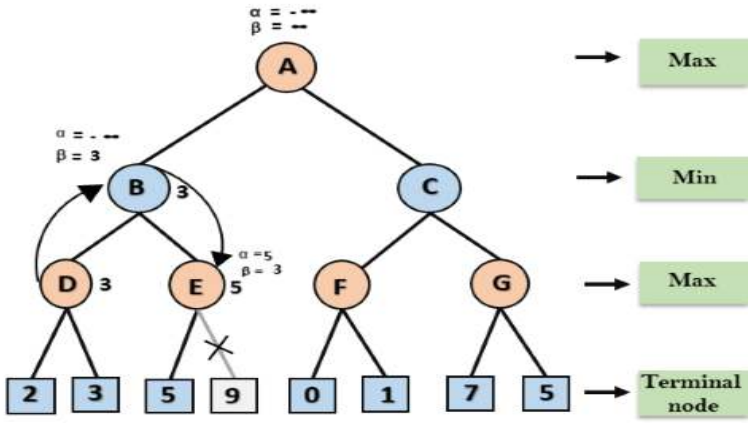
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

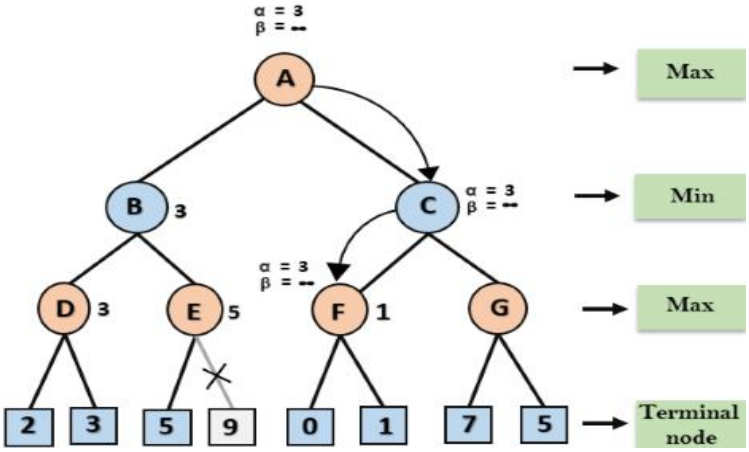
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



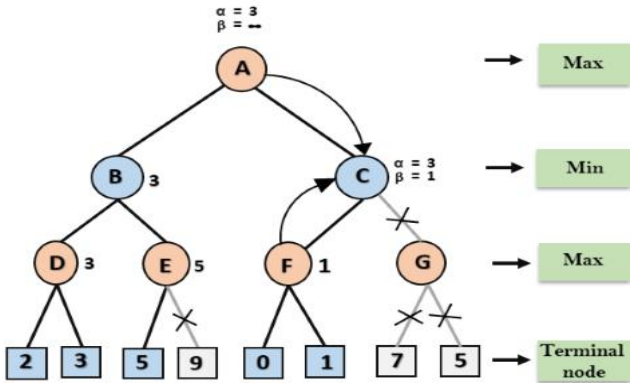
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

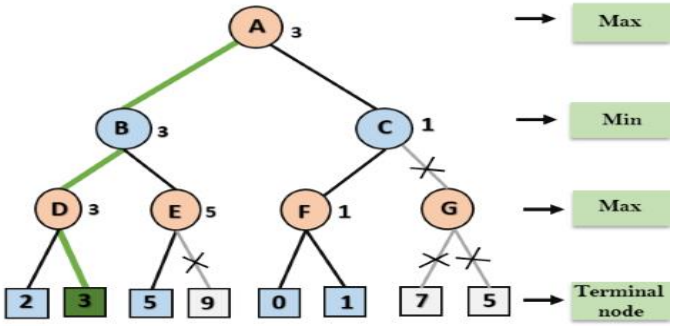
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

- **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is $O(b^m)$.
- **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.
- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.

3.4 Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning.

In tree search, there's always the possibility that the current best action is actually not the most optimal action.

In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the " ***exploration-exploitation trade-off*** ".

It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.

Exploration helps in exploring and discovering the unexplored parts of the tree, which could result in finding a more optimal path. In other words, we can say that exploration expands the tree's

breadth more than its depth. Exploration can be useful to ensure that MCTS is not overlooking any potentially better paths.

Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach and this will extend the tree's depth more than its breadth. In simple words, UCB formula applied to trees helps to balance the exploration-exploitation trade-off by periodically exploring relatively unexplored nodes of the tree and discovering potentially more optimal paths than the one it is currently exploiting.

Monte Carlo Tree Search (MCTS) algorithm:

In MCTS, nodes are the building blocks of the search tree. These nodes are formed based on the outcome of a number of simulations. The process of Monte Carlo Tree Search can be broken down into four distinct steps, viz., selection, expansion, simulation and back propagation. Each of these steps is explained in details below:

- **Selection:** In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value. The parameters are characterized by the formula that is typically used for this purpose is given below.

$$S_i = x_i + C\sqrt{\frac{\ln(t)}{n_i}}$$

- where;

  S_i = value of a node i

  x_i = empirical mean of a node i

  C = a constant

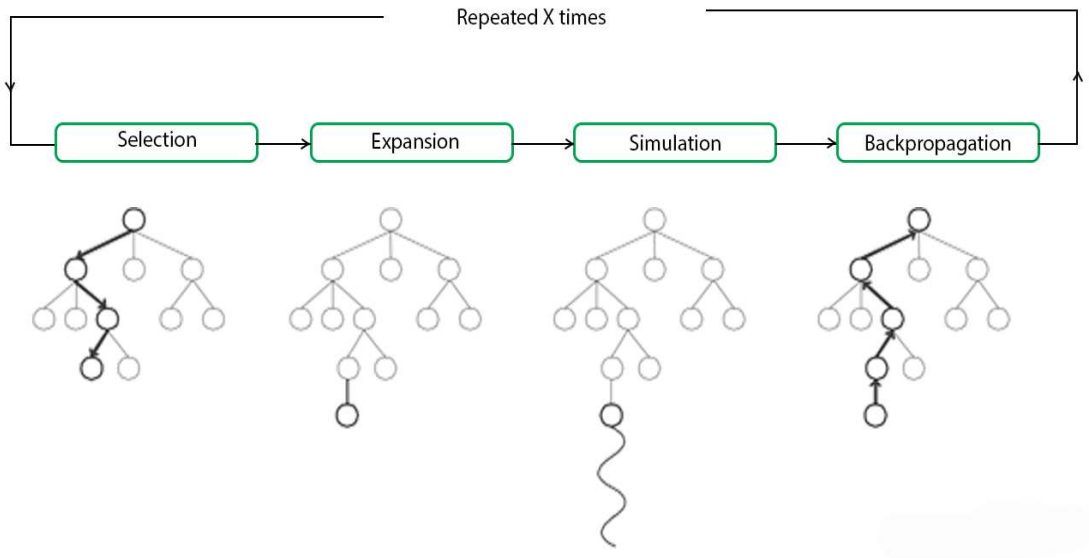
  t = total number of simulations

  When traversing a tree during the selection process, the child node that returns the greatest value from the above equation will be one that will get selected. During traversal, once a child node is found which is also a leaf node, the MCTS jumps into the expansion step.

- **Expansion:** In this process, a new child node is added to the tree to that node which was optimally reached during the selection process.

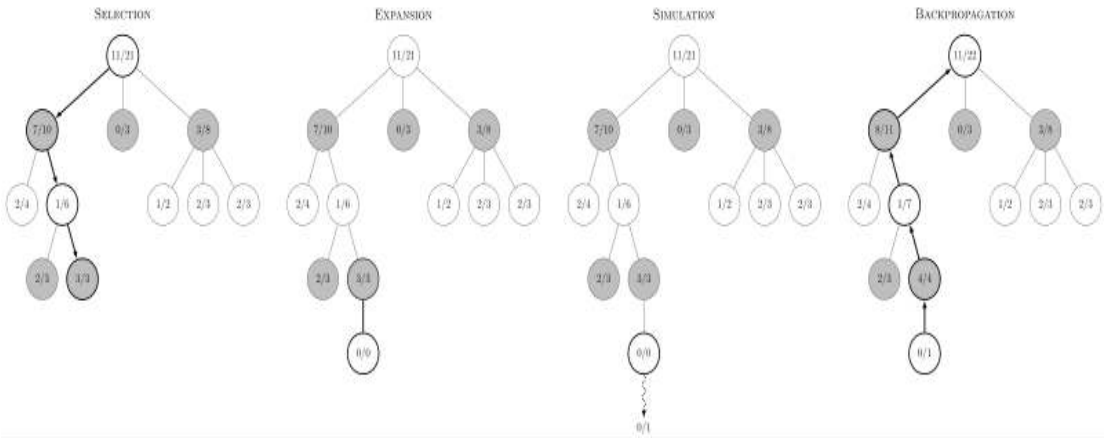
- **Simulation:** In this process, a simulation is performed by choosing moves or strategies until a result or predefined state is achieved.
- **Backpropagation:** After determining the value of the newly added node, the remaining tree must be updated. So, the backpropagation process is performed, where it backpropagates from the new node to the root node. During the process, the number of simulation stored in each node is incremented. Also, if the new node's simulation results in a win, then the number of wins is also incremented.

The above steps can be visually understood by the diagram given below:



These types of algorithms are particularly useful in turn based games where there is no element of chance in the game mechanics, such as Tic Tac Toe, Connect 4, Checkers, Chess, Go, etc.

This has recently been used by Artificial Intelligence Programs like AlphaGo, to play against the world's top Go players. But, its application is not limited to games only. It can be used in any situation which is described by state-action pairs and simulations used to forecast outcomes.



Advantages of Monte Carlo Tree Search:

1. MCTS is a simple algorithm to implement.
2. Monte Carlo Tree Search is a heuristic algorithm. MCTS can operate effectively without any knowledge in the particular domain, apart from the rules and end conditions, and can can find its own moves and learn from them by playing random playouts.
3. The MCTS can be saved in any intermediate state and that state can be used in future use cases whenever required.
4. MCTS supports asymmetric expansion of the search tree based on the circumstances in which it is operating.

Disadvantages of Monte Carlo Tree Search:

1. As the tree growth becomes rapid after a few iterations, it requires a huge amount of memory.
2. There is a bit of a reliability issue with Monte Carlo Tree Search. In certain scenarios, there might be a single branch or path, that might lead to loss against the opposition when implemented for those turn-based games. This is mainly due to the vast amount of combinations and each of the nodes might not be visited enough number of times to understand its result or outcome in the long run.
3. MCTS algorithm needs a huge number of iterations to be able to effectively decide the most efficient path. So, there is a bit of a speed issue there.

3.5 Stochastic games

In real life, many unpredictable external events can put us into unforeseen situations.

Many games mirror this unpredictability by including a random element, such as the throwing of dice. We call these stochastic games.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the legal moves.

In the backgammon position of Figure, for example, White has rolled a 6–5 and has four possible moves.

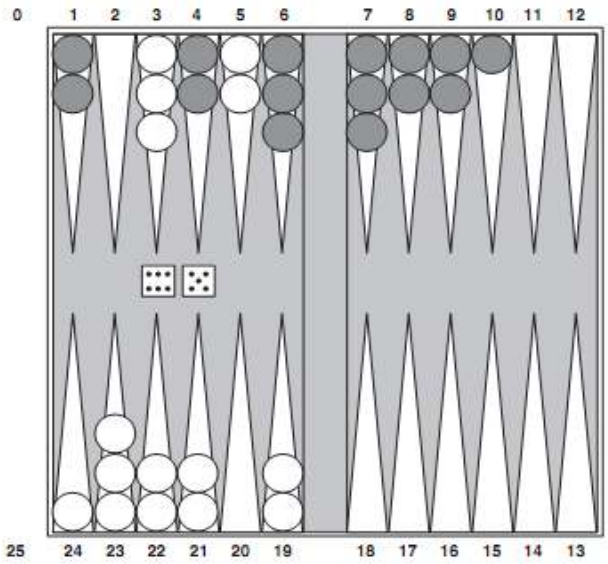


Figure A typical backgammon position.

The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0.

A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over.

White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

Although White knows what his or her own legal moves are, White does not know what Black is going to roll and thus does not know what Black's legal moves will be. That means White cannot construct a standard game tree of the sort we saw in chess and tic-tac-toe. A game tree in backgammon must include chance nodes in addition to MAX and MIN nodes.

Chance nodes are shown as circles in Figure . The branches leading from each chance node denote the possible dice rolls; each branch is labeled with the roll and its probability.

There are 36 ways to roll two dice, each equally likely; but because a 6–5 is the same as a 5–6, there are only 21 distinct rolls. The six doubles (1–1 through 6–6) each have a probability of 1/36, so we say P(1–1)=1/36. The other 15 distinct rolls each have a 1/18 probability.

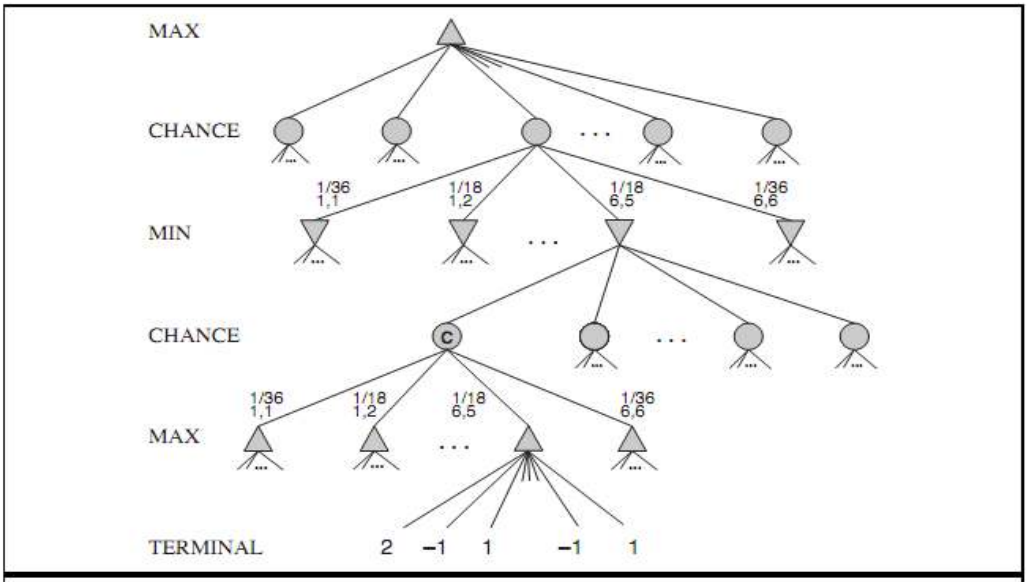


Figure Schematic game tree for a backgammon position.

The next step is to understand how to make correct decisions.

Obviously, we want to pick the move that leads to the best position.

Positions do not have definite minimax values.

We can only calculate the expected value of a position: the average over all possible outcomes of the chance nodes.

This leads us to generalize the minimax value for deterministic games to an expecti- minimax value for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before. For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

$$\text{EXPECTIMINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if TERMINAL-TEST}(s) \\ \max_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MAX} \\ \min_a \text{EXPECTIMINIMAX}(\text{RESULT}(s,a)) & \text{if PLAYER}(s) = \text{MIN} \\ \sum_r P(r)\text{EXPECTIMINIMAX}(\text{RESULT}(s,r)) & \text{if PLAYER}(s) = \text{CHANCE} \end{cases}$$

where r represents a possible dice roll (or other chance event) and RESULT(s, r) is the same state as s, with the additional fact that the result of the dice roll is r.

3.5.1 Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expecti minimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions for

chess—they just need to give higher scores to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean.

Figure shows what happens: with an evaluation function that assigns the values [1, 2,3, 4] to the leaves, move a1 is best; with values [1, 20, 30, 400], move a2 is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values! It turns out that to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position.

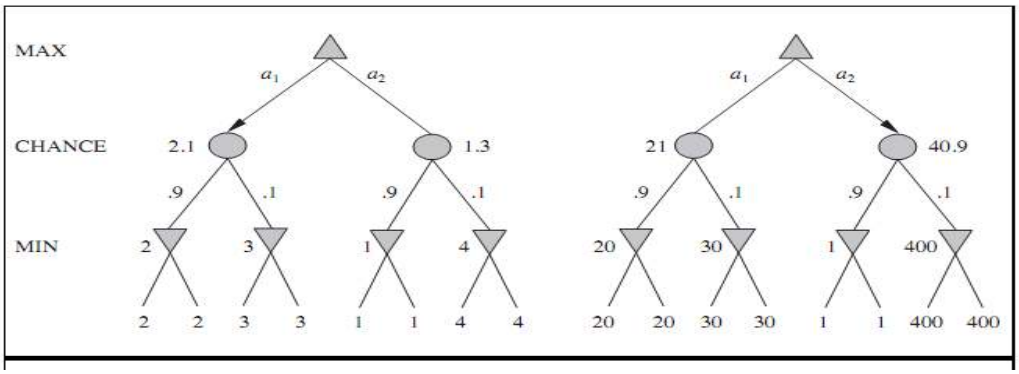


Figure An order-preserving transformation on leaf values changes the best move.

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in $O(b^m)$ time, where b is the branching factor and m is the maximum depth of the game tree. Because expectiminimax is also considering all the possible dice-roll sequences, it will take $O(b^m n^m)$,where n is the number of distinct rolls.

Even if the search depth is limited to some small depth d, the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon n is 21 and b is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage.

The analysis for MIN and MAX nodes is unchanged, but we can also prune chance nodes, using a bit of ingenuity. Consider the chance node C in Figure and what happens to its value as we examine and evaluate its children.

Is it possible to find an upper bound on the value of C before we have looked at all its children?

At first sight, it might seem impossible because the value of C is the average of its children's values, and in order to compute the average of a set of numbers, we must look at all the numbers.

But if we put bounds on the possible values of the utility function, then we can arrive at bounds for the average without looking at every number. For example, say that all utility values are between -2 and +2; then the value of leaf nodes is bounded, and in turn we can place an upper bound on the value of a chance node without looking at all its children.

An alternative is to do Monte Carlo simulation to evaluate a position.

Start with an alpha–beta (or other) search algorithm.

From a start position, have the algorithm play thousands of games against itself, using random dice rolls.

In the case of backgammon, the resulting win percentage has been shown to be a good approximation of the value of the position, even if the algorithm has an imperfect heuristic and is searching only a few plies.

For games with dice, this type of simulation is called a rollout.

3.6 Partially observable games

Chess has often been described as war in miniature, but it lacks at least one major characteristic of real wars, namely, partial observability. In the "fog of war," the existence and disposition of enemy units is often unknown until revealed by direct contact. As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy. Partially observable games share these characteristics and are thus qualitatively different from the games.

3.6.1 Kriegspiel: Partially observable chess

In deterministic partially observable games, uncertainty about the state of the board arises entirely from lack of access to the choices made by the opponent.

This class includes children's games such as Battleships (where each player's ships are placed in locations hidden from the opponent but do not move) and Stratego (where piece locations are known but piece types are hidden).

We will examine the game of Kriegspiel, a partially observable variant of chess in which pieces can move but are completely invisible to the opponent.

The rules of Kriegspiel are as follows:

- White and Black each see a board containing only their own pieces.

- A referee, who can see all the pieces, adjudicates the game and periodically makes announcements that are heard by both players.
- On his turn, White proposes to the referee any move that would be legal if there were no black pieces.
- If the move is in fact not legal (because of the black pieces), the referee announces "illegal."
- In this case, White may keep proposing moves until a legal one is found—and learns more about the location of Black's pieces in the process.
- Once a legal move is proposed, the referee announces one or more of the following: "Capture on square X" if there is a capture, and "Check by D" if the black king is in check, where D is the direction of the check, and can be one of "Knight," "Rank," "File," "Long diagonal," or "Short diagonal."
- If Black is checkmated or stalemated, the referee says so; otherwise, it is Black's turn to move.
- Kriegspiel may seem terrifyingly impossible, but humans manage it quite well and computer programs are beginning to catch up.

The set of all logically possible board states given the complete history of percepts to date.

Initially, White's belief state is a singleton because Black's pieces haven't moved yet.

After White makes a move and Black responds, White's belief state contains 20 positions because Black has 20 replies to any White move.

Keeping track of the belief state as the game progresses is exactly the problem of state estimation, for which the update step is given in Equation.

We can map Kriegspiel state estimation directly onto the partially observable, nondeterministic framework of Section 4.4 if we consider the opponent as the source of nondeterminism; that is, the RESULTS of White's move are composed from the (predictable) outcome of White's own move and the unpredictable outcome given by Black's reply.

Given a current belief state, White may ask, "Can I win the game?" For a partially observable game, the notion of a strategy is altered; instead of specifying a move to make for each possible move the opponent might make, we need a move for every possible percept sequence that might be received.

For Kriegspiel, a winning strategy, or guaranteed checkmate, is one that, for each possible percept sequence, leads to an actual checkmate for every possible board state in the current belief state, regardless of how the opponent moves.

With this definition, the opponent's belief state is irrelevant—the strategy has to work even if the opponent can see all the pieces.

This greatly simplifies the computation. Figure shows part of a guaranteed checkmate for the KRK (king and rook against king) endgame.

In this case, Black has just one piece (the king), so a belief state for White can be shown in a single board by marking each possible position of the Black king.

The general AND-OR search algorithm can be applied to the belief-state space to find guaranteed checkmates

The incremental belief-state algorithm mentioned in that section often finds midgame checkmates up to depth 9—probably well beyond the abilities of human players.

In addition to guaranteed checkmates, Kriegspiel admits an entirely new concept that makes no sense in fully observable games: probabilistic checkmate.

Such checkmates are still required to work in every board state in the belief state; they are probabilistic with respect to randomization of the winning player's moves.

To get the basic idea, consider the problem of finding a lone black king using just the white king. Simply by moving randomly, the white king will eventually bump into the black king even if the latter tries to avoid this fate, since Black cannot keep guessing the right evasive moves indefinitely. In the terminology of probability theory, detection occurs with probability

1. The KBNK endgame—king, bishop

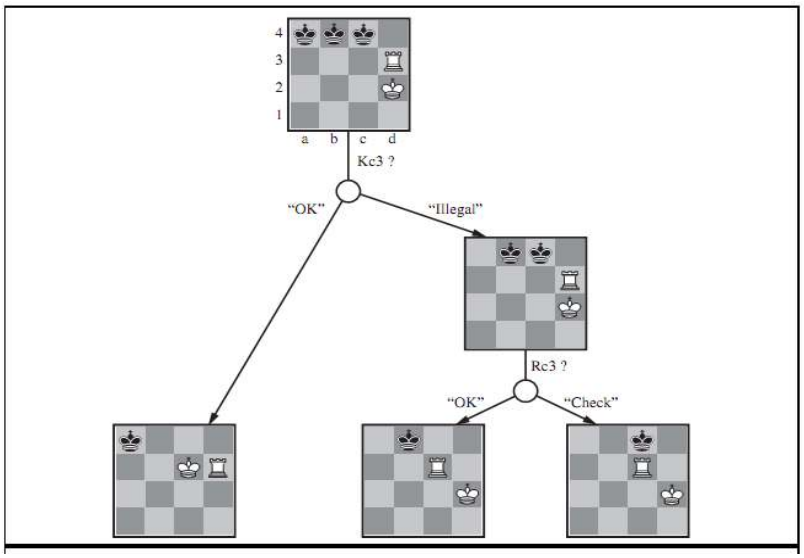


Figure Part of a guaranteed checkmate in the KRK endgame, shown on a reduced board. In the initial belief state, Black's king is in one of three possible locations. By a combination of probing moves, the strategy narrows this down to one. Completion of the checkmate is left as an exercise.
and knight against king—is won in this sense; White presents Black with an infinite random sequence of choices, for one of which Black will guess incorrectly and reveal his position, leading to checkmate.

The KBBK endgame, on the other hand, is won with probability 1 - ε.

White can force a win only by leaving one of his bishops unprotected for one move.

If Black happens to be in the right place and captures the bishop (a move that would lose if the bishops are protected), the game is drawn. White can choose to make the risky move at some randomly chosen point in the middle of a very long sequence, thus reducing ε to an arbitrarily small constant, but cannot reduce ε to zero.

It is quite rare that a guaranteed or probabilistic checkmate can be found within any reasonable depth, except in the endgame.

Sometimes a checkmate strategy works for some of the board states in the current belief state but not others.

Trying such a strategy may succeed, leading to an accidental checkmate—accidental in the sense that White could not know that it would be checkmate—if Black's pieces happen to be in the right places.

This idea leads naturally to the question of how likely it is that a given strategy will win, which leads in turn to the question of how likely it is that each board state in the current belief state is the true board state.
One's first inclination might be to propose that all board states in the current belief state are equally likely—but this can't be right.

Consider, for example, White's belief state after Black's first move of the game.

By definition, Black must have played an optimal move, so all board states resulting from suboptimal moves ought to be assigned zero probability.

This argument is not quite right either, because each player's goal is not just to move pieces to the right squares but also to minimize the information that the opponent has about their location.

Playing any predictable "optimal" strategy provides the opponent with information. Hence, optimal play in partially observable games requires a willingness to play somewhat randomly.

3.6.2 Card games

Card games provide many examples of stochastic partial observability, where the missing information is generated randomly.

For example, in many games, cards are dealt randomly at the beginning of the game, with each player receiving a hand that is not visible to the other players. Such games include bridge, whist, hearts, and some forms of poker.

At first sight, it might seem that these card games are just like dice games: the cards are dealt randomly and determine the moves available to each player, but all the "dice" are rolled at the beginning!

Even though this analogy turns out to be incorrect, it suggests an effective algorithm: consider all possible deals of the invisible cards; solve each one as if it were a fully observable game; and then choose the move that has the best outcome averaged over all the deals.

Suppose that each deal s occurs with probability P(s); then the move we want is

$$\underset{a}{\operatorname{argmax}} \sum_{s} P(s)\, \textsc{Minimax}(\textsc{Result}(s,a)) .$$

Here, we run exact MINIMAX if computationally feasible; otherwise, we run H-MINIMAX.

Now, in most card games, the number of possible deals is rather large. For example, in bridge play, each player sees just two of the four hands; there are two unseen hands of 13 cards each, so the number of deals is 26^{13}=10, 400, 600.

Solving even one deal is quite difficult, so solving ten million is out of the question. Instead, we resort to a Monte Carlo approximation: instead of adding up all the deals, we take a random sample of N deals, where the probability of deal s appearing in the sample is proportional to P(s):

$$\underset{a}{\operatorname{argmax}} \frac{1}{N} \sum_{i=1}^{N} \textsc{Minimax}(\textsc{Result}(s_i,a)) .$$

As N grows large, the sum over the random sample tends to the exact value, but even for fairly small N—say, 100 to 1,000—the method gives a good approximation.

It can also be applied to deterministic games such as Kriegspiel, given some reasonable estimate of P(s).

For games like whist and hearts, where there is no bidding or betting phase before play commences, each deal will be equally likely and so the values of P(s) are all equal.

For bridge, play is preceded by a bidding phase in which each team indicates how many tricks it expects to win.

Since players bid based on the cards they hold, the other players learn more about the probability of each deal.

Consider the following story:

Day 1: Road A leads to a heap of gold; Road B leads to a fork. Take the left fork and you'll find a bigger heap of gold, but take the right fork and you'll be run over by a bus.

Day 2: Road A leads to a heap of gold; Road B leads to a fork. Take the right fork and you'll find a bigger heap of gold, but take the left fork and you'll be run over by a bus.

Day 3: Road A leads to a heap of gold; Road B leads to a fork. One branch of the fork leads to a bigger heap of gold, but take the wrong fork and you'll be hit by a bus.

Unfortunately, you don't know which fork is which.

Averaging over clairvoyance leads to the following reasoning: on Day 1, B is the right choice; on Day 2, B is the right choice; on Day 3, the situation is the same as either Day 1 or Day 2, so B must still be the right choice.

Now we can see how averaging over clairvoyance fails: it does not consider the belief state that the agent will be in after acting.

A belief state of total ignorance is not desirable, especially when one possibility is certain death. Because it assumes that every future state will automatically be one of perfect knowledge, the approach never selects actions that gather information; nor will it choose actions that hide information from the opponent or provide information to a partner because it assumes that they already know the information; and it will never bluff in poker, because it assumes the opponent can see its cards.

3.7. CONSTRAINT SATISFACTION PROBLEMS(CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables ,** $X_1,X_2,\ldots X_n$, and a set of constraints $C_1,C_2,\ldots,C_m$. Each variable X_i has a nonempty **domain** D,of possible **values**. Each constraint C_i involves some subset of 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,$\{X_i = v_i,X_j = v_j,\ldots\}$. 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 **objective function**.

Example for Constraint Satisfaction Problem :

Given the task of coloring each region either red,green,or blue in such a way that the neighboring regions 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 more succinctly as the inequality WA not = NT,provided the constraint satisfaction algorithm has some way to evaluate such expressions.) There are many possible solutions such as { WA = red, NT = green,Q = red, NSW = green, V = red ,SA = blue,T = red}. The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.

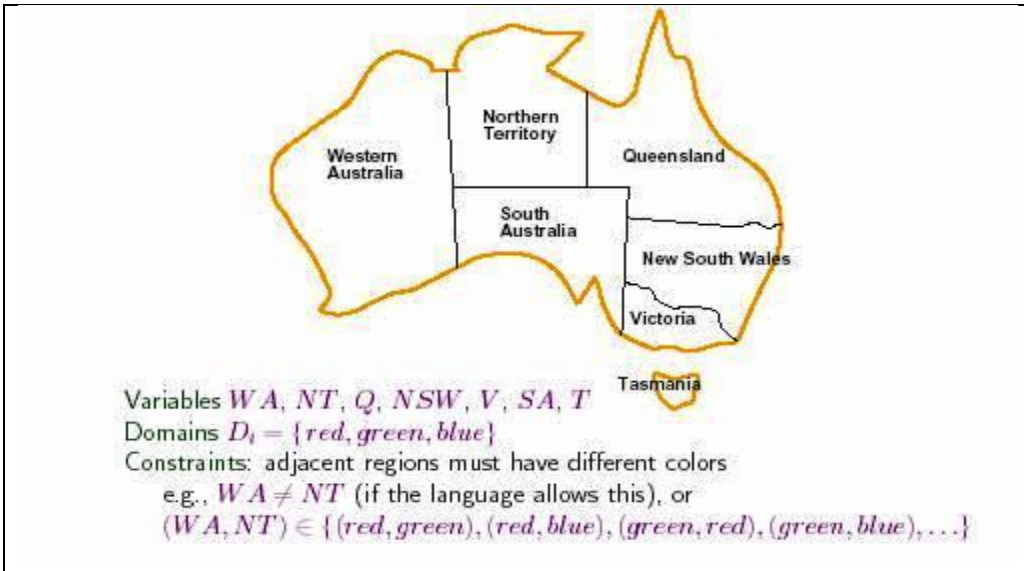


Figure 2.15 (a) Principle states and territories of Australia. Coloring this map can be viewed as ac onstraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

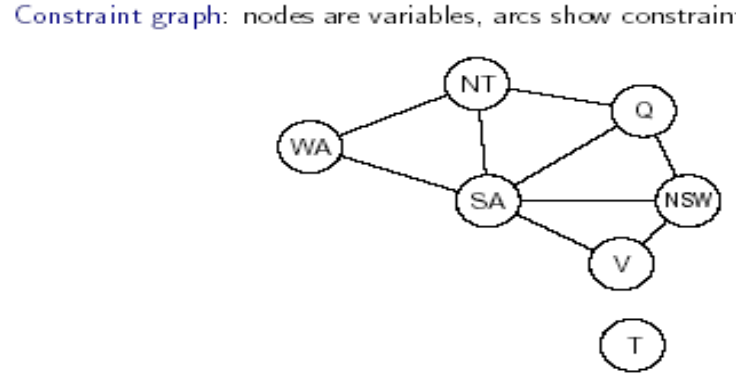


Figure 2.15 (b) The map coloring problem represented as a constraint graph.

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.

Every solution must be a complete assignment and therefore appears at depth n if there are n variables.

Depth first search algorithms are popular for CSPs

Varieties of CSPs

(i) Discrete variables

Finite domains

The simplest kind of CSP involves variables that are **discrete** and have **finite domains.** Map coloring problems are of this kind. The 8-queens problem can also be viewed as finite-domain CSP,where the variables Q_1,Q_2,…..Q_8 are the positions each queen in columns 1,….8 and each variable has the domain {1,2,3,4,5,6,7,8}. If the maximum domain size of any variable in a CSP is d,then the number of possible complete assignments is $O(d^n)$ – that is,exponential in the number of variables. Finite domain CSPs include **Boolean CSPs**,whose variables can be either *true* or *false*.

Infinite domains

Discrete variables can also have **infinite domains** – for example,the set of integers or the set of strings. With infinite domains,it is no longer possible to describe constraints by enumerating all allowed combination of values. Instead a constraint language of algebric inequalities such as $Startjob_1 + 5 <= Startjob_3$.

(ii) CSPs with continuous domains

CSPs with continuous domains are very common in real world. For example ,in operation research field,the scheduling of experiments on the Hubble Telescope requires very precise timing of observations; the start and finish of each observation and maneuver are continuous-valued variables that must obey a variety of astronomical,precedence and power constraints. The best known category of continuous-domain CSPs is that of **linear programming** problems,where the constraints must be linear inequalities forming a *convex* region. Linear programming problems can be solved in time polynomial in the number of variables.

Varieties of constraints :

(i) unary constraints involve a single variable. Example : SA # green

(ii) Binary constraints involve paris of variables Example : SA # WA

(iii) Higher order constraints involve 3 or more variables.

Example : cryptarithmetic puzzles.

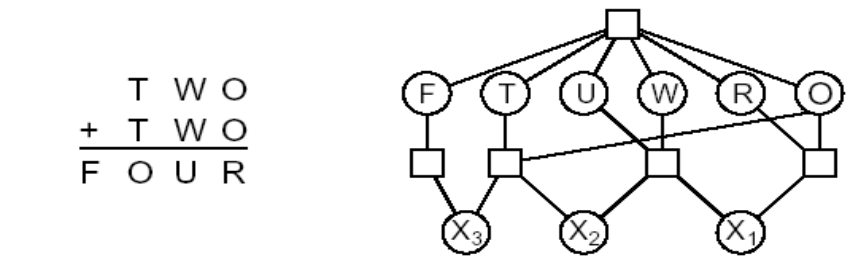


Figure Cryptarithmetic problem. Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed. (b) The constraint hypergraph for the cryptarithmetic problem,showint the *Alldiff* constraint as well as the column addition constraints. Each constraint is a square box connected to the variables it contains.

3.9 Backtracking Search for CSPs

The term **backtracking search** is used for depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. The algorithm is shown in figure 2.17.

```
function BACKTRACKING-SEARCH(csp) returns solution/failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns soln/failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment given CONSTRAINTS[csp] then
             add {var = value} to assignment
             result ← RECURSIVE-BACKTRACKING(assignment, csp)
             if result ≠ failure then return result
             remove {var = value} from assignment
   return failure
```

Figure A simple backtracking algorithm for constraint satisfaction problem. The algorithm is modeled on the recursive depth-first search

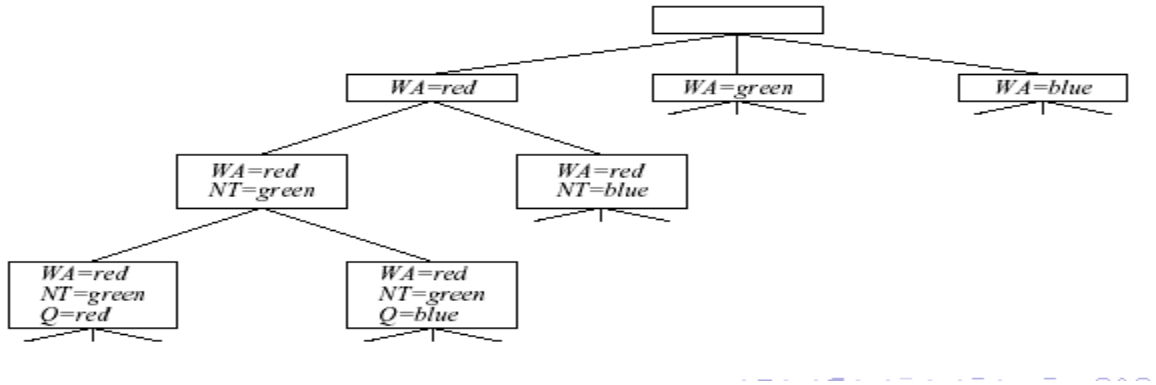


Figure Part of search tree generated by simple backtracking for the map coloring problem.

Propagating information through constraints

So far our search algorithm considers the constraints on a variable only at the time that the variable is chosen by SELECT-UNASSIGNED-VARIABLE. But by looking at some of the constraints earlier in the search, or even before the search has started, we can drastically reduce the search space.

Forward checking

One way to make better use of constraints during search is called **forward checking**. Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X. Figure 5.6 shows the progress of a map-coloring search with forward checking.

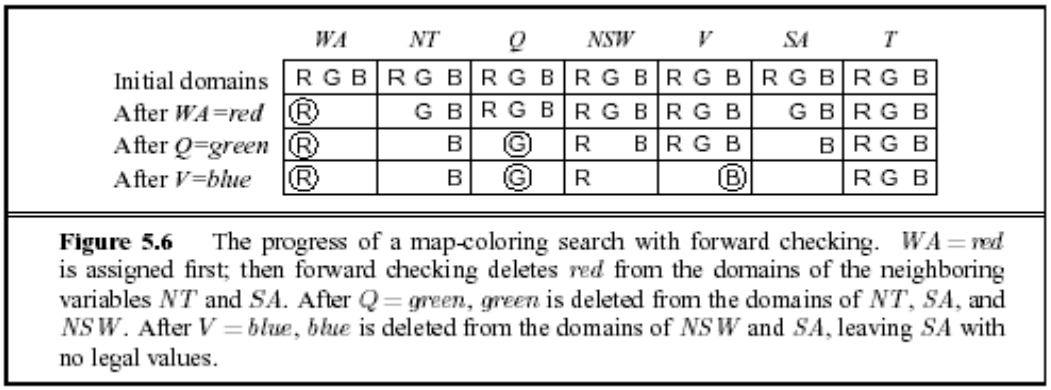


Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes red from the domains of the neighboring variables NT and SA. After $Q = green$, $green$ is deleted from the domains of NT, SA, and NSW. After $V = blue$, $blue$ is deleted from the domains of NSW and SA, leaving SA with no legal values.

3.8 Constraint propagation

Constraint propagation is the general term for propagating the implications of a constraint on one variable onto other variables.

Constraint Propagation

In local state-spaces, the choice is only one, i.e., to search for a solution. But in CSP, we have two choices either:

- We can search for a solution or
- We can perform a special type of inference called **constraint propagation**.

Constraint propagation is a special type of inference which helps in reducing the legal number of values for the variables. The idea behind constraint propagation is **local consistency.**

In local consistency, variables are treated as **nodes**, and each binary constraint is treated as an **arc** in the given problem. **There are following local consistencies which are discussed below:**

- **Node Consistency:** A single variable is said to be node consistent if all the values in the variable's domain satisfy the unary constraints on the variables.
- **Arc Consistency:** A variable is arc consistent if every value in its domain satisfies the binary constraints of the variables.
- **Path Consistency:** When the evaluation of a set of two variable with respect to a third variable can be extended over another variable, satisfying all the binary constraints. It is similar to arc consistency.
- **k-consistency:** This type of consistency is used to define the notion of stronger forms of propagation. Here, we examine the k-consistency of the variables.

Arc Consistency

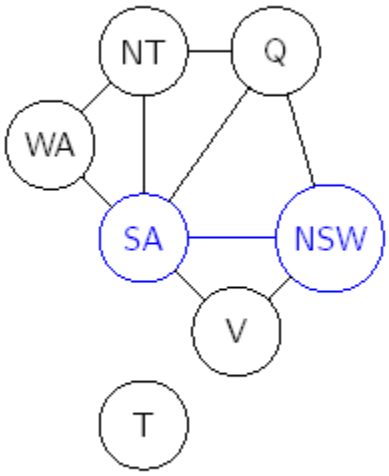


Figure: Australian Territories

- One method of constraint propagation is to enforce **arc consistency**
  - Stronger than forward checking
  - Fast
- *Arc* refers to a *directed* arc in the constraint graph
- Consider two nodes in the constraint graph (e.g., *SA* and *NSW*)
  - An arc is **consistent** if
  - For every value x of *SA*
  - There is some value y of *NSW* that is consistent with x
- Examine arcs for consistency in *both* directions

k-Consistency

- Can define stronger forms of consistency

k-Consistency

A CSP is *k*-**consistent** if, for any consistent assignment to $k-1$ variables, there is a consistent assignment for the *k*-th variable

- **1-consistency** (**node consistency**)
  - Each variable by itself is consistent (has a non-empty domain)
- **2-consistency** (**arc consistency**)
- **3-consistency** (**path consistency**)
  - Any pair of adjacent variables can be extended to a third

3.10 Local Search for CSPs

- Local search algorithms good for many CSPs
- Use complete-state formulation
  - Value assigned to every variable
  - Successor function changes one value at a time
- Have already seen this
  - Hill climbing for 8-queens problem (AIMA § 4.3)
- Choose values using **min-conflicts** heuristic
  - Value that results in the minimum number of conflicts with other variables

```
function MIN-CONFLICTS(csp, max_steps) returns a solution or failure
  inputs: csp, a constraint satisfaction problem
          max_steps, the number of steps allowed before giving up

  current ← an initial complete assignment for csp
  for i = 1 to max_steps do
    if current is a solution for csp then return current
    var ← a randomly chosen conflicted variable from csp.VARIABLES
    value ← the value v for var that minimizes CONFLICTS(var, v, current, csp)
    set var = value in current
  return failure
```

Figure 6.8 The MIN-CONFLICTS algorithm for solving CSPs by local search. The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. The CONFLICTS function counts the number of constraints violated by a particular value, given the rest of the current assignment.

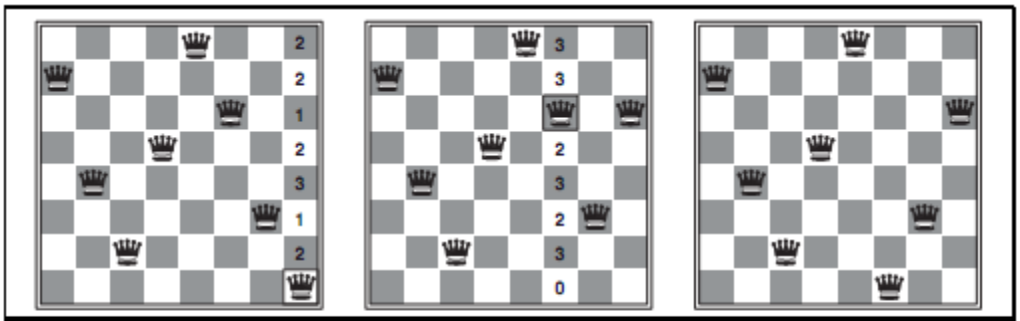


Figure 6.9 A two-step solution using min-conflicts for an 8-queens problem. At each stage, a queen is chosen for reassignment in its column. The number of conflicts (in this case, the number of attacking queens) is shown in each square. The algorithm moves the queen to the min-conflicts square, breaking ties randomly.

Local search algorithms turn out to be effective in solving many CSPs. They use a complete-state formulation: the initial state assigns a value to every variable, and the search changes the value of one variable at a time.

For example, in the 8-queens problem, the initial state might be a random configuration of 8 queens in 8 columns, and each step moves a single queen to a new position in its column.

Typically, the initial guess violates several constraints. The point of local search is to eliminate the violated constraints. In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables—the min-conflicts heuristic.

Min-conflicts is surprisingly effective for many CSPs.

Amazingly, on the n-queens problem, if you don't count the initial placement of queens, the run time of min-conflicts is roughly independent of problem size.

It solves even the million-queens problem in an average of 50 steps (after the initial assignment).

Roughly speaking, n-queens is easy for local search because solutions are densely distributed throughout the state space. Min-conflicts also works well for hard problems. For example, it has been used to schedule observations for the Hubble Space Telescope, reducing the time taken to schedule a week of observations from three weeks (!) to around 10 minutes.

Another technique, called constraint weighting, can help concentrate the search on the important constraints. Each constraint is given a numeric weight, W, initially all 1.

At each step of the search, the algorithm chooses a variable/value pair to change that will result in the lowest total weight of all violated constraints.

The weights are then adjusted by incrementing the weight of each constraint that is violated by the current assignment. This has two benefits: it adds topography to plateaux, making sure that it is possible to improve from the current state, and it also, over time, adds weight to the constraints that are proving difficult to solve.

Another advantage of local search is that it can be used in an online setting when the problem changes. This is particularly important in scheduling problems.

A week's airline schedule may involve thousands of flights and tens of thousands of personnel assignments, but bad weather at one airport can render the schedule infeasible.

Repair the schedule with a minimum number of changes. This can be easily done with a local search algorithm starting from the current schedule.

A backtracking search with the new set of constraints usually requires much more time and might find a solution with many changes from the current schedule.

3.11 The Structure of Problems

Problem Structure

- Consider ways in which the structure of the problem's constraint graph can help find solutions
- Real-world problems require decomposition into subproblems

Independent Subproblems

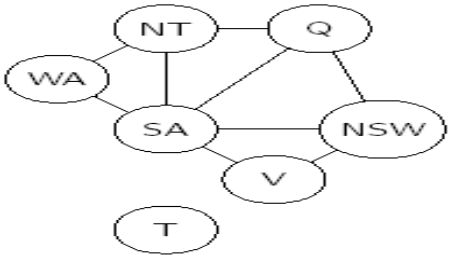


Figure: Australian Territories

- *T* is not connected
- Coloring *T* and coloring remaining nodes are **independent subproblems**
- *Any* solution for *T* combined with *any* solution for remaining nodes solves the problem
- Independent subproblems correspond to **connected components** of the constraint graph
- Sadly, such problems are rare

Tree-Structured CSPs

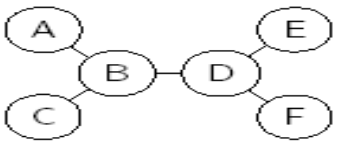


Figure: Tree-Structured CSP

- In most cases, CSPs are connected
- A simple case is when the constraint graph is a **tree**
- Can be solved in time linear in the number of variables
  - Order variables so that each parent precedes its children
  - Working "backward," apply arc consistency between child and parent
  - Working "forward," assign values consistent with parent

Figure: Linear ordering

function TREE-CSP-SOLVER(*csp*) **returns** a solution, or failure
inputs: *csp*, a CSP with components X, D, C

$n \leftarrow$ number of variables in X
assignment $\leftarrow$ an empty assignment
root $\leftarrow$ any variable in X
$X \leftarrow$ TOPOLOGICALSORT(X, *root*)
for j = n **down to** 2 **do**
  MAKE-ARC-CONSISTENT(PARENT(X_j), X_j)
  if it cannot be made consistent **then return** *failure*
for i = 1 **to** n **do**
  assignment[X_i] $\leftarrow$ any consistent value from D_i
  if there is no consistent value **then return** *failure*
return *assignment*

Figure 6.11 The TREE-CSP-SOLVER algorithm for solving tree-structured CSPs. If the CSP has a solution, we will find it in linear time; if not, we will detect a contradiction.

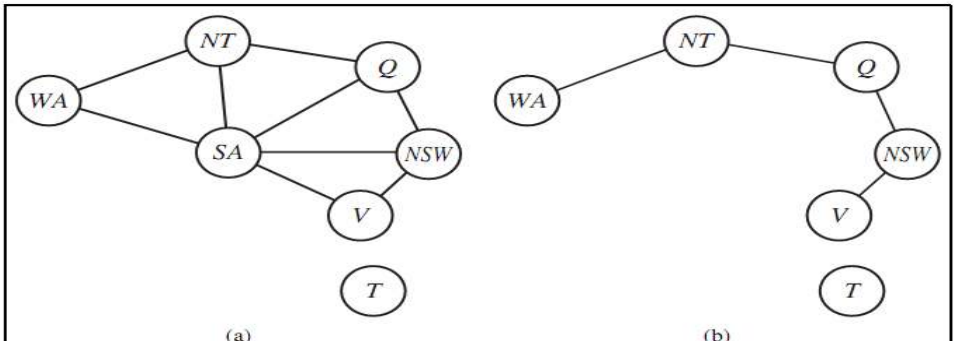


Figure 6.12 (a) The original constraint graph from Figure 6.1. (b) The constraint graph after the removal of *SA*.

Completely independent subproblems are delicious, then, but rare. Fortunately, some other graph structures are also easy to solve.

For example, a constraint graph is a tree when any two variables are connected by only one path. Any tree-structured CSP can be solved in time linear in the number of variables.

The key is a new notion of consistency, called directed arc consistency or DAC. A CSP is defined to be directed arc-consistent under an ordering of variables $X_1, X_2, ..., Xn$ if and only if every X is arc-consistent with each Xj i for j>i.

To solve a tree-structured CSP, first pick any variable to be the root of the tree, and choose an ordering of the variables such that each variable appears after its parent in the tree.

Such an ordering is called a topological sort. Figure shows a sample tree and (b) shows one possible ordering.

Any tree with n nodes has n-1 arcs, so we can make this graph directed arc-consistent in O(n) steps, each of which must compare up to d possible domain values for two variables, for a total time of $O(n^d)$.

Since each link from a parent to its child is arc consistent, we know that for any value we choose for the parent, there will be a valid value left to choose for the child.

There are two primary ways to do this, one based on removing nodes and one based on collapsing nodes together.

The first approach involves assigning values to some variables so that the remaining variables form a tree.

Consider the constraint graph for Australia, shown again in Figure 6.12(a). If we could delete South Australia, the graph would become a tree, as in (b). Fortunately, we can do this by fixing a value for SA and deleting from the domains of the other variables any values that are inconsistent with the value chosen for SA.

Now, any solution for the CSP after SA and its constraints are removed will be consistent with the value chosen for SA. Therefore, we can solve the remaining tree with the algorithm given above and thus solve the whole problem.

The general algorithm is as follows:

1. Choose a subset S of the CSP's variables such that the constraint graph becomes a tree after removal of S. S is called a cycle cutset.
2. For each possible assignment to the variables in S that satisfies all constraints on S,
   (a) remove from the domains of the remaining variables any values that are inconsistent with the assignment for S,and

(b) If the remaining CSP has a solution, return it together with the assignment for S.

If the cycle cutset has size c, then the total run time is $O(d^c c \cdot (n - c)d^2$ combinations of values for the variables in S, and for each combination we must solve a tree problem of size n - c.

If the graph is "nearly a tree," then c will be small and the savings over straight backtracking will be huge. In the worst case, however, c can be as large as (n - 2).

Finding the smallest cycle cutset is NP-hard, but several efficient approximation algorithms are known.

The second approach is based on constructing a tree decomposition of the constraint graph into a set of connected subproblems.

Each subproblem is solved independently, and the resulting solutions are then combined. Like most divide-and-conquer algorithms, this works well if no subproblem is too large. Figure shows a tree decomposition of the mapcoloring problem into five subproblems.

A tree decomposition must satisfy the following three requirements:

- Every variable in the original problem appears in at least one of the subproblems.
- If two variables are connected by a constraint in the original problem, they must appear together (along with the constraint) in at least one of the subproblems.
- If a variable appears in two subproblems in the tree, it must appear in every subproblem along the path connecting those subproblems.

The first two conditions ensure that all the variables and constraints are represented in the decomposition.

The third condition seems rather technical, but simply reflects the constraint that any given variable must have the same value in every subproblem in which it appears; the links joining subproblems in the tree enforce this constraint.

For example, SA appears in all four of the connected subproblems in Figure 6.13.
We solve each subproblem independently; if any one has no solution, we know the entire problem has no solution.

If we can solve all the subproblems, then we attempt to construct a global solution as follows. First, we view each subproblem as a "mega-variable" whose domain is the set of all solutions for the subproblem.

For example, the leftmost subproblem in Figure 6.13 is a map-coloring problem with three variables and hence has six solutions—one is {WA = red , SA = blue, NT = green}.

Then, we solve the constraints connecting the subproblems, using the efficient algorithm for trees given earlier. The constraints between subproblems simply insist that the subproblem solutions agree on their shared variables.

For example, given the solution {WA = red , SA = blue, NT = green} for the first subproblem, the only consistent solution for the next subproblem is {SA = blue, NT = green,Q= red }.

A given constraint graph admits many tree decompositions; in choosing a decomposition, the aim is to make the subproblems as small as possible.

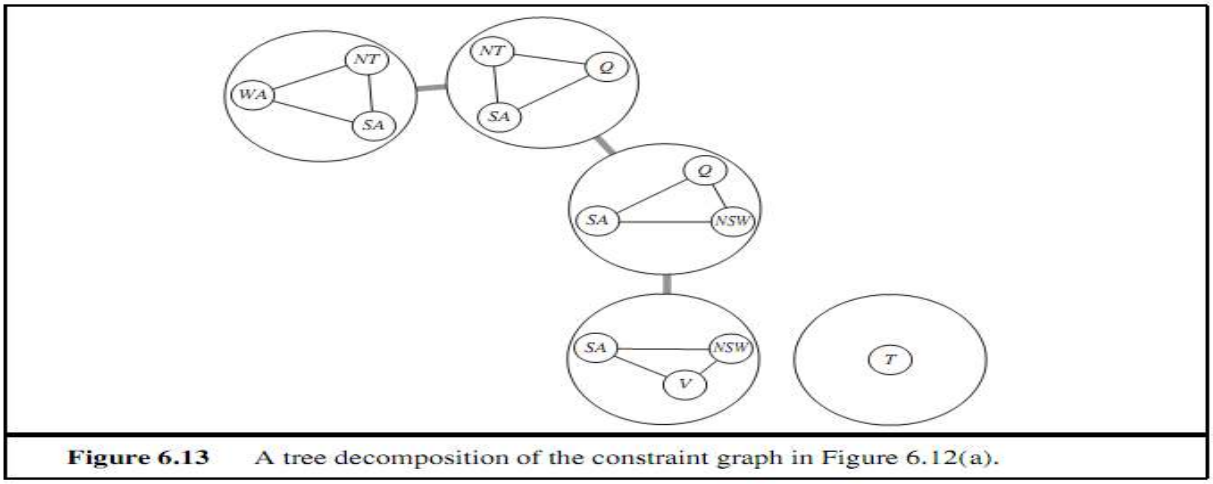


Figure 6.13 A tree decomposition of the constraint graph in Figure 6.12(a).

The tree width of a tree decomposition of a graph is one less than the size of the largest subproblem; the tree width of the graph itself is defined to be the minimum tree width among all its tree decompositions.

If a graph has tree width w and we are given the corresponding tree decomposition, then the problem can be solved in $O(n^{d\ w+1})$ time.

Hence, CSPs with constraint graphs of bounded tree width are solvable in polynomial time.

Unfortunately, finding the decomposition with minimal tree width is NP-hard, but there are heuristic methods that work well in practice.

So far, we have looked at the structure of the constraint graph. There can be important structure in the values of variables as well.

Consider the map-coloring problem with n colors. For every consistent solution, there is actually a set of n! solutions formed by permuting the color names.

For example, on the Australia map we know that WA, NT,and SA must all have different colors, but there are 3! = 6 ways to assign the three colors to these three regions. This is called value symmetry.

UNIT IV
LOGICAL AGENTS

Knowledge-based agents – propositional logic – propositional theorem proving - propositional model checking – agents based on propositional logic -First-order logic – syntax and semantics – knowledge representation and engineering – inferences in first-order logic – forward chaining – backward chaining -- resolution

LOGICAL AGENTS

Knowledge and reasoning also play a crucial role in dealing with *partially observable* environments.

A knowledge-based agent can combine general knowledge with current percepts to infer hidden aspects of the current state prior to selecting actions.

For example, a physician diagnoses a patient-that is, infers a disease state that is not directly observable prior to choosing a treatment.

Some of the knowledge that the physician uses is in the form of rules learned from textbooks and teachers, and some is in the form of patterns of association that the physician may not be able to consciously describe.

4.1 Knowledge Based agents:

The central component of a knowledge-based agent is its knowledge base, or KB.

A knowledge base is a set of sentences.

Each sentence is expressed in a language called a knowledge representation language and represents some assertion about the world.

There must be a way to add new sentences to the knowledge base and a way to query what is known.

The standard names for these tasks are TELL and ASK, respectively. Both tasks may involve inference-that is, deriving new sentences from old.

```
function KB-AGENT(percept) returns an action
  static: KB, a knowledge base
          t, a counter, initially 0, indicating time

  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
  action ← ASK(KB, MAKE-ACTION- QUERY(^))
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))
  t ← t + 1
  return action
```

A generic Knowledge based Agent

An agent takes a percept as input and returns an action.

The agent maintains a knowledge base, KB, which may initially contain some **background knowledge.**

Each time the agent program is called, it does three things.

First, it TELLS the knowledge base what it perceives.

Second, it ASKS the knowledge base what action it should perform.

The details of the representation language are hidden inside three functions that implement the interface between the sensors and actuators and the core representation and reasoning system.

MAKE-PERCEPT-SENTENCE constructs a sentence asserting that the agent perceived the given percept at the given time.

MAKE-ACTION-QUERY constructs a sentence that asks what action should be done at the current time.

Finally, MAKE-ACTION-SENTENCE constructs a sentence asserting that the chosen action was executed..

For example, an automated taxi might have the goal of delivering a passenger to Marin County and might know that it is in San Francisco and that the Golden Gate Bridge is the only link between the two locations.

Then we can expect it to cross the Golden Gate Bridge *because it knows that that will achieve its goal.* The analysis is independent of how the taxi works at the **implementation level.**

WUMPUS WORLD :

The **wumpus world** is a cave consisting of rooms connected by passageways. Lurking somewhere in the cave is the wumpus, a beast that eats anyone who enters its room.

The wumpus can be shot by an agent, but the agent has only one arrow.

Some rooms contain bottomless pits that will trap anyone who wanders into these rooms.

The only mitigating feature of living in this environment is the possibility of finding a heap of gold.

The precise definition of the task environment is given, by the PEAS description:

Performance measure: +I000 for picking up the gold, -1000 for falling into a pit or being eaten by the wumpus, -1 for each action taken and -10 for using up the arrow.

Environment: A 4 **x** 4 grid of rooms. The agent always starts in the square labeled [1,1], facing to the right. The locations of the gold and the wumpus are chosen randomly, with a uniform distribution, from the squares other than the start square

Actuators: The agent can move forward, turn left by 90°, or turn right by 90". The agent dies a miserable death if it enters a square containing a pit or a live wumpus.

Moving forward has no effect if there is a wall in front of the agent.

The action Grab can be used to pick up an object that is in the same square as the agent.

The action *Shoot* can be used to fire an arrow in a straight line in the direction the agent is facing. The arrow continues until it either hits (or hence kills) the wumpus or hits a wall. The agent only has one arrow, so only the first *Shoot* action has any effect.

Sensors: The agent has five sensors, each of which gives a single bit of information:

- In the square containing the wumpus and in the directly (not diagonally) adjacent squares the agent will perceive a stench.
- In the squares directly adjacent to a pit, the agent will perceive a breeze.
- In the square where the gold is, the agent will perceive a glitter.
- When an agent walks into a wall, it will perceive a bump.
- When the wumpus is killed, it emits a woeful scream that can be perceived anywhere in the cave.

The percepts given to the agent in the form of a list of five symbols; for example, if there is a stench and a breeze, but no glitter, bump, or scream, the agent will receive the percept *[Stench, Breeze, None, None, None].*

In most instances of the wumpus world, it is possible for the agent to retrieve the gold safely.

The agent must choose between going home empty-handed and risking death to find the gold. About 21% of the environments are utterly unfair, because the gold is in a pit or surrounded by pits.

Let us watch a knowledge-based wumpus agent exploring the environment shown in Figure below.

The agent's initial knowledge base contains the rules of the environment, as listed previously; in particular, it knows that it is in [1,1] and that [1,1] is a safe square.

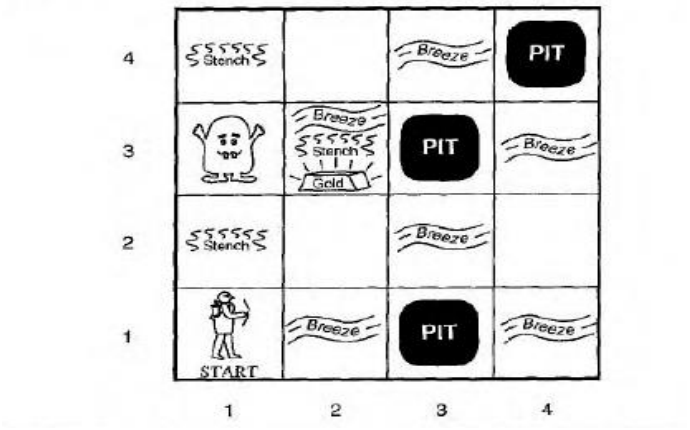


Figure 4.1: A typical wumpus world. The agent is in the bottom left corner

The first percept is *[None, None, None, None, None],* from which the agent can conclude that its neighboring squares are safe.

Figure 4.1 shows the agent's state of knowledge at this point.

We list (some of) the sentences in the knowledge base using letters such as B (breezy) and OK (safe, neither pit nor wumpus) marked in the appropriate squares. Figure 4.1, on the other hand, depicts the world itself.

They are marked with an OK to indicate this. A cautious agent will move only into a square that it knows is OK.

The agent detects a breeze in [2,1], so there must be a part in a neighboring square. The pit cannot be in [I, 11, by the rules of the game, so there must be a pit in [2,2] or [3,1] or both.

At this point, there is only one known square that is OK and has not been visited yet. So the prudent agent will turn around, go back to [I, 11, and then proceed to [1,2].

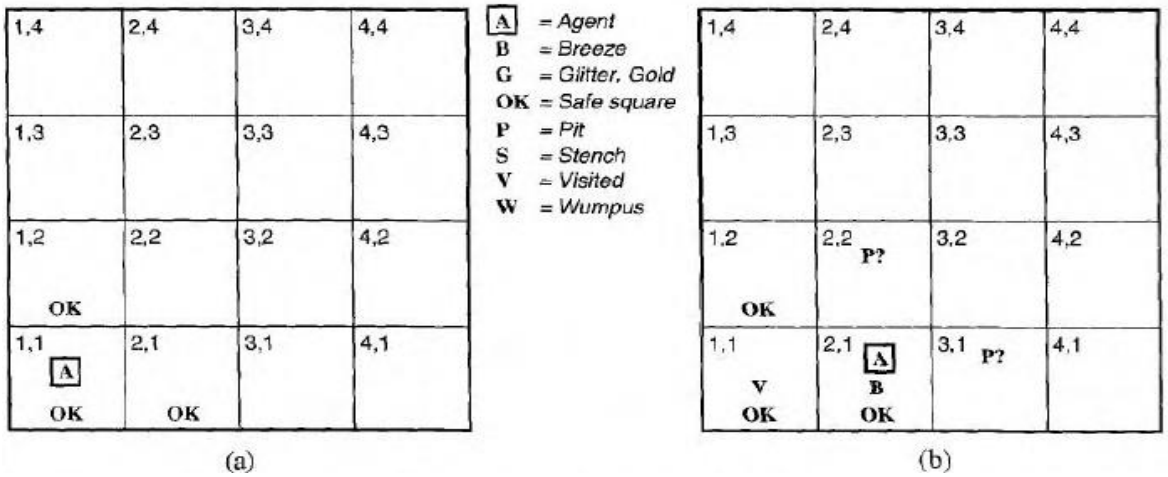


Figure 4.2: The first step taken by the agent in the wumpus world a) The initial situation first percept is *[None, None, None, None, None]*, b) After one move with percept [None, Breeze, None,None,None]

The new percept in [1,2] is *[Stench, None, None, None, None],* resulting in the state of knowledge shown in Figure 4.3 (a).

The stench in [1,2] means that there must be a wumpus nearby.

But the wumpus cannot be in [1,1], by the rules of the game, and it cannot be in [2,2].

Therefore, the agent can infer that the wumpus is in [1,3]. The notation W! indicates this.

The lack of a *Breeze* in [1,2] implies that there is no pit in [2,2].

Already inferred that there must be a pit in either [2,2] or [3,1], so this means it must be in [3,1].

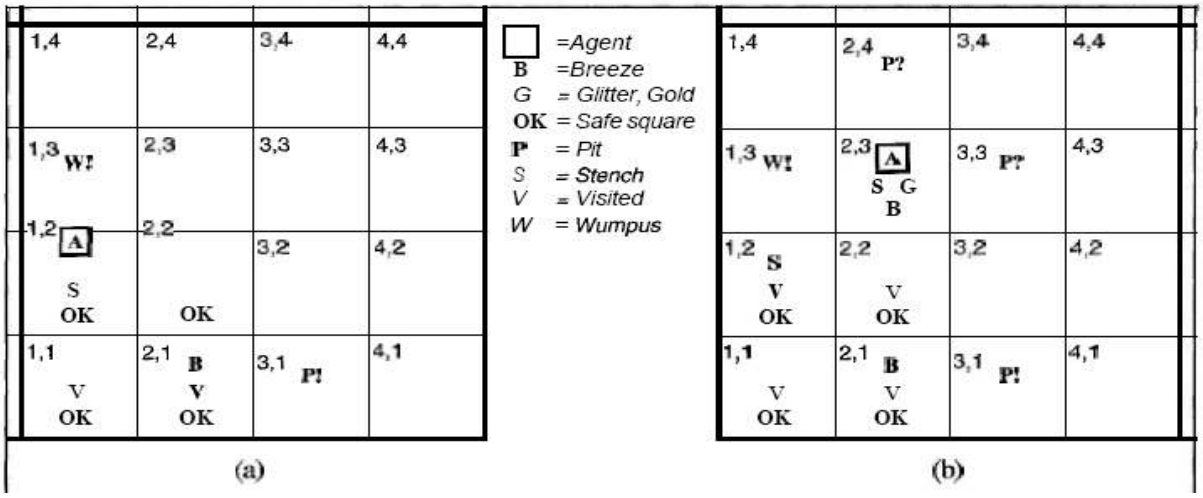


Figure: 4.3 Two later stages of the agent a) after the third move with percept *[Stench, None, None, None, None],b) after fifth move with percept [Stench, Breeze, Glitter, None, None],*

The agent has now proved to itself that there is neither a pit nor a wumpus in *[2,2],* so it is OK to move there.

We will not show the agent's state of knowledge at *[2,2];* we just assume that the agent turns and moves to *[2,3],* giving us Figure 4.3(b).

In *[2,3],* the agent detects a glitter, so it should grab the gold and thereby end the game.

Logic:

These sentences are expressed according to the **syntax** of the representation language, which specifies all the sentences that are well formed.

A logic must also define the **semantics** of the language. Loosely speaking, semantics has to do with the "meaning" of sentences. In logic the definition is more precise.

The semantics of the language defines the **truth** of each sentence with respect to each **possible world.**

For example, the usual semantics adopted for arithmetic specifies that the sentence "x +y = 4" is true in a world where x is 2 and y is 2, but false in a world where x is 1 and y is 1.

We have a notion of truth; we are ready to talk about logical reasoning. This involves the relation of logical **entailment** between sentences-the idea that a sentence *follows* from another sentence. In mathematical notation, we write as

$$\alpha \models \beta$$

to mean that the sentence entails the sentence *p.* The formal definition of entailment is this: $\alpha \models \beta$ if and only if, in every model in which *a* is true, is also true. Another way to say this is that if *a!* is true, then *p must* also be true.

We can apply the same kind of analysis to the wumpus-world reasoning example given in the preceding section. Consider the situation in Figure 4.2(b): the agent has detected nothing in [1,1] and a breeze in [2,1].

These percepts, combined with the agent's knowledge of the rules of the wumpus world, constitute the KB.

The agent is interested in whether the adjacent squares [1,2], [2,2], and [3,1] contain pits.

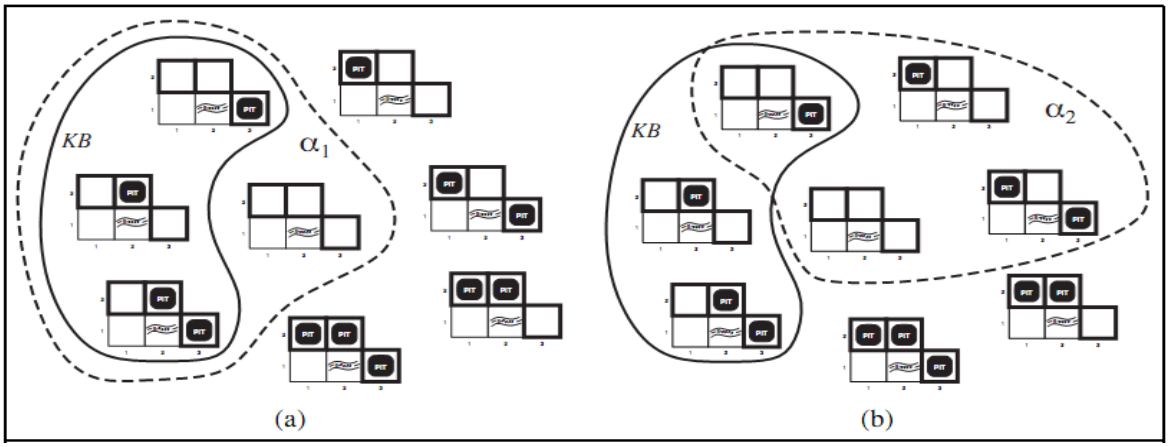


Figure 4.4 Possible models for the presence of pits in squares [1,2], [2,2], and [3,1]. The KB corresponding to the observations of nothing in [1,1] and a breeze in [2,1] is shown by the solid line. (a) Dotted line shows models of α1 (no pit in [1,2]). (b) Dotted line shows models of a α2 (no pit in [2,2]).

The KB is false in models that contradict what the agent knows-for example, the KB is false in any model in which [1,2] contains a pit, because there is no breeze in [1,1]. There are in fact just three models in which the KB is true, and these are shown as a subset of the models in Figure 4.4.

Now let us consider two possible conclusions:

a1 = "There is no pit in [1,2] ."

a2 = "There is no pit in [2,2]."

By inspection, we see the following:

in every model in which *KB* is true, *a1* is also true.

An inference algorithm that derives only entailed sentences is called **sound** or **truth-preserving.**

The property of **completeness** is also desirable: an inference algorithm is complete if it can derive any sentence that is entailed.

A reasoning process whose conclusions are guaranteed to be true in any world in which the premises are true; in particular, if *KB is true in the* real *world, then any sentence α derived from KB by a sound inference procedure is also true in the real world*

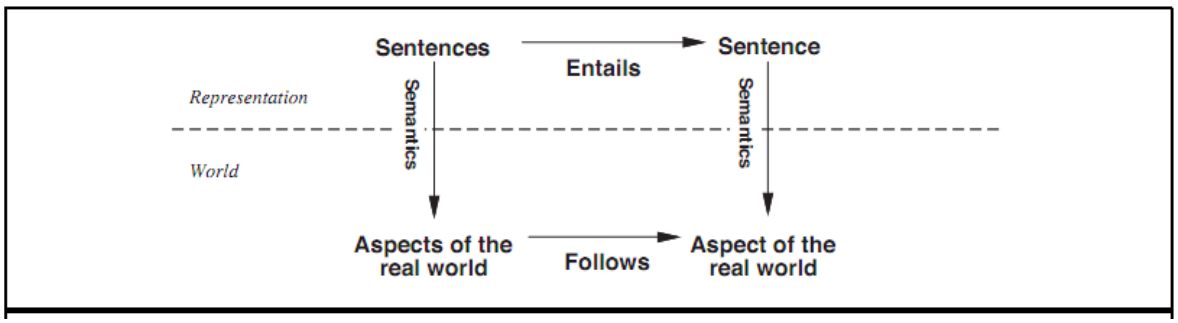


Figure 4.5 Sentences are physical configurations of the agent, and reasoning is a process of constructing new physical configurations from old ones. Logical reasoning should ensure that the new configurations represent aspects of the world that actually follow from the aspects that the old configurations represent.

4.2 Propositional Logic

A very simple logic called **propositional logic.** Then we look at **entailment-the** relation between a sentence and another sentence that follows from it-and see how this leads to a simple algorithm for logical inference.

Syntax

The **syntax** of propositional logic defines the allowable sentences. The **atomic sentences** the indivisible syntactic elements-consist of a single **proposition symbol.** Each such symbol stands for a proposition that can be true or false. We will use uppercase names forsymbols: P, Q, R, and so on.

Complex sentences are constructed from simpler sentences using **logical connectives.** There are five connectives in common use:

¬(not). A sentence such as ¬W1,3is called the **negation** of **A literal** is either an atomic sentence (a **positive literal)** or a negated atomic sentence (a **negative literal).**

∧(and). A sentence whose main connective is A, such as *W1,3∧P3,~,* is called a **conjunction;** its parts are the **conjuncts.** (The A looks like an "A" for "And.")

V (or). A sentence using V, such as *(W 1,3 ∧ P3,~)* ∨ *W2,2..* is a **disjunction** of the **disjuncts**

➔(implies). Implications are also known as **rules** or **if-then** statements.

<➔(if and only if). The sentence is a **biconditional.**

Sentence	→	*AtomicSentence* | *ComplexSentence*
AtomicSentence	→	*True* | *False* | *P* | *Q* | *R* | ...
ComplexSentence	→	(*Sentence*) | [*Sentence*]
	|	¬ *Sentence*
	|	*Sentence* ∧ *Sentence*
	|	*Sentence* ∨ *Sentence*
	|	*Sentence* ⇒ *Sentence*
	|	*Sentence* ⇔ *Sentence*
OPERATOR PRECEDENCE	:	¬, ∧, ∨, ⇒, ⇔

Figure 4.6 A BNF (Backus–Naur Form) grammar of sentences in propositional logic, along with operator precedences, from highest to lowest.

Semantics

The semantics defines the rules for determining the truth of a sentence with respect to a particular model.

In propositional logic, a model simply fixes the truth value-true or false-for every proposition symbol.

For example, if the sentences in the knowledge base make use of the proposition symbols P2,2, and P3,1, then one possible model is

{ml= P1,1=false, P2,2 =false, P3,1 = true} .

The semantics for propositional logic must specify how to compute the truth value of any sentence, given a model.

All sentences are constructed from atomic sentences and the five connectives; therefore, we need to specify how to compute the truth of atomic sentences and how to compute the truth of sentences formed with each of the five connectives.

Atomic sentences are easy:

True is true in every model and False is false in every model.

The truth value of every other proposition symbol must be specified directly in the model. For example, in the model ml given earlier, Pl,2 is false.

For complex sentences, we have rules such as

- $\neg P$ is true iff P is false in m.
- $P \wedge Q$ is true iff both P and Q are true in m.
- $P \vee Q$ is true iff either P or Q is true in m.
- $P \Rightarrow Q$ is true unless P is true and Q is false in m.
- $P \Leftrightarrow Q$ is true iff P and Q are both true or both false in m.

For any sentence s and any model m, the sentence i s is true in m if and only if *s* is false in m.

A knowledge base consists of a set of sentences.

A logical knowledge base is a conjunction of those sentences.

That is, if we start with an empty KB and do TELL(KB, Sl) . . . TELL(KB, *S,)* then we have KB = S1 A . . . A *S,*. This means that we can treat knowledge bases and sentences interchangeably.

The truth tables for "and," "or," and "not" are in close accord with our intuitions about the English words. The main point of possible confusion is that P V Q is true when P is true or *Q* is true *or both.*

There is a different connective called "exclusive or" ("xor" for short) that yields false when both disjuncts are true. There is no consensus on the symbol for exclusive or; two choices are ⅟and @.

P	Q	$\neg P$	$P \wedge Q$	$P \vee Q$	$P \Rightarrow Q$	$P \Leftrightarrow Q$
false	*false*	*true*	*false*	*false*	*true*	*true*
false	*true*	*true*	*false*	*true*	*true*	*false*
true	*false*	*false*	*false*	*true*	*false*	*false*
true	*true*	*false*	*true*	*true*	*true*	*true*

4.7 Truth table of logical connectives

A simple knowledge base

The semantics for propositional logic, we can construct a knowledge base for the wumpus world. For simplicity, we will deal only with pits; the wumpus itself is left as an exercise. First, we need to choose our vocabulary of proposition symbols. For each *i, j:*

$P_{i,j}$ be true if there is a pit in [i, *j].*

$B_{i,j}$ be true if there is a breeze in [i, *j]* .

Wi,j is true if there is a wumpus in [i, j], dead or alive.

S i,j is true if the agent perceives a stench in [i, j].

The knowledge base includes the following sentences, each one labeled for convenience:

There is no pit in [I, 11:

$R_1 : \quad \neg P_{1,1}$.

A square is breezy if and only if there is a pit in a neighboring square. This has to be stated for each square; for now, we include just the relevant squares:

$$R_2 : \quad B_{1,1} \quad \Leftrightarrow \quad (P_{1,2} \vee P_{2,1}) .$$
$$R_3 : \quad B_{2,1} \quad \Leftrightarrow \quad (P_{1,1} \vee P_{2,2} \vee P_{3,1}) .$$

The preceding sentences are true in all wumpus worlds.

$$R_4 : \quad \neg B_{1,1} .$$
$$R_5 : \quad B_{2,1} .$$

The knowledge base, then, consists of sentences R1 through R5. It can also be considered as a single sentence-the conjunction R1 ^R2 ^R3^R4 ^R5-because it asserts that all the individual sentences are true.

Inference

Our goal now is to decide whether KB |= a for some sentence a. For example, is ¬P entailed by our KB?

Our first algorithm for inference is a model-checking approach that is a direct implementation of the definition of entailment: enumerate the models, and check that a is true in every model in which KB is true.

Models are assignments of true or false to every proposition symbol.

Returning to our wumpus-world example, the relevant proposition symbols are B1,1, B2,1, P1,1, P1,2, P2,1, P2,2,and P3,1.

With seven symbols, there are 2^7 =128 possible models; in three of these, KB is true (Figure 4.8).

In those three models,¬P1,2 is true, hence there is no pit in [1,2]. On the other hand, P 2,2 is true in two of the three models and false in one, so we cannot yet tell whether there is a pit in [2,2].

Our first algorithm for inference will be a direct implementation of the definition of entailment: enumerate the models, and check that *a* is true in every model in which KB is true. For propositional logic, models are assignments of true or false to every proposition symbol.

A general algorithm for deciding entailment in propositional logic is shown in Figure 4.9.

Like the BACKTRACKING-SEARCH algorithm TT-ENTAILS? performs a recursive enumeration of a finite space of assignments to variables.

The algorithm is **sound,** because it implements directly the definition of entailment, and **complete,** because it works for any KB and *a* and always terminates-there are only finitely many models to examine.

$B_{1,1}$	$B_{2,1}$	$P_{1,1}$	$P_{1,2}$	$P_{2,1}$	$P_{2,2}$	$P_{3,1}$	R_1	R_2	R_3	R_4	R_5	KB
false	*false*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*false*	*false*
false	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*false*	*true*	*false*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
false	*true*	*false*	*false*	*false*	*false*	*false*	*true*	*true*	*false*	*true*	*true*	*false*
false	*true*	*false*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*false*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*false*	*true*	*true*	*true*	*true*	*true*	*true*	*true*	*true*
false	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*false*	*false*	*true*	*true*	*false*
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
true	*true*	*true*	*true*	*true*	*true*	*true*	*false*	*true*	*true*	*false*	*true*	*false*

Figure 4.8 A truth table constructed for the knowledge base given in the text. KB is true if R1 through R are true, which occurs in just 3 of the 128 rows (the ones underlined in the right-hand column). In all 3 rows, P1,2 is false, so there is no pit in [1,2]. On the other hand, there might (or might not) be a pit in [2,2].

```
function TT-ENTAILS?(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  symbols ← a list of the proposition symbols in KB and α
  return TT-CHECK-ALL(KB, α, symbols, { })

function TT-CHECK-ALL(KB, α, symbols, model) returns true or false
  if EMPTY?(symbols) then
      if PL-TRUE?(KB, model) then return PL-TRUE?(α, model)
      else return true // when KB is false, always return true
  else do
      P ← FIRST(symbols)
      rest ← REST(symbols)
      return (TT-CHECK-ALL(KB, α, rest, model ∪ {P = true})
              and
              TT-CHECK-ALL(KB, α, rest, model ∪ {P = false }))
```

Figure 4.9 A truth-table enumeration algorithm for deciding propositional entailment. (TT stands for truth table.) PL-TRUE? returns true if a sentence holds within a model. The variable model represents a partial model—an assignment to some of the symbols. The keyword "and" is used here as a logical operation on its two arguments, returning true or false.

$$
\begin{aligned}
(\alpha \wedge \beta) &\equiv (\beta \wedge \alpha) && \text{commutativity of } \wedge \\
(\alpha \vee \beta) &\equiv (\beta \vee \alpha) && \text{commutativity of } \vee \\
((\alpha \wedge \beta) \wedge \gamma) &\equiv (\alpha \wedge (\beta \wedge \gamma)) && \text{associativity of } \wedge \\
((\alpha \vee \beta) \vee \gamma) &\equiv (\alpha \vee (\beta \vee \gamma)) && \text{associativity of } \vee \\
\neg(\neg\alpha) &\equiv \alpha && \text{double-negation elimination} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\beta \Rightarrow \neg\alpha) && \text{contraposition} \\
(\alpha \Rightarrow \beta) &\equiv (\neg\alpha \vee \beta) && \text{implication elimination} \\
(\alpha \Leftrightarrow \beta) &\equiv ((\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)) && \text{biconditional elimination} \\
\neg(\alpha \wedge \beta) &\equiv (\neg\alpha \vee \neg\beta) && \text{De Morgan} \\
\neg(\alpha \vee \beta) &\equiv (\neg\alpha \wedge \neg\beta) && \text{De Morgan} \\
(\alpha \wedge (\beta \vee \gamma)) &\equiv ((\alpha \wedge \beta) \vee (\alpha \wedge \gamma)) && \text{distributivity of } \wedge \text{ over } \vee \\
(\alpha \vee (\beta \wedge \gamma)) &\equiv ((\alpha \vee \beta) \wedge (\alpha \vee \gamma)) && \text{distributivity of } \vee \text{ over } \wedge
\end{aligned}
$$

Figure 4.10 Standard logical equivalences. The symbols α,β,γ stand for arbitrary sentences of propositional logic

4.3 Propositional theorem proving:

The first concept is logical equivalence: two sentences α and β are logically equivalent if they are true in the same set of models.

We write this as α ≡ β. For example, we can easily show (using truth tables) that P ∧ Q and Q ∧ P are logically equivalent; other equivalences are shown in Figure 4.10. These equivalences play much the same role in logic as arithmetic identities do in ordinary mathematics.

An alternative definition of equivalence is as follows: any two sentences α and β are equivalent only if each of them entails the other:

α ≡β if and only if α ⊨ β and β ⊨ α

The second concept we will need is validity. A sentence is valid if it is true in all models. For example, the sentence P ∨ ¬P is valid. Valid sentences are also known as tautologies—they are necessarily true. Because the sentence True is true in all models, every valid sentence is logically equivalent to True. What good are valid sentences? From our definition of entailment, we can derive the deduction theorem, which was known to the ancient Greeks:

For any sentence α and β, α ⊨ β if and only if the sentence (α⇒β) is valid

Hence, we can decide if α |= β by checking that (α ⇒ β) is true in every model—which is essentially what the inference algorithm in Figure 4.9 does—or by proving that (α ⇒ β) is equivalent to True.

The final concept is satisfiability.

A sentence is satisfiable if it is true in, or satisfied by, some model.

For example, the knowledge base given earlier, (R1 ∧ R2 ∧ R3 ∧ R4 ∧ R5), is satisfiable because there are three models in which it is true, as shown in Figure 4.8.

Satisfiability can be checked by enumerating the possible models until one is found that satisfies the sentence.

The problem of determining the satisfiability of sentences in propositional logic—the SAT problem—was the first problem proved to be NP-complete. Many problems in computer science are really satisfiability problems.

Validity and satisfiability are of course connected: α is valid iff ¬α is unsatisfiable; contrapositively, α is satisfiable iff ¬α is not valid. We also have the following useful result: *α |= β if and only if the sentence (α ∧ ¬β) is unsatisfiable.*

Inference and proofs

Inference rules that can be applied to derive a proof—a chain of conclusions that leads to the desired goal. The best-known rule is called Modus Ponens (Latin for MODUS PONENS mode that affirms) and is written

$$\frac{\alpha \Rightarrow \beta, \quad \alpha}{\beta}.$$

The notation means that, whenever any sentences of the form α ⇒ β and α are given, then the sentence β can be inferred. For example, if (WumpusAhead ∧WumpusAlive) ⇒ Shoot and (WumpusAhead ∧ WumpusAlive) are given, then Shoot can be inferred.

Another useful inference rule is And-Elimination, which says that, from a conjunction, any of the conjuncts can be inferred

$$\frac{\alpha \wedge \beta}{\alpha}.$$

For example, from (WumpusAhead ∧ WumpusAlive), WumpusAlive can be inferred. By considering the possible truth values of α and β, one can show easily that Modus Ponens and And-Elimination are sound once and for all.

These rules can then be used in any particular instances where they apply, generating sound inferences without the need for enumerating models.

All of the logical equivalences in Figure 4.10 can be used as inference rules. For example, the equivalence for biconditional elimination yields the two inference rules

$$\frac{\alpha \Leftrightarrow \beta}{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}$$

$$\frac{(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)}{\alpha \Leftrightarrow \beta}.$$

Not all inference rules work in both directions like this. For example, we cannot run Modus Ponens in the opposite direction to obtain α ⇒ β and α from β.

Let us see how these inference rules and equivalences can be used in the wumpus world.

We start with the knowledge base containing R1 through R5 and show how to prove ¬P1,2, that is, there is no pit in [1,2]. First, we apply biconditional elimination to R2 to obtain

R6 : (B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

Then we apply And-Elimination to R6 to obtain

R7 : ((P1,2 ∨ P2,1) ⇒ B1,1) .

Logical equivalence for contrapositives gives

R8 : (¬B1,1 ⇒ ¬(P1,2 ∨ P2,1)) .

Now we can apply Modus Ponens with R8 and the percept R4 (i.e., ¬B1,1), to obtain

R9 : ¬(P1,2 ∨ P2,1) .

Finally, we apply De Morgan's rule, giving the conclusion

R10 : ¬P1,2 ∧ ¬P2,1 .

That is, neither [1,2] nor [2,1] contains a pit.

Define a proof problem as follows:

- INITIAL STATE: the initial knowledge base.
- ACTIONS: the set of actions consists of all the inference rules applied to all the sentences that match the top half of the inference rule.
- RESULT: the result of an action is to add the sentence in the bottom half of the inference rule.
- GOAL: the goal is a state that contains the sentence we are trying to prove.

Thus, searching for proofs is an alternative to enumerating models. In many practical cases finding a proof can be more efficient because the proof can ignore irrelevant propositions, no matter how many of them there are.

For example, the proof given earlier leading to $\neg P1,2 \wedge \neg P2,1$ does not mention the propositions B2,1, P1,1, P2,2, or P3,1.

They can be ignored because the goal proposition, P1,2, appears only in sentence R2; the other propositions in R2 appear only in R4 and R2; so R1, R3, and R5 have no bearing on the proof..

One final property of logical systems is monotonicity, which says that the set of entailed sentences can only increase as information is added to the knowledge base. For any sentences α and β,

$$\text{if} \quad KB \models \alpha \quad \text{then} \quad KB \wedge \beta \models \alpha \,.$$

For example, suppose the knowledge base contains the additional assertion β stating that there are exactly eight pits in the world.

This knowledge might help the agent draw additional conclusions, but it cannot invalidate any conclusion α already inferred—such as the conclusion that there is no pit in [1,2].

Monotonicity means that inference rules can be applied whenever suitable premises are found in the knowledge base—the conclusion of the rule must follow regardless of what else is in the knowledge base.

Proof by resolution

We begin by using a simple version of the resolution rule in the wumpus world. Let us consider the steps leading up to Figure 4.3 (a): the agent returns from [2,1] to [1,1] and then goes to [1,2], where it perceives a stench, but no breeze. We add the following facts to the knowledge base:

R11 : $\neg B1,2$.

R12 : $B1,2 \Leftrightarrow (P1,1 \vee P2,2 \vee P1,3)$.

By the same process that led to R10 earlier, we can now derive the absence of pits in [2,2] and [1,3] (remember that [1,1] is already known to be pitless):

R13 : $\neg P2,2$.

R14 : $\neg P1,3$.

We can also apply biconditional elimination to R3, followed by Modus Ponens with R5, to obtain the fact that there is a pit in [1,1], [2,2], or [3,1]:

R15 : $P1,1 \vee P2,2 \vee P3,1$.

Now comes the first application of the resolution rule: the literal $\neg P2,2$ in R13 resolves with the literal P2,2 in R15 to give the resolvent

R16 : $P1,1 \vee P3,1$.

In English; if there's a pit in one of [1,1], [2,2], and [3,1] and it's not in [2,2], then it's in [1,1] or [3,1]. Similarly, the literal $\neg P1,1$ in R1 resolves with the literal P1,1 in R16 to give

R17 : P3,1 .

In English: if there's a pit in [1,1] or [3,1] and it's not in [1,1], then it's in [3,1]. These last two inference steps are examples of the unit resolution inference rule,

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k},$$

k , where each is a literal and l_i and m_j are complementary literals (i.e., one is the negation of the other).

Thus, the unit resolution rule takes a clause a disjunction of literals and a literal and produces a new clause.

Note that a single literal can be viewed as a disjunction of one literal, also known as a unit clause.

The unit resolution rule can be generalized to the full resolution rule,

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

where l_i and mj are complementary literals. This says that resolution takes two clauses and produces a new clause containing all the literals of the two original clauses except the two complementary literals. For example, we have

$$\frac{P_{1,1} \vee P_{3,1}, \qquad \neg P_{1,1} \vee \neg P_{2,2}}{P_{3,1} \vee \neg P_{2,2}}.$$

There is one more technical aspect of the resolution rule: the resulting clause should contain only one copy of each literal.

The removal of multiple copies of literals is called factoring. For example, if we resolve (A ∨ B) with (A ∨ ¬B), we obtain (A ∨ A), which is reduced to just A.

The soundness of the resolution rule can be seen easily by considering the literal l_i that is complementary to literal mj in the other clause. If l_i is true, then $_{mj}$ is false, and hence m_1 ∨⋯∨ m_{j-1} ∨ m_{j+1} ∨⋯∨ m_n must be true, because m_1 ∨⋯∨ m_n is given. If l_i is false, then l_1 ∨⋯∨l_{i-1}∨ l_{i+1}∨⋯∨ k must be true because l_1 ∨⋯∨ l_k is given. Now l_i is either true or false, so one or other of these conclusions holds—exactly as the resolution rule states.

Conjunctive normal form

A sentence expressed as a conjunction of clauses is said to be in conjunctive normal form or CNF (see Figure 4.13). We now describe a procedure for converting to CNF. We illustrate the procedure by converting the sentence B1,1 ⇔ (P1,2 ∨ P2,1) into CNF. The steps are as follows:

1. Eliminate ⇔, replacing α ⇔ β with (α ⇒ β) ∧ (β ⇒ α).

(B1,1 ⇒ (P1,2 ∨ P2,1)) ∧ ((P1,2 ∨ P2,1) ⇒ B1,1) .

2. Eliminate ⇒, replacing α ⇒ β with ¬α ∨ β:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬(P1,2 ∨ P2,1) ∨ B1,1) .

3. CNF requires ¬ to appear only in literals, so we "move ¬ inwards" by repeated application of the following equivalences from Figure 7.11:

¬(¬α) ≡ α (double-negation elimination)

¬(α ∧ β) ≡ (¬α ∨ ¬β) (De Morgan)

¬(α ∨ β) ≡ (¬α ∧ ¬β) (De Morgan)

In the example, we require just one application of the last rule:

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ ((¬P1,2 ∧ ¬P2,1) ∨ B1,1) .

4. Now we have a sentence containing nested ∧ and ∨ operators applied to literals. We apply the distributivity law from Figure 7.11, distributing ∨ over ∧ wherever possible.

(¬B1,1 ∨ P1,2 ∨ P2,1) ∧ (¬P1,2 ∨ B1,1) ∧ (¬P2,1 ∨ B1,1) .

The original sentence is now in CNF, as a conjunction of three clauses. It is much harder to read, but it can be used as input to a resolution procedure.

A resolution algorithm

A resolution algorithm is shown in Figure 4.11. First, (KB ∧ ¬α) is converted into CNF. Then, the resolution rule is applied to the resulting clauses. Each pair that contains complementary literals is resolved to produce a new clause, which is added to the set if it is not already present.

The process continues until one of two things happens:

• there are no new clauses that can be added, in which case KB does not entail α; or,

• two clauses resolve to yield the empty clause, in which case KB entails α.

The empty clause—a disjunction of no disjuncts—is equivalent to False because a disjunction is true only if at least one of its disjuncts is true.

Another way to see that an empty clause represents a contradiction is to observe that it arises only from resolving two complementary unit clauses such as P and ¬P.

We can apply the resolution procedure to a very simple inference in the wumpus world. When the agent is in [1,1], there is no breeze, so there can be no pits in neighboring squares. The relevant knowledge base is

KB = R2 ∧ R4 = (B1,1 ⇔ (P1,2 ∨ P2,1)) ∧ ¬B1,1

and we wish to prove α which is, say, ¬P1,2. When we convert (KB ∧ ¬α) into CNF, we obtain the clauses shown at the top of Figure 4.12.

The second row of the figure shows clauses obtained by resolving pairs in the first row. Then, when P1,2 is resolved with ¬P1,2, we obtain the empty clause, shown as a small square. Inspection of Figure 7.13 reveals that many resolution steps are pointless.

For example, the clause B1,1 ∨¬B1,1∨P1,2 is equivalent to True ∨ P1,2 which is equivalent to True. Deducing that True is true is not very helpful. Therefore, any clause in which two complementary literals appear can be discarded

```
function PL-RESOLUTION(KB, α) returns true or false
  inputs: KB, the knowledge base, a sentence in propositional logic
          α, the query, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of KB ∧ ¬α
  new ← { }
  loop do
    for each pair of clauses Ci, Cj in clauses do
      resolvents ← PL-RESOLVE(Ci, Cj)
      if resolvents contains the empty clause then return true
      new ← new ∪ resolvents
    if new ⊆ clauses then return false
    clauses ← clauses ∪ new
```

Figure 7.12 A simple resolution algorithm for propositional logic. The function PL-RESOLVE returns the set of all possible clauses obtained by resolving its two inputs.

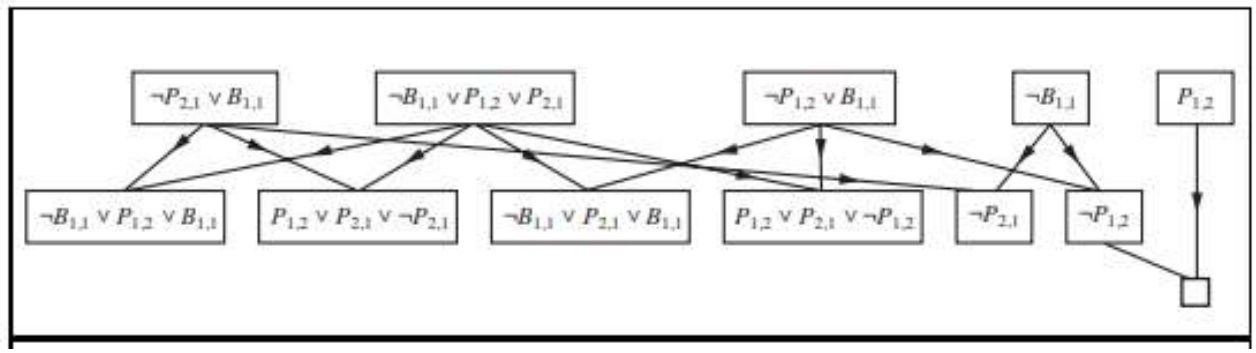


Figure 7.13 Partial application of PL- RESOLUTI ON to a simple inference in the wumpus world. ¬P1,2 is shown to follow from the first four clauses in the top row.

Completeness of resolution

The resolution closure RC (S) of a set of clauses S, which is the set of all clauses derivable by repeated application of the resolution rule to clauses in S or their derivatives. The resolution closure is what PL-RESOLUTION computes as the final value of the variable clauses. It is easy to see that RC (S) must be finite, because there are only finitely many distinct clauses that can be constructed out of the symbols P1,...,Pk that appear in S.

Hence, PL-RESOLUTION always terminates.

The completeness theorem for resolution in propositional logic is called the ground resolution theorem:

If a set of clauses is unsatisfiable, then the resolution closure of those clauses contains the empty clause.

This theorem is proved by demonstrating its contrapositive: if the closure RC(S) does not contain the empty clause, then S is satisfiable. In fact, we can construct a model for S with suitable truth values for P1,...,Pk. The construction procedure is as follows: For i from 1 to k,

– If a clause in RC(S) contains the literal ¬Pi and all its other literals are false under the assignment chosen for P1,...,Pi−1, then assign false to Pi.

– Otherwise, assign true to Pi. This assignment to P1,...,Pk is a model of S. To see this, assume the opposite—that, at some stage i in the sequence, assigning symbol Pi causes some clause C to become false. For this to happen, it must be the case that all the other literals in C must already have been falsified by assignments to P1,...,Pi−1.

Thus, C must now look like either (false ∨ false ∨ ⋯ false∨Pi) or like (false∨false∨⋯ false∨¬Pi). If just one of these two is in RC(S), then the algorithm will assign the appropriate truth value to Pi to make C true, so C can only be falsified if both of these clauses are in RC(S).

Now, since RC(S) is closed under resolution, it will contain the resolvent of these two clauses, and that resolvent will have all of its literals already falsified by the assignments to P1,...,Pi−1.

This contradicts our assumption that the first falsified clause appears at stage i. Hence, we have proved that the construction never falsifies a clause in RC(S); that is, it produces a model of RC(S) and thus a model of S itself (since S is contained in RC(S)).

Horn clauses and definite clauses

The completeness of resolution makes it a very important inference method. In many practical situations, however, the full power of resolution is not needed.

Some real-world knowledge bases satisfy certain restrictions on the form of sentences they contain, which enables them to use a more restricted and efficient inference algorithm.

One such restricted form is the definite clause, which is a disjunction of literals of which exactly one is positive. For example, the clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) is a definite clause, whereas (¬B1,1 ∨ P1,2 ∨ P2,1) is not.

Slightly more general is the Horn clause, which is a disjunction of literals of which at most one is positive. So all definite clauses are Horn clauses, as are clauses with no positive literals; these are called goal clauses. Horn clauses are closed under resolution: if you resolve two Horn clauses, you get back a Horn clause.

Knowledge bases containing only definite clauses are interesting for three reasons:

1. Every definite clause can be written as an implication whose premise is a conjunction of positive literals and whose conclusion is a single positive literal.

For example, the definite clause (¬L1,1 ∨ ¬Breeze ∨ B1,1) can be written as the implication (L1,1 ∧ Breeze) ⇒ B1,1.

In the implication form, the sentence is easier to understand: it says that if the agent is in [1,1] and there is a breeze, then [1,1] is breezy.

In Horn form, the premise is called the body and the conclusion is called the head. A sentence consisting of a single positive literal, such as L1,1, is called a fact. It too can be written in implication form as True ⇒ L1,1, but it is simpler to write just L1,1

$$
\begin{aligned}
\mathit{CNFSentence} &\rightarrow \mathit{Clause}_1 \wedge \cdots \wedge \mathit{Clause}_n \\
\mathit{Clause} &\rightarrow \mathit{Literal}_1 \vee \cdots \vee \mathit{Literal}_m \\
\mathit{Literal} &\rightarrow \mathit{Symbol} \mid \neg \mathit{Symbol} \\
\mathit{Symbol} &\rightarrow P \mid Q \mid R \mid \ldots \\
\mathit{HornClauseForm} &\rightarrow \mathit{DefiniteClauseForm} \mid \mathit{GoalClauseForm} \\
\mathit{DefiniteClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{Symbol} \\
\mathit{GoalClauseForm} &\rightarrow (\mathit{Symbol}_1 \wedge \cdots \wedge \mathit{Symbol}_l) \Rightarrow \mathit{False}
\end{aligned}
$$

Figure 4.13 A grammar for conjunctive normal form, Horn clauses, and definite clauses. A clause such as A ∧ B ⇒ C is still a definite clause when it is written as ¬A ∨ ¬B ∨ C, but only the former is considered the canonical form for definite clauses. One more class is the *k*-CNF sentence, which is a CNF sentence where each clause has at most *k* literals.

2. Inference with Horn clauses can be done through the forward-chaining and backward chaining algorithms.
3. Deciding entailment with Horn clauses can be done in time that is linear in the size of the knowledge base.

Forward and backward chaining

The forward-chaining algorithm PL-FC-ENTAILS?(KB, q) determines if a single proposition symbol q—the query—is entailed by a knowledge base of definite clauses.

It begins from known facts (positive literals) in the knowledge base.

If all the premises of an implication are known, then its conclusion is added to the set of known facts.

For example, if L1,1 and Breeze are known and (L1,1 ∧ Breeze) ⇒ B1,1 is in the knowledge base, then B1,1 can be added.

This process continues until the query q is added or until no further inferences can be made.

```
function PL-FC-ENTAILS?(KB, q) returns true or false
  inputs: KB, the knowledge base, a set of propositional definite clauses
          q, the query, a proposition symbol
  count ← a table, where count[c] is the number of symbols in c's premise
  inferred ← a table, where inferred[s] is initially false for all symbols
  agenda ← a queue of symbols, initially symbols known to be true in KB

  while agenda is not empty do
    p ← POP(agenda)
    if p = q then return true
    if inferred[p] = false then
      inferred[p] ← true
      for each clause c in KB where p is in c.PREMISE do
        decrement count[c]
        if count[c] = 0 then add c.CONCLUSION to agenda
  return false
```

Figure 4.14 The forward-chaining algorithm for propositional logic

.

It is easy to see that forward chaining is sound: every inference is essentially an application of Modus Ponens.

Forward chaining is also complete: every entailed atomic sentence will be derived. The easiest way to see this is to consider the final state of the inferred table.

The table contains true for each symbol inferred during the process, and false for all other symbols.

We can view the table as a logical model; moreover, every definite clause in the original KB is true in this model.

To see this, assume the opposite, namely that some clause a1∧...∧ak ⇒ b is false in the model. Then a1 ∧ ... ∧ ak must be true in the model and b must be false in the model.

But this contradicts our assumption that the algorithm has reached a fixed point! We can conclude, therefore, that the set of atomic sentences inferred at the fixed point defines a model of the original KB.

Furthermore, any atomic sentence q that is entailed by the KB must be true in all its models and in this model in particular. Hence, every entailed atomic sentence q must be inferred by the algorithm.

Forward chaining is an example of the general concept of data-driven reasoning—that is, reasoning in which the focus of attention starts with the known data. It can be used within an agent to derive conclusions from incoming percepts, often without a specific query in mind.

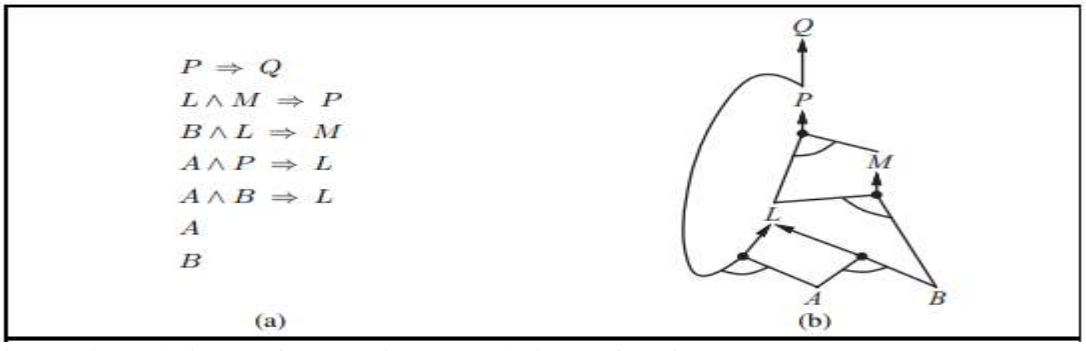


Figure 4.15 a) A set of Horn Clauses B) AND-OR Graph

The backward-chaining algorithm, as its name suggests, works backward from the query. If the query q is known to be true, then no work is needed.

Otherwise, the algorithm finds those implications in the knowledge base whose conclusion is q. If all the premises of one of those implications can be proved true (by backward chaining), then q is true. When applied to the query Q in Figure 4.15, it works back down the graph until it reaches a set of known facts, A and B, that forms the basis for a proof.

4.4 Propositional model checking:

Two families of efficient algorithms for general propositional inference based on model checking were described in this section: One approach based on backtracking search, and one on local hill-climbing search. These algorithms are part of the "technology" of propositional logic.

A complete backtracking algorithm

The first algorithm we consider is often called the Davis–Putnam algorithm, after the sem- inal paper by Martin Davis and Hilary Putnam (1960).

The algorithm is in fact the version described by Davis, Logemann, and Loveland (1962), so we will call it DPLL after the initials of all four authors. DPLL takes as input a sentence in conjunctive normal form—a set of clauses.

Like BACKTRACKING-SEARCH and TT-ENTAILS?, it is essentially a recursive, depth-first enumeration of possible models. It embodies three improvements over the simple scheme of TT-ENTAILS?:

- **Early termination:**

The algorithm detects whether the sentence must be true or false, even with a partially completed model. A clause is true if any literal is true, even if the other literals do not yet have truth values; hence, the sentence as a whole could be judged true even before the model is complete.

For example, the sentence $(A \vee B) \wedge (A \vee C)$ is true if A is true, regardless of the values of B and C.

Similarly, a sentence is false if any clause is false, which occurs when each of its literals is false. Again, this can occur long before the model is complete. Early termination avoids examination of entire subtrees in the search space. PURE SYMBOL

- **Pure symbol heuristic:**

A pure symbol is a symbol that always appears with the same "sign" in all clauses. For example, in the three clauses $(A \vee \neg B)$, $(\neg B \vee \neg C)$, and $(C \vee A)$, the symbol A is pure because only the positive literal appears, B is pure because only the negative literal appears, and C is impure.

It is easy to see that if a sentence has a model, then it has a model with the pure symbols assigned so as to make their literals true, because doing so can never make a clause false.

For example, if the model contains B = false, then the clause $(\neg B \vee \neg C)$ is already true, and in the remaining clauses C appears only as a positive literal; therefore C becomes pure.

- **Unit clause heuristic:**

A unit clause was defined earlier as a clause with just one literal. In the context of DPLL, it also means clauses in which all literals but one are already assigned false by the model.

For example, if the model contains B = true, then $(\neg B \vee \neg C)$ simplifies to $\neg C$, which is a unit clause.

Obviously, for this clause to be true, C must be set to false.

The unit clause heuristic assigns all such symbols before branching on the remainder. One important consequence of the heuristic is that any attempt to prove (by refutation) a literal that is already in the knowledge base will succeed immediately.

Notice also that assigning one unit clause can create another unit clause—for example, when C is set to false, (C ∨ A) becomes a unit clause, causing true to be assigned to A. This "cascade" of forced assignments is called unit propagation

```
function DPLL-SATISFIABLE?(s) returns true or false
  inputs: s, a sentence in propositional logic

  clauses ← the set of clauses in the CNF representation of s
  symbols ← a list of the proposition symbols in s
  return DPLL(clauses, symbols, { })

function DPLL(clauses, symbols, model) returns true or false

  if every clause in clauses is true in model then return true
  if some clause in clauses is false in model then return false
  P, value ← FIND-PURE-SYMBOL(symbols, clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P, value ← FIND-UNIT-CLAUSE(clauses, model)
  if P is non-null then return DPLL(clauses, symbols – P, model ∪ {P=value})
  P ← FIRST(symbols); rest ← REST(symbols)
  return DPLL(clauses, rest, model ∪ {P=true}) or
         DPLL(clauses, rest, model ∪ {P=false}))
```

Figure 4.16 The DPLL algorithm for checking satisfiability of a sentence in propositional logic.

The DPLL algorithm is shown in Figure 4.16, which gives the the essential skeleton of the search process

It is interesting that most of these tricks are in fact rather general, and we have seen them before in other guises:

1. **Component analysis**: As DPLL assigns truth values to variables, the set of clauses may become separated into disjoint subsets, called components, that share no unassigned variables. Given an efficient way to detect when this occurs, a solver can gain considerable speed by working on each component separately.

2. **Variable and value ordering**: Our simple implementation of DPLL uses an arbitrary variable ordering and always tries the value true before false. The degree heuristic suggests choosing the variable that appears most frequently over all remaining clauses.

3. Intelligent backtracking : Many problems that cannot be solved in hours of run time with chronological backtracking can be solved in seconds with intelligent backtracking that backs up all the way to the relevant point of conflict.

4. Random restarts: Sometimes a run appears not to be making progress. In this case, we can start over from the top of the search tree, rather than trying to continue. After restarting, different random choices are made. Restarting does not guarantee that a solution will be found faster, but it does reduce the variance on the time to solution.

5. Clever indexing: The speedup methods used in DPLL itself, as well as the tricks used in modern solvers, require fast indexing of such things as "the set of clauses in which variable Xi appears as a positive literal."

Local search algorithms

Several local search algorithms, including HILL-CLIMBING and SIMULATED-ANNEALING.

These algorithms can be applied directly to satisfiability problems, provided that we choose the right evaluation function.

Because the goal is to find an assignment that satisfies every clause, an evaluation function that counts the number of unsatisfied clauses will do the job.

In fact, this is exactly the measure used by the MIN-CONFLICTS algorithm for CSPs. All these algorithms take steps in the space of complete assignments, flipping the truth value of one symbol at a time.

The space usually contains many local minima, to escape from which various forms of randomness are required.

One of the simplest and most effective algorithms to emerge from all this work is called WALKSAT (Figure 4.17).

On every iteration, the algorithm picks an unsatisfied clause and picks a symbol in the clause to flip.

It chooses randomly between two ways to pick which symbol to flip:

(1) a "min-conflicts" step that minimizes the number of unsatisfied clauses in the new state and

(2) a "random walk" step that picks the symbol randomly.

When WALKSAT returns a model, the input sentence is indeed satisfiable, but when it returns failure, there are two possible causes: either the sentence is unsatisfiable or we need to give the algorithm more time.

If we set max flips = ∞ and p > 0, WALKSAT will eventually return a model (if one exists), because the random-walk steps will eventually hit upon the solution.

Alas, if max flips is infinity and the sentence is unsatisfiable, then the algorithm never terminates!

```
function WALKSAT(clauses, p, max_flips) returns a satisfying model or failure
  inputs: clauses, a set of clauses in propositional logic
          p, the probability of choosing to do a "random walk" move, typically around 0.5
          max_flips, number of flips allowed before giving up

  model ← a random assignment of true/false to the symbols in clauses
  for i = 1 to max_flips do
      if model satisfies clauses then return model
      clause ← a randomly selected clause from clauses that is false in model
      with probability p flip the value in model of a randomly selected symbol from clause
      else flip whichever symbol in clause maximizes the number of satisfied clauses
  return failure
```

4.5Agents based on propositional logic

The first step is to enable the agent to deduce, to the extent possible, the state of the world given its percept history.

Also the agent can keep track of the world efficiently without going back into the percept history for each inference.

Finally, we show how the agent can use logical inference to construct plans that are guaranteed to achieve its goals.

The current state of the world

The knowledge base is composed of axioms general knowledge about how the world works and percept sentences obtained from the agent's experience in a particular world.

The problem of deducing the current state of the wumpus world where am I, is that square safe, and so on.

The agent knows that the starting square contains no pit (¬P1,1) and no wumpus (¬W1,1).

Furthermore, for each square, it knows that the square is breezy if and only if a neighboring square has a pit; and a square is smelly if and only if a neighboring square has a wumpus.

Thus, we include a large collection of sentences of the following form

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$S_{1,1} \Leftrightarrow (W_{1,2} \vee W_{2,1})$$
$$\ldots$$

The agent also knows that there is exactly one wumpus. This is expressed in two parts. First, we have to say that there is at least one wumpus:

$$W_{1,1} \vee W_{1,2} \vee \cdots \vee W_{4,3} \vee W_{4,4} .$$

.

Then, we have to say that there is at most one wumpus. For each pair of locations, we add a sentence saying that at least one of them must be wumpus-free:

$$\neg W_{1,1} \vee \neg W_{1,2}$$
$$\neg W_{1,1} \vee \neg W_{1,3}$$
$$\ldots$$
$$\neg W_{4,3} \vee \neg W_{4,4} .$$

Let's consider the agent's percepts.

If there is currently a stench, one might suppose that a proposition Stench should be added to the knowledge base.

If there was no stench at the previous time step, then ¬Stench would already be asserted, and the new assertion would simply result in a contradiction.

The problem is solved when we realize that a percept asserts something only about the current time.

Thus, if the time step is 4, then we add Stench4 to the knowledge base, rather than Stench—neatly avoiding any contradiction with ¬Stench3. T

he same goes for the breeze, bump, glitter, and scream percepts.

The idea of associating propositions with time steps extends to any aspect of the world that changes over time.

Symbols associated with permanent aspects of the world do not need a time superscript and are sometimes called a temporal variables.

We can connect stench and breeze percepts directly to the properties of the squares where they are experienced through the location fluent as follows.10 For any time step t and any square [x, y], we assert

$$L^t_{x,y} \Rightarrow (Breeze^t \Leftrightarrow B_{x,y})$$
$$L^t_{x,y} \Rightarrow (Stench^t \Leftrightarrow S_{x,y}) .$$

First, we need proposition symbols for the occurrences of actions.

As with percepts, these symbols are indexed by time; thus, Forward 0 means that the agent executes the Forward action at time 0.

By convention, the percept for a given time step happens first, followed by the action for that time step, followed by a transition to the next time step.

To describe how the world changes, we can try writing effect axioms that specify the outcome of an action at the next time step.

For example, if the agent is at location [1, 1] facing east at time 0 and goes Forward, the result is that the agent is in square [2, 1] and no longer is in [1, 1]:

$$L^0_{1,1} \wedge FacingEast^0 \wedge Forward^0 \Rightarrow (L^1_{2,1} \wedge \neg L^1_{1,1}) .$$

We would also need similar sentences for the other actions: Grab, Shoot, Climb, TurnLeft, and TurnRight.

Let us suppose that the agent does decide to move Forward at time 0 and asserts this fact into its knowledge base.

Given the effect axiom in Equation, combined with the initial assertions about the state at time 0, the agent can now deduce that it is in [2, 1]. That is, ASK(KB, L1 2,1) = true.

Unfortunately, the news elsewhere is less good: if we ASK(KB, HaveArrow1), the answer is false, that is, the agent cannot prove it still has the arrow; nor can it prove it doesn't have it! The information has been lost because the effect axiom fails to state what remains unchanged as the result of an action.

$$Forward^t \Rightarrow (HaveArrow^t \Leftrightarrow HaveArrow^{t+1})$$
$$Forward^t \Rightarrow (WumpusAlive^t \Leftrightarrow WumpusAlive^{t+1})$$

The solution to the problem involves changing one's focus from writing axioms about actions to writing axioms about fluents.

Thus, for each fluent F, we will have an axiom that defines the truth value of Ft+1 in terms of fluents (including F itself) at time t and the actions that may have occurred at time t.

Now, the truth value of Ft+1 can be set in one of two ways: either the action at time t causes F to be true at t + 1, or F was already true at time t and the action at time t does not cause it to be false. An axiom of this form is called a successor-state axiom and has this schema

$$F^{t+1} \Leftrightarrow ActionCausesF^t \vee (F^t \wedge \neg ActionCausesNotF^t) .$$

One of the simplest successor-state axioms is the one for HaveArrow. Because there is no action for reloading, the ActionCausesFt part goes away and we are left with

$$HaveArrow^{t+1} \Leftrightarrow (HaveArrow^t \wedge \neg Shoot^t) .$$

For the agent's location, the successor-state axioms are more elaborate. For example, Lt+1 1,1 is true if either

(a) the agent moved Forward from [1, 2] when facing south, or from [2, 1] when facing west; or

(b) Lt 1,1 was already true and the action did not cause movement

$$\begin{aligned} L^{t+1}_{1,1} \Leftrightarrow\ & (L^t_{1,1} \wedge (\neg Forward^t \vee Bump^{t+1})) \\ \vee\ & (L^t_{1,2} \wedge (South^t \wedge Forward^t)) \\ \vee\ & (L^t_{2,1} \wedge (West^t \wedge Forward^t)) . \end{aligned}$$

Given a complete set of successor-state axioms and the other axioms listed at the beginning of this section, the agent will be able to ASK and answer any answerable question about the current state of the world.

For example, the initial sequence of percepts and actions is

$$\neg Stench^0 \wedge \neg Breeze^0 \wedge \neg Glitter^0 \wedge \neg Bump^0 \wedge \neg Scream^0 \ ; \ Forward^0$$
$$\neg Stench^1 \wedge Breeze^1 \wedge \neg Glitter^1 \wedge \neg Bump^1 \wedge \neg Scream^1 \ ; \ TurnRight^1$$
$$\neg Stench^2 \wedge Breeze^2 \wedge \neg Glitter^2 \wedge \neg Bump^2 \wedge \neg Scream^2 \ ; \ TurnRight^2$$
$$\neg Stench^3 \wedge Breeze^3 \wedge \neg Glitter^3 \wedge \neg Bump^3 \wedge \neg Scream^3 \ ; \ Forward^3$$
$$\neg Stench^4 \wedge \neg Breeze^4 \wedge \neg Glitter^4 \wedge \neg Bump^4 \wedge \neg Scream^4 \ ; \ TurnRight^4$$
$$\neg Stench^5 \wedge \neg Breeze^5 \wedge \neg Glitter^5 \wedge \neg Bump^5 \wedge \neg Scream^5 \ ; \ Forward^5$$
$$Stench^6 \wedge \neg Breeze^6 \wedge \neg Glitter^6 \wedge \neg Bump^6 \wedge \neg Scream^6$$

At this point, we have ASK(KB, L6 1,2) = true, so the agent knows where it is. Moreover, ASK(KB, W1,3) = true and ASK(KB, P3,1) = true, so the agent has found the wumpus and one of the pits.

The most important question for the agent is whether a square is OK to move into, that is, the square contains no pit nor live wumpus. It's convenient to add axioms for this, having the form

$$OK^t_{x,y} \Leftrightarrow \neg P_{x,y} \wedge \neg(W_{x,y} \wedge WumpusAlive^t) \ .$$

Finally, ASK(KB, OK6 2,2) = true, so the square [2, 2] is OK to move into. In fact, given a sound and complete inference algorithm such as DPLL, the agent can answer any answerable question about which squares are OK and can do so in just a few milliseconds for small-to medium wumpus worlds.

A hybrid agent

The ability to deduce various aspects of the state of the world can be combined fairly straightforwardly with condition–action rules and with problem-solving algorithms from to produce a hybrid agent for the wumpus world.

The agent program maintains and updates a knowledge base as well as a current plan. The initial knowledge base contains the a temporal axioms—those that don't depend on t, such as the axiom relating the breeziness of squares to the presence of pits.

At each time step, the new percept sentence is added along with all the axioms that depend on t, such as the successor-state axioms.

Then, the agent uses logical inference, by ASKing questions of the knowledge base, to work out which squares are safe and which have yet to be visited.

The main body of the agent program constructs a plan based on a decreasing priority of goals.

First, if there is a glitter, the program constructs a plan to grab the gold, follow a route back to the initial location, and climb out of the cave.

Otherwise, if there is no current plan, the program plans a route to the closest safe square that it has not visited yet, making sure the route goes through only safe squares.

Route planning is done with A∗ search, not with ASK. If there are no safe squares to explore, the next step—if the agent still has an arrow—is to try to make a safe square by shooting at one of the possible wumpus locations. These are determined by asking where ASK(KB,¬Wx,y) is false that is, where it is not known that there is not a wumpus.

The function PLAN-SHOT (not shown) uses PLAN-ROUTE to plan a sequence of actions that will line up this shot.

If this fails, the program looks for a square to explore that is not provably unsafe that is, a square for which ASK(KB,¬OKt x,y) returns false. If there is no such square, then the mission is impossible and the agent retreats to [1, 1] and climbs out of the cave.

Logical state estimation

The past history of percepts and all their ramifications can be replaced by the belief state that is, some representation of the set of all possible current states of the world.

The process of updating the belief state as new percepts arrive is called state estimation. The belief state was an explicit list of states, here we can use a logical sentence involving the proposition symbols associated with the current time step, as well as the a temporal symbols.

For example, the logical sentence

$$WumpusAlive^1 \wedge L^1_{2,1} \wedge B_{2,1} \wedge (P_{3,1} \vee P_{2,2})$$

represents the set of all states at time 1 in which the wumpus is alive, the agent is at [2, 1], that square is breezy, and there is a pit in [3, 1] or [2, 2] or both.

Maintaining an exact belief state as a logical formula turns out not to be easy.

If there are n fluent symbols for time t, then there are 2n possible states—that is, assignments of truth values to those symbols.

function HYBRID-WUMPUS-AGENT(*percept*) **returns** an *action*
  inputs: *percept*, a list, [*stench*, *breeze*, *glitter*, *bump*, *scream*]
  persistent: *KB*, a knowledge base, initially the atemporal "wumpus physics"
    t, a counter, initially 0, indicating time
    plan, an action sequence, initially empty

  TELL(*KB*, MAKE-PERCEPT-SENTENCE(*percept*, *t*))
  TELL the *KB* the temporal "physics" sentences for time *t*
  $safe \leftarrow \{[x, y] : \text{ASK}(KB, OK^t_{x,y}) = true\}$
  if $\text{ASK}(KB, Glitter^t) = true$ **then**
    $plan \leftarrow [Grab] + \text{PLAN-ROUTE}(current, \{[1,1]\}, safe) + [Climb]$
  if *plan* is empty **then**
    $unvisited \leftarrow \{[x, y] : \text{ASK}(KB, L^{t'}_{x,y}) = false \text{ for all } t' \leq t\}$
    $plan \leftarrow \text{PLAN-ROUTE}(current, unvisited \cap safe, safe)$
  if *plan* is empty and $\text{ASK}(KB, HaveArrow^t) = true$ **then**
    $possible\_wumpus \leftarrow \{[x, y] : \text{ASK}(KB, \neg W_{x,y}) = false\}$
    $plan \leftarrow \text{PLAN-SHOT}(current, possible\_wumpus, safe)$
  if *plan* is empty **then** // no choice but to take a risk
    $not\_unsafe \leftarrow \{[x, y] : \text{ASK}(KB, \neg OK^t_{x,y}) = false\}$
    $plan \leftarrow \text{PLAN-ROUTE}(current, unvisited \cap not\_unsafe, safe)$
  if *plan* is empty **then**
    $plan \leftarrow \text{PLAN-ROUTE}(current, \{[1, 1]\}, safe) + [Climb]$
  $action \leftarrow \text{POP}(plan)$
  TELL(*KB*, MAKE-ACTION-SENTENCE(*action*, *t*))
  $t \leftarrow t + 1$
  return *action*

function PLAN-ROUTE(*current*, *goals*, *allowed*) **returns** an action sequence
  inputs: *current*, the agent's current position
    goals, a set of squares; try to plan a route to one of them
    allowed, a set of squares that can form part of the route

  problem $\leftarrow$ ROUTE-PROBLEM(*current*, *goals*, *allowed*)
  return A*-GRAPH-SEARCH(*problem*)

Figure 4.19 A hybrid agent program for the wumpus world. It uses a propositional knowledge base to infer the state of the world, and a combination of problem-solving search and domain-specific code to decide what actions to take.

One very common and natural scheme for approximate state estimation is to represent belief states as conjunctions of literals, that is, 1-CNF formulas. To do this, the agent program simply tries to prove Xt and ¬Xt for each symbol Xt (as well as each atemporal symbol whose truth value is not yet known), given the belief state at t-1.

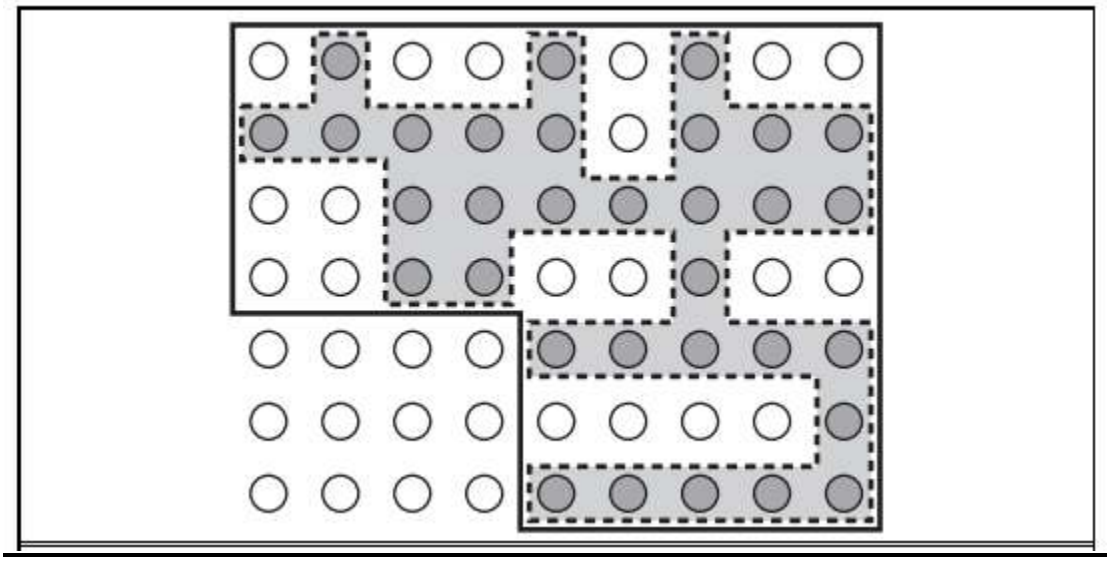


Figure 4.20 Depiction of a 1-CNF belief state (bold outline) as a simply representable, conservative approximation to the exact (wiggly) belief state (shaded region with dashed outline). Each possible world is shown as a circle; the shaded ones are consistent with all the percepts.

The conjunction of provable literals becomes the new belief state, and the previous belief state is discarded.

Making plans by propositional inference

The agent in Figure 7.20 uses logical inference to determine which squares are safe, but uses A∗ search to make plans. In this section, we show how to make plans by logical inference. The basic idea is very simple:

1. Construct a sentence that includes

(a) Init0, a collection of assertions about the initial state;

(b) Transition1,..., Transitiont , the successor-state axioms for all possible actions at each time up to some maximum time t;

(c) the assertion that the goal is achieved at time t: HaveGoldt ∧ ClimbedOutt .

2.Present the whole sentence to a SAT solver. If the solver finds a satisfying model, then the goal is achievable; if the sentence is unsatisfiable, then the planning problem is impossible.

3. Assuming a model is found, extract from the model those variables that represent actions and are assigned true. Together they represent a plan to achieve the goals.

A propositional planning procedure, SATPLAN, is shown in Figure 4.21. It implements the basic idea just given, with one twist. Because the agent does not know how many steps it will take to reach the goal, the algorithm tries each possible number of steps t, up to some maximum conceivable plan length Tmax. In this way, it is guaranteed to find the shortest plan if one exists. Because of the way SATPLAN searches for a solution, this

approach cannot be used in a partially observable environment; SATPLAN would just set the unobservable variables to the values it needs to create a solution.

```
function SATPLAN(init, transition, goal, T_max) returns solution or failure
  inputs: init, transition, goal, constitute a description of the problem
          T_max, an upper limit for plan length

  for t = 0 to T_max do
    cnf ← TRANSLATE-TO-SAT(init, transition, goal, t)
    model ← SAT-SOLVER(cnf)
    if model is not null then
      return EXTRACT-SOLUTION(model)
  return failure
```

Figure 4.21 The SATPLAN algorithm.

4.6 First order logic:

FOL is sufficiently expressive to represent a good deal of our commonsense knowledge. **First-Order Logic** is a logic which is sufficiently expressive to represent a good deal of our commonsense knowledge.

It is also either includes or forms the foundation of many other representation languages.

It is also called as **First-Order Predicate calculus.**

It is abbreviated as **FOL** or **FOPC**

FOL adopts the foundation of propositional logic with all its advantages to build a more expressive logic on that foundation, borrowing representational ideas from natural language while avoiding its drawbacks

Representation Revisited

The syntax of natural language, the most obvious elements are nouns and noun phrases that refer to objects (squares, pits, wumpuses) and verbs and verb phrases that refer to relations among objects (is breezy, is adjacent to, shoots).

Some of these relations are functions-relations in which there is only one "value" for a given "input."

Objects: people, houses, numbers, theories, Ronald McDonald, colors, baseball games, wars, centuries . . .

Relations: these can be unary relations or properties such as red, round, bogus, prime, multistoried . . ., or more general n-ary relations such as brother of, bigger than, inside, part of, has color, occurred after, owns, comes between, . . .

Functions: father of, best friend, third inning of, one more than, beginning of. . .

Some examples follow:

"One plus two equals three"

Objects: one, two, three, one plus two;

Relation: equals; Function: plus.

"Squares neighboring the wumpus are smelly."
Objects: wumpus, squares;
Property: smelly;
Relation: neighboring.
"Evil King John ruled England in 1200."
Objects: John, England, 1200;
Relation: ruled;
Properties: evil, king.

Objects: One, Two, Three, One plus Two
Relations: equals
Function: plus

- Ontological commitment of First-Order logic language is "Facts", "Objects", and "Relations".
- Where ontological commitment means "WHAT EXISTS IN THE WORLD".
- Epistemological Commitment of First-Order logic language is "True", "False", and "Unknown".

Where epistemological commitment means "WHAT AN AGENT BELIEVES ABOUT FACTS".

Advantages:

It has been so important to mathematics, philosophy, and Artificial Intelligence precisely because those fields can be usefully thought of as dealing with objects and the relations among them.

It can also express facts about some or all of the objects in the universe.

It enables one to represent general laws or rules, such as the statement "Squares neighboring the wumpus are smelly".

4.7 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.

Models for propositional logic are just sets of truth values for the proposition symbols.

Models for first-order logic are more interesting.

First they have objects in them.

The domain of a model is the set of objects it contains; these objects are sometimes called domain elements.

The following diagram shows a model with five objects

The **domain** of a model is the set of objects it contains; these objects are sometimes called **domain elements.**

Figure below shows a model with five objects:

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; and a crown.

The objects in the model may be related in various ways.

In the figure, Richard and John are brothers. Formally speaking, a relation is just the set of **tuples** of objects that are related.

A tuple is a collection of objects arranged in ,a fixed order and is written with angle brackets surrounding the objects.

Thus, the brotherhood relation in this model is the set

{ (Richard the Lionheart, King John), (King John, Richard the Lionheart) }

The crown is on King John's head, so the "on head relation contains just one tuple, (the crown, King John).

The "brother" and "on head" relations are binary relations-that is, they relate pairs of objects.

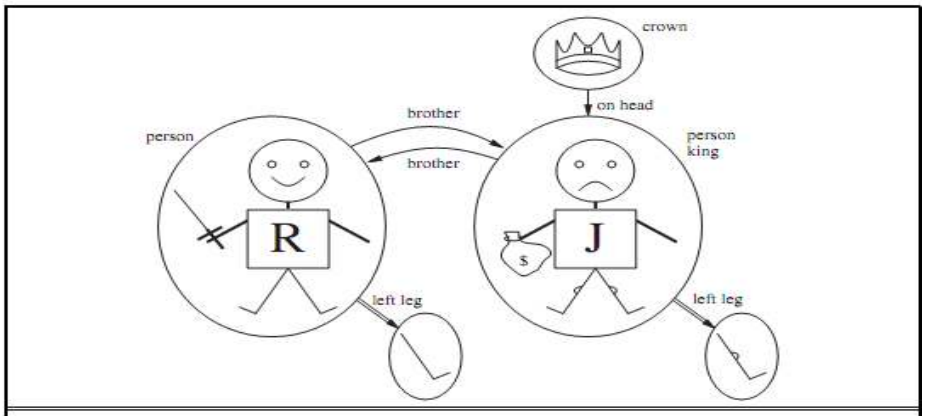


Figure 4.22 A model containing five objects, two binary relations, three unary relations (indicated by labels on the objects), and one unary function, left-leg.

The relation can be binary relation relating pairs of objects (Ex:- "Brother") or unary relation representing a common object (Ex:- "Person" representing both Richard and John)

Certain kinds of relationships are best considered as functions, in that a given object must be related to exactly one object in this way.

For example, each person has one left leg, so the model has a unary "left leg" function that includes the following mappings:

(Richard the Lionheart) ->Richard's left leg

(King John) ->John's left leg .

Symbols and interpretations

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

The symbols, therefore, come in three kinds:

- constant symbols, which stand for objects;
- predicate symbols, which stand for relations; and
- function symbols, which stand for functions.
- Symbols will begin with uppercase letters

The choice of names is entirely up to the user

Each predicate and function symbol comes with an arity

Arity fixes the number of arguments.

The semantics must relate sentences to models in order to determine truth.

To do this, an interpretation is needed specifying exactly which **objects, relations** and **functions** are referred to by the **constant, predicate and function symbols**.

One possible interpretation called as the intended interpretation- is as follows;

Richard refers to **Richard the Lionheart** and **John** refers to the **evil King John.**

Brother refers to the brotherhood relation, that is the set of tuples of objects given in equation

{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}

OnHead refers to the "on head" relation that holds between the crown and King John;

Person, King and **Crown** refer to the set of objects that are persons, kings and crowns. **Leftleg** refers to the "left leg" function, that is, the mapping given in **{(Richard the Lionheart, King John),(King John, Richard the Lionheart)}**

There are many other possible interpretations relating these symbols to this particular model.

The truth of any sentence is determined by a model and an interpretation for the sentence symbols.

The syntax of the FOL with equality specified in BNF is as follows

Sentence	⟶	AtomicSentence
|		(Sentence Connective Sentence)
|		Quantifier Variable,…Sentence
|		- Sentence
AtomicSentence	⟶	Predicate (Term…) | Term = Term
Term	⟶	Function (Term,….)
|		Constatnt
|		Variable
Connective	⟶	⇒ | Λ | V | □
□		
Quantifier	⟶	∀ | ∃
Constant	⟶	**A | X1 | John**
Variable	⟶	**a | x | s | …**
Predicate	⟶	**Before | HasColor | Raining | ….**
Function	⟶	**Mother | LeftLeg | ….**

Where,

Λ	Logical Conjunction
V	Logical disjunction
∀	Universal Quantification
∃	Existential Quantification
⇔	Material Equivalence
⇒	Material Implication

□ **Terms:**

Terms

A **term** is a logical expression that refers to an object.

Constant symbols are therefore terms, but it is not always convenient to have a distinct symbol to name every object.

For example, in English we might use the expression "King John's left leg" rather than giving a name to his leg.

A complex term is formed by a function symbol followed by a parenthesized list

of terms as arguments to the function symbol.

It is just like a complicated kind of name. It's not a "subroutine call" that returns a value".

The formal semantics of terms is straight forward, Consider a term $f(t1 \ldots\ldots t_n)$

Where

f- some function in the model (call it F)

The argument terms – objects in the domain

The term – object that is the value of the function F applied to the domain

For Example:- suppose the **LeftLeg** function symbol refers to the function is, (Richard the Lionheart) ------ Richards left leg

(King John) ------ Johns left leg

John refers to King John, then **LeftLeg** (John) refers to king Johns 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:

Brother(Richard, John).

This states, under the intended interpretation given earlier, that Richard the Lionheart is the brother of King John Atomic sentences can have complex terms as arguments. Thus,

Married (Father(Richard), Mother(John))

states that Richard the Lionheart's father is married to King John's mother

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

We can use **logical connectives** to construct more complex sentences, just as in propositional calculus. The semantics of sentences formed with logical connectives is identical to that in the propositional case. Here are four sentences that are true in the model of Figure above under our intended interpretation:

¬Rrother(LeftLeg(Richard,)J ohn)

Brother(Richard, John) ∧Brother(John, Richard)

King (Richard) ∨King (John)

¬King (Richard) →King (John) .

Quantifiers

we have a logic that allows objects, it is only natural to want to express properties of entire collections of objects, instead of enumerating the objects by name. **Quantifiers** let us do this. First-order logic contains two standard quantifiers, called ***univevsal*** and***existential.***

Universal quantification (∀)

The second rule, "All kings are persons," is written in first-order logic as

∀ xKing(x) →Person(x)

'∀is usually pronounced "For all . . .". Thus, the sentence says, "For all *x,* if *x* is a king, then z is a person." The symbol x is called a **variable.**

A term with no variables is called a **ground term.**

We can extend the interpretation in five 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 ∀ ***xKing(x) →Person(x)*** is true under the originalinterpretation if the sentence *Kzng(x) ->Person(x)* is true in each of the five extended pretations. That is, the universally quantified sentence is equivalent to asserting the followingfive 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. For example, that King John has a crown on his head, we write

∃x Crown(x) ^OnHead (x, John) .

∃x is pronounced "There exists an x such that . . ." or "For some x . . .".
For our example, this means that at least one of the following must be true:

Richard the Lionheart is a crown ^Richard the Lionh~earits on John's head;
King John is a crown ^ King John is on John's head;
Richard's left leg is a crown ^Richard's left leg is on John's head;
John's left leg is a crown ^ John's left leg is on John's head;
The crown is a crown ^ the crown is on John's head.

Nested quantifiers

We will often want to express more complex sentences using multiple quantifiers. The simplest case is where the quantifiers are of the same type. 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) .**

"Everybody loves somebody" means that for every person, there is someone that person loves:

∀x∃y Loves(x, y) .

"There is someone who is loved by everyone," we write

∃y ∀xLoves(x, y) .

The order of quantification is therefore very important. It becomes clearer if we insert parentheses. '∀*x*(∃*y Loves (x, y))* says that *everyone* has a particular property, namely, the property that they love someone. On the other hand, ∃*y (*∀*x Loves(x, y))* says that *someone* in the world has a particular property, namely the property of being loved by everybody.

Some confusion can arise when two quantifiers are used with the same variable name. Consider the sentence

∀x [Crown(x)ᵛ(∃x Brother (Richard,x))].

Here the *x* in *Brother(Richard, x)* is *existentially* quantified. The rule is that the variable belongs to the innermost quantifier that mentions it; then it will not be subject to any other quantification. Another way to think of it is this: ∃***x*** *Brother(Richard, x)* is a sentence about Richard (that he has a brother), not about *x;* so putting a∀***x***outside it has no effect. It could equally well have been written ∃*z Brother (Richard, z).* Because this can be a source of confusion, we will always use different variables.

Connections between ∀ and ∃

The two quantifiers are actually intimately connected with each other, through negation. Asserting that everyone dislikes parsnips is the same **as** asserting there does not exist someone who likes them, and vice versa:

***'⌐∀ x Lkes(x, Parsnips)* is equivalent to ⌐∃*x Likes(x, Parsnips)* .**

We can go one step further: "Everyone likes ice cream" means that there is no one who does not like ice cream:

∀ *x Lkes(x, Ice Cream)* is equivalent to ⌐∃*x'⌐ Likes(x, Ice Cream)* .

Because ∀ is really a conjunction over the universe of objects and ∃ is a disjunction, it should not be surprising that they obey De Morgan's rules. The De Morgan rules for quantified and unquantified sentences are as follows:

$$\begin{array}{lll} \forall x\ \neg P \equiv \neg\exists x\ P & & \neg(P \vee Q) \equiv \neg P \wedge \neg Q \\ \neg\forall x\ P \equiv \exists x\ \neg P & & \neg(P \wedge Q) \equiv \neg P \vee \neg Q \\ \forall x\ P \equiv \neg\exists x\ \neg P & & P \wedge Q \equiv \neg(\neg P \vee \neg Q) \\ \exists x\ P \equiv \neg\forall x\ \neg P & & P \vee Q \equiv \neg(\neg P \wedge \neg Q)\ . \end{array}$$

Equality

First-order logic includes one more way to make atomic sentences, other than using a predicate and terms. We can use the **equality symbol** to make statements to the effect 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.

4.8 Knowledge Engineering in First-Order Logic:

The general process of knowledge base construction process is called knowledge engineering.

A knowledge engineer is someone who investigates a particular domain, learns what concepts are important in that domain, and creates a formal representation of the objects and relations in the domain.

The steps associated with the knowledge engineering process are:

Identify the Task: - The Knowledge engineer should identify the **PEAS** description of the domain.

Assemble the relevant knowledge: - The idea of combining expert's knowledge of that domain (i.e.) a process called **knowledge acquisition.**

Decide on a vocabulary of predicates, functions and constants: - Translate the important domain-level concepts into logical level name. The resulting vocabulary is known as **ontology** of the domain, which determines what kinds of things exist, but does not determine their specific properties and inter relationships.

Encode general knowledge about the domain: - The knowledge engineer writes the axioms (rules) for all the vocabulary terms. The misconceptions are clarified from step 3 and the process is iterated.

Encode a description of the specific problem instance: - To write simple atomic sentences about instances of concepts that are already part of the ontology.

Pose queries to the inference procedure and get answers: - For the given query the inference procedure operate on the axioms and problem specific facts to derive the answers.

Debug the knowledge base: - For the given query , if the result is not a user expected on then **KB is** updated with relevant or missing axioms.

The seven step process is explained with the domain of **ELECTRONIC CIRCUITS DOMAIN.**

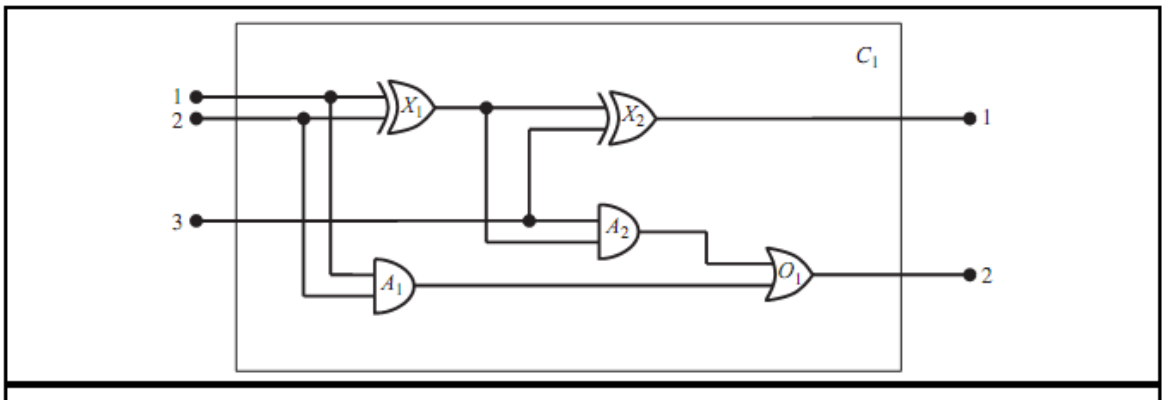


Figure 4.23 A digital circuit C1, purporting to be a one-bit full adder.

Identify the task

Analyse the circuit functionality, does the circuit actually add properly? (Circuit Verification).

Assemble the relevant knowledge

The circuit is composed of wire and gates.

The four types of gates (AND, OR, NOT, XOR) with two input terminals and one output terminal knowledge is collected.

Decide on a vocabulary

The functions, predicates and constants of the domain are identified.

Functions are used to refer the type of gate.

Type (x1) = XOR,

where x1 ---- Name of the gate and Type ---- function

The same can be represented by either binary predicate (or) individual type predicate.

Type (x1, XOR) – binary predicate

XOR (x1) – Individual type

A gate or circuit can have one or more terminals. For x1, the terminals are x1In1, x1In2 and x1 out1

Where x1 In1 ----- 1st input of gate x1 x1 In2 ----- 2nd input of gate x1

x1 out1 ---- output of gate x1

Then the connectivity between the gates represented by the predicate connected. (i.e.) connected (out(1, x1), In(1,x2)).

Finally the possible values of the output terminal C1, as true or false, represented as a signal with 1 or 0.

Encode general knowledge of the domain

is example needs only seven simple rules to describe everything we need to know about circuits:

1. If two terminals are connected, then they have the same signal:
2. The signal at every terminal is either 1 or 0 (but not both):
3. Connected is a commutative predicate:
4. An OR gate's output is 1 if and only if any of its inputs is 1:
5. An **A.ND** gate's output is 0 if and only if any of its inputs is 0:
6. An XOR gate's output is 1 if and only if its inputs are different:
7. A NOT gate's output is different from its input:

Encode the specific problem instance

The circuit shown in Figure 8.4 is encoded as circuit *C1* with the following description. First, we categorize the gates:

Type(X1)=XOR Type(X2)= XOR

Pose queries to the inference procedure

What combinations of inputs would cause the first output of *Cl*(the sum bit) to be 0 and the second output of *C1* (the carry bit) to be l?

Debug the knowledge base:

The knowledge base is checked with different constraints.

Using First-Order Logic

Assertions and queries in first-order logic

Sentences are added to a knowledge base using TELL, exactly as in propositional logic. Such ASSERTIONS sentences are called **assertions.** For example, we can assert that John is a king and that kings are persons:

TELL(KB, King (John)) .

TELL(KB, ∀ x King(x) ->Person(x)).

We can ask questions of the knowledge base using ASK. For example,

ASK(KB,king(John)

returns*true.* Questions asked using ASK are called **queries** or **goals**. Generally speaking, any query that is logically entailed by the knowledge base should be answered affirmatively. For example, given the two assertions in the preceding paragraph, the query

ASK(KB,Person(John))

should also return *true.* We can also ask quantified queries, such as

$$\text{ASK}(KB, \exists x \ \ Person(x))\ .$$

A query with existential variables is asking "Is there an *x* such that . . .," and we solve it by providing such an *x.* The standard form for an answer of this sort is a **substitution** or **binding list,** which is a set of variable/term BINDING LIST pairs. In this particular case, given just the two assertions, the answer would be *{xlJohn).* If there is more than one possible answer, a list of substitutions can be returned.

The kinship domain

The first example we consider is the domain of family relationships, or kinship. This domain includes facts such as "Elizabeth is the mother of Charles" and "Charles is the father of

William' and rules such as "One's grandmother is the mother of one's parent."

Clearly, the objects in our domain are people. We will have two unary predicates, *Male* and *Female.* Kinship relations-parenthood, brotherhood, marriage, and so on-will be represented by binary predicates: *Parent, Sibling, Brother, Sister, Child, Daughter,Son, Spouse, Wife, Husband, Grandparent, Grandchild, Cousin, Aunt,* and *Uncle.* We will use functions for *Mother* and *Father,* because every person has exactly one of each of these .

For example, one's mother is one's female parent:

∀m, c Mother(c) = m ↔ Female(m) ∧ Parent(m, c) .

One's husband is one's male spouse:

∀w,h Husband(h,w) ↔ Male(h) ∧ Spouse(h,w).

Male and female are disjoint categories:

∀x Male (x) ↔ ⌐ Female(x) .

Parent and child are inverse relations:

∀p, c Parent (p, c) ↔ Child (c, p) .

A grandparent is a parent of one's pai-ent:

$\forall g,c\ Grandparentc(g,c) \leftrightarrow \exists p\ Parent(g,P) \wedge Parent(p,c)$

A sibling is another child of one's parents:

$\forall x, y\ Szbling(x, y)\ e\ x \neq y \wedge \exists p\ Parent(p, x) \wedge Parent(p, y)$.

Numbers, sets, and lists

The theory of **natural numbers** or nonnegative integers. We need a predicate *NatNum*that will be true of natural numbers; we need one constant symbol, *0;* and we need one function symbol, *S* (successor). The **Peano axioms** define natural numbers and addition .Na tural 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(O),S (S(O))*a,n d so on. We also need axioms to constrain the successor function:

$'\forall n 0 \neq S(n).$

$\forall m,n\ m \neq n \rightarrow S(m) \neq S(n)$.

Now we can define addition in terms of the successor function:

$\forall m\ NatNum(m) \rightarrow +(0,m) = m.$

$\forall m,n\ NatNum(m) \wedge NatNum(n) \rightarrow +(S(m)n,) = S(+(m,n)).$

We will use the normal vocabulary of set theory as syntactic sugar. 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)* and *slc s2* (set *sl*is a subset, not necessarily proper, of set *s2).* The binary functions are *sl^ s2* (the intersection of two sets), *sl*U *s2* (the union of two sets), and *{x(s)*(the set resulting f~omad joining element *x* to set *s).* One possible set of axioms is as follows:

1. The only sets are the empty set and those made by adjoining something to a set:
2. 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:
3. Adjoining an element already in the set has no effect:
4. 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 I the same as *x* or *x* is a member of *s2:*
5. 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:
6. Two sets are equal if and only if each is a subset of the other:
7. An object is in the intersection of two sets if and only if it is a member of both sets:
8. An object is in the union of two sets if and only if it is a member of either set:

Lists are similar to sets. The differences are that lists are ordered and the same element can appear 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; and *Find* is the predicate that does for lists what *Member* does for sets.

The wumpus world

The first order axioms in this section are much more concise, capturing in a very natural way exactly what we want to say.

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. 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, Release, Climb .

To determine which is best, the agent program constructs a query such as

ASKVARS(∃ a BestAction(a, 5)) ,

ASK should solve this query and return a binding list such as *{alGrab).* The agent program can then return *Grab* as the action to take, but first it must TELL its own knowledge base that it is performing a *Grab.*

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([sb, , Glitter, m, c],t) ->Glitter(t),

and so on. These rules exhibit a trivial form of the reasoning process called **perception** the sentences dealing with time have been **synchronic** ("same time") sentences, that is, they relate properties of a world state to other properties of the same world state. Sentences that allow reasoning "across time" are called **diachronic**

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,

∀s Breezy(s) -> ∃r Adjacent (r, s) ^Pit(r) ,

and that if a square is not breezy, no adjacent square contains a pit: lo

***∀s1 Breezy (s) ->⌐∃r Adjacent (r, s) ^ Pit* (,r)**

Combining these two, we obtain the biconditional sentence

∀s Breezy (s) ↔ ∃ r Adjacent(r, s) ^ 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:

∀rPit->*[∀s *Adjacent (r, s) ->Breezy (s) .

∀s*[∀r *Adjacent (r, s) ->⌐Pit (r)] ->⌐Breezy (s) .

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

4.9 INFERENCE IN FIRST-ORDER LOGIC

Propositional versus First- Order Inference

These rules lead naturally to the idea that first-order inference can be done by converting the knowledge base to propositional logic and using propositional inference, which we already know how to do.

Let us begin with universal quantifiers. Suppose our knowledge base contains the standard axiom stating that all greedy kings are evil:

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

Then it seems quite permissible to infer any of the following sentences:

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

The rule of **Universal Instantiation** says that we can infer any sentence obtained by substituting a **ground term** for the variable. To write out the inference rule formally, we use the notion of **substitutions**.

Let $\text{SUBST}(\theta, a)$ denote the result of applying the substitution *8* 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, the three sentences given earlier are obtained with the substitutions *1x1 John), {x/Richard),* arid *{x/Father(John)).*

The corresponding **Existential Instantiation** rule: for the existential quantifier is slightly more complicated. 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

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

we can infer the sentence

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

as long as *C1* does not appear elsewhere in the knowledge base. Basically, the existential sentence says there is some object satisfying a condition, and the instantiation process is just giving a name to that object.

Reduction to propositional inference

The first idea is that, just as an existentially quantified sentence can be replaced by one instantiation, a universally quantified sentence can be replaced by the set of *all* possible instantiations. For example, suppose our knowledge base contains just the sentences

∀xKzng (x) ^ Greedy(x) ->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) ^ Greedy(John) ->Evil (John) ,
King (Richard) ^ Greedy (Richard) ->Evil (Richard) ,

and 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 so on-as proposition symbols.

4.10 Forward Chaining:

A forward-chaining algorithm idea is simple it starts with the atomic sentences in the knowledge base and apply Modus Ponens in the forward direction, adding new atomic sentences, until no further inferences can be made.

Definite clauses such as *Situation =>Response* are especially useful for systems that make inferences in response to newly arrived information.

First-order definite clauses

First-order definite clauses closely resemble propositional definite clauses they are disjunctions of literals of which *exactly one is positive.*

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) ∧ Greedy (x) ->Evil (x) .

Kzng(John) .

Greedy(y) .

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 prove that West is a criminal. First, we will represent these facts as first-order definite clauses. The next section shows how the forward-chaining algorithm solves the prloblem.

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

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

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

Owns (Nono, M I ***2***

Mzssile(Ml)

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

missile (x) ∧ Owns (Nono, x) ->Sells(West,* z, *Nono) ***3***

We will also need to know that missiles are weapons:

Missile (x) =>Weapon (x) ***4***

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

Enemy(x, America) =>Hostile(x) ***5***

"West, who is American . . .":

American(West) ***6***

"The country Nono, an enemy of America . . .":

Enemy(Nono, Anzerica) . ***7***

This knowledge base contains no function symbols and is therefore an instance of' the class Of **Datalog** knowledge bases.

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 (assuming that just one answer is required) or no new facts are added.

. One sentence is a renaming of another if they are identical except for the names of the variables.

For example, *Likes(x. Icecream)* and *Likes(y, Icecream)* are renamings of each other because they differ only in the choice of *x* or y; their meanings are identical: everyone likes ice cream.

We will use our crime problem to illustrate how FOL-FC-ASK works. The implication sentences are (1), (3), (4), and (5).

Two iterations are required:

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

Rule (3) is satisfied with *{x/Ml),* and *Sells(West,* MI, *Nono)* is added.

Rule (4) is satisfied with *{ x / M1)* an,d*Weapon(M1)*i s added.
Rule (5) is satisfied with *{x/Nono},* and *Hostile(Nono)* is added.

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

Figure shows the proof tree that is generated. Notice that no new inferences are possible at this point because every sentence that could be concluded by forward chaining is already contained explicitly in the KB. Such a knowledge base is called a **fixed point** of the inference process.

FOL-FC-ASK is easy to analyze.

First, it is **sound,** because every inference is just an application of Generalized Modus Ponens, which is sound.

Second, 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.

Efficient forward chaining

There are three possible sources of complexity.

First, the "inner loop" of the algorithm involves finding all possible unifiers such that the piremise of a rule unifies with a suitable set of facts in the knowledge base. This is often called **pattern matching** and can be very expensive.

Second, the algorithm rechecks every rule on every iteration to see whether its premises are satisfied, even if very few additions are made to the knowledge base on each iteration.

Finally, the algorithm might generate many facts that are irrelevant to the goal. We will address each of these sources in turn. **Matching rules against known .**The problem of matching the premise of a rule against the facts in the knowledge base might seem simple enough. For example, suppose we want to apply the rule

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

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

Figure 4.23 A forward-chaining algorithm.

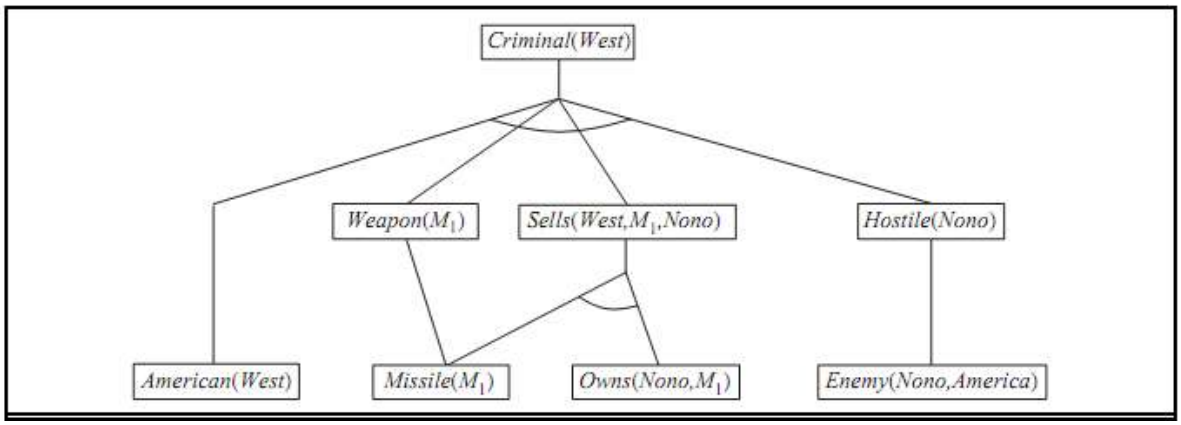


Figure 4.44 The proof tree generated by forward chaining on the crime example.

Then we need to find all the facts that unify with *Missile(x);* in a suitably indexed knowledge
base, this can be done in constant time per fact. Now consider a rule such as

Missile(x) ^Owns(Nono, x) ->Sells(West, x, Nono) .

Find all the objects owned by Nono in constant time per object; then, for each object, we could check whether it is a missile.

If the knowledge base contains many objects owned by Nono and very few missiles, however, it would be better to find all the missiles first and then check whether they are owned by Nono.

This is the **conjunct ordering** problem: find an ordering to solve the conjuncts of the rule premise so that the total cost is minimized.

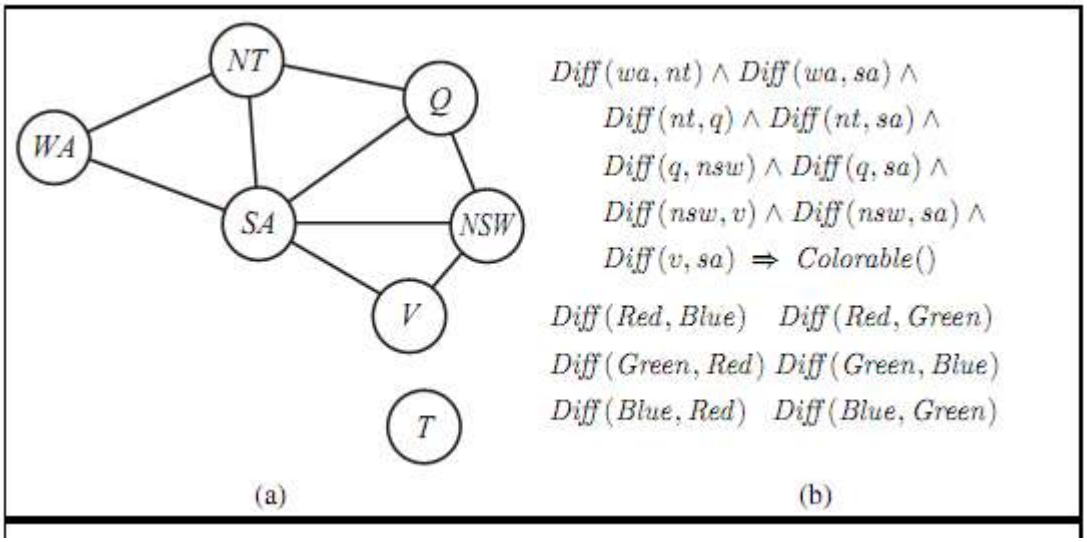


Figure 4.45 (a) Constraint graph for coloring the map of Australia. (b) The map-coloring CSP expressed as a single definite clause. Each map region is represented as a variable whose value can be one of the constants Red, Green or Blue.

4.12 Backward chaiining

Backward chaining algorithms work backward from the goal, chaining through rules to find known facts that support the proof.

A backward chaining algorithm

A simple backward-chaining algorithm, FOL-BC-ASK. It is called with a list of goals containing a single element, the original query, and returns the set of all substitutions satisfying the query.

The list of goals can be thought of as a "stack" waiting to be worked on; if all of them can be satisfied, then the current branch of the proof succeeds.

The algorithm takes the first goal in the list and finds every clause in the knowledge base whose positive literal, or **head,** unifies with the goal.

Each such clause creates a new recursive call in which the premise, or **body,** of the clause is added to the goal stack.

```
function FOL-BC-ASK(KB, query) returns a generator of substitutions
  return FOL-BC-OR(KB, query, { })

generator FOL-BC-OR(KB, goal, θ) yields a substitution
  for each rule (lhs ⇒ rhs) in FETCH-RULES-FOR-GOAL(KB, goal) do
    (lhs, rhs) ← STANDARDIZE-VARIABLES((lhs, rhs))
    for each θ' in FOL-BC-AND(KB, lhs, UNIFY(rhs, goal, θ)) do
      yield θ'

generator FOL-BC-AND(KB, goals, θ) yields a substitution
  if θ = failure then return
  else if LENGTH(goals) = 0 then yield θ
  else do
    first, rest ← FIRST(goals), REST(goals)
    for each θ' in FOL-BC-OR(KB, SUBST(θ, first), θ) do
      for each θ'' in FOL-BC-AND(KB, rest, θ') do
        yield θ''
```

Figure 4.46 A simple backward-chaining algorithm for first-order knowl edge bases.

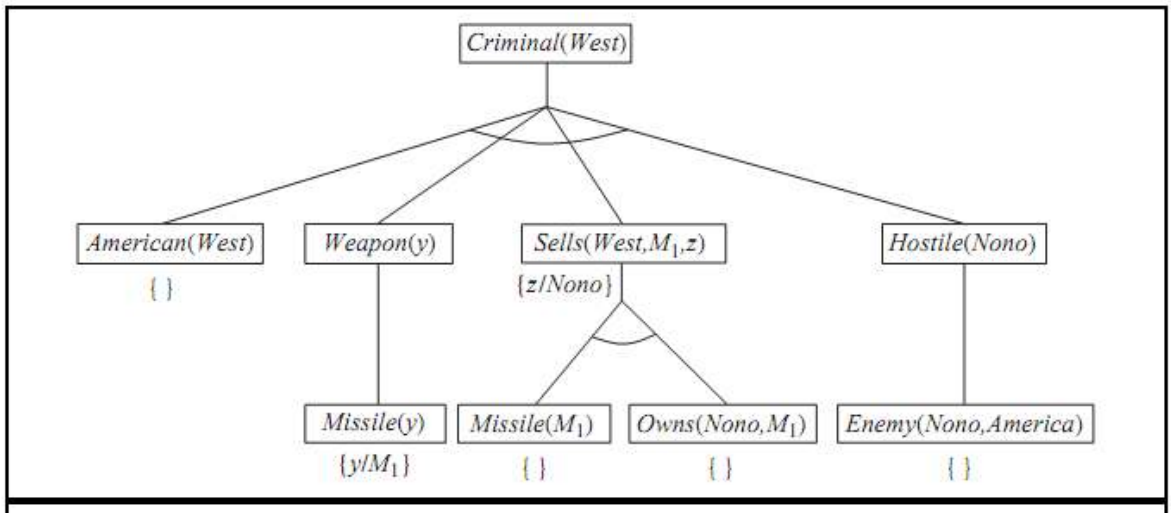


Figure 4.47 Proof tree constructed by backward chaining to prove that West is a criminal.

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 subgoal.

Note that once one subgoal in a conjunction succeeds, its substitution is applied to subsequent subgoals. Thus, by the time FOL- BC- ASK gets to the last conjunct, originally Hostile(z), z is already bound to Nono.

The algorithm uses **composition** of substitutions. COMPOSE(Θ1,Θ2), is the substitution whose effect is identical to the effect of applying each substitution in turn.

Logic programming

Logic programming is a technology that comes fairly close to embodying the declarative ideal should be constructed by expressing knowledge in a formal language and that problems should be solved by running inference processes on that knowledge.

***Algorithm = Logic + Control* .**

PROLOG

Prolog is the most widely used logic programming language. Its users number in the hundreds of thousands.

It is used primarily as a rapid-prototyping language and for symbol manipulation.Many expert systems have been written in Prolog for legal, medical, financial, and other domains.

Prolog programs are sets of definite clauses written in a notation somewhat 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 sepearate literals in the body, and a period marks the end of a sentence:

Prolog includes "syntactic sugar" for list notation and arithmetic.

Example , here is a Prolog program for append (X, Y, Z) , which succeeds if list Z is the result of appending lists X and Y:

The execution of Prolog programs is done via depth-first backward chaining, where clauses are tried in the order in which they are written in the knowledge base.

Aspects of Prolog fall outside standard logical inference:

There is a set of built-in functions for arithmetic. Literals using these function symbols are "proved by executing code rather than doing further inference.

There are built-in predicates that have side effects when executed. These include input output predicates and the assert/retract predicates for modifying the knowledge base. Such predicates have no counterpart in logic and can produce some confusing effects.

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 m o r n i n g s t a r = e v e n i n g s t a r fails.

The **occur check** is omitted from Prolog's unification algorithm. This means that some unsound inferences can be made; these are seldom a problem except when using Prolog for mathematical theorem proving.

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. We say "essentially," because Prolog interpreters contain a variety of improvements designed to maximize speed.

First, instead of constructing the list of all possible answers for each subgoal before continuing to the next, to perform interpretation. This promise is called a **choicepoint.**

```
procedure APPEND(ax, y, az, continuation)
    trail ← GLOBAL-TRAIL-POINTER()
    if ax = [] and UNIFY(y, az) then CALL(continuation)
    RESET-TRAIL(trail)
    a, x, z ← NEW-VARIABLE(), NEW-VARIABLE(), NEW-VARIABLE()
    if UNIFY(ax, [a | x]) and UNIFY(az, [a | z]) then APPEND(x, y, z, continuation)
```

Figure 4.24 Pseudocode representing the result of compiling the Append predicate.

Parallelization

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

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.

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. Each conjunctive branch must communicate with the other branches to ensure a global solution.

4.13 Resolution

Resolution can be extended to first-order logic. The question of the existence of complete proof procedures is of direct concern to mathematicians. If a complete proof procedure can be found for mathematical statements, two things follow:

first, all conjectures can be established mechanically;

second, all of mathematics can be established as the logical consequence of a set of fundamental axioms.

Conjunctive normal form for first-order logic

The first-order resolution requires that sentences be in **conjunctive normal form** (CNF) that is, a conjunction of clauses, where each clause is a disjunction ofliterals. Literals can contain variables, which are assumed to be universally quantified.

Forexample, the sentence

∀x American(x) ∧ Weapon(y) ∧ Sells(z, y, z) ∧ Hostile(z) ->Criminal (x)

becomes, in CNF,

¬American(x) ∨¬ Weapon(y) ∨¬ S e l l s (x , y, z) ∨¬Hostile(z) ∨¬Criminal (x) .

Every sentence of first-order logic can be converted into an inferentially equivalent CNF sentence.

The procedure for conversion to CNF has the following steps.We will illustrate the procedure by translating the sentence "Everyone who loves all animals is loved by someone," or ***∀x['∀y Animal(y) ->Loves(x, y)] ->[∃ y Loves(y, x)]***

The steps are as follows:

Eliminate implications:

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

Move ¬ inwards:

In addition to the usual rules for negated connectives, we need rulesfor negated quantifiers. Thus, we have

¬∀*x p* becomes ∃ *x* ¬*p*

¬∃*x p* becomes ∀*x* ¬*p* .

Our sentence goes through the following transformations:

$\forall x\ [\exists y \neg(\neg Anima1\ (Y) \vee Loves(x, y))] \vee [\exists y\ Loves(y, x)]$.

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

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

Notice how a universal quantifier (∀*y)* in the premise of the implication has become an existential quantifier.

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

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

Skolemize:

Skolemization is the process of removing existential quantifiers by elimination. It is just like the Existential Instantiation rule. translate∀ *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.

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 ∨ over ∧:

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

This step may also require flattening out nested conjunctions and disjunctions. The sentence is now in CNF and consists of two clauses

The resolution inference rule

The resolution rule for first-order clauses is simply a lifted version of the propositional resolution rule. Two clauses, which are assumed to be standardized apart so that they share no variables, can be resolved if they contain complementary literals.

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.

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

where $\text{UNIFY}(\ell_i, \neg m_j) = \theta$. 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 $\neg Loves(u, v)$, with unifier $\theta = \{u/G(x), v/x\}$, to produce the **resolvent** clause

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

The rule we have just given is the binary resolution rule, because it resolves exactly two literals.The binary resolution rule by itself does not yield a complete inference procedure.

Example proofs

Resolution proves that KB ├*a* by proving *KB* A*l a* unsatisfiable, i.e., by deriving the empty clause. The algorithmic approach is identical to the propositional case, described in

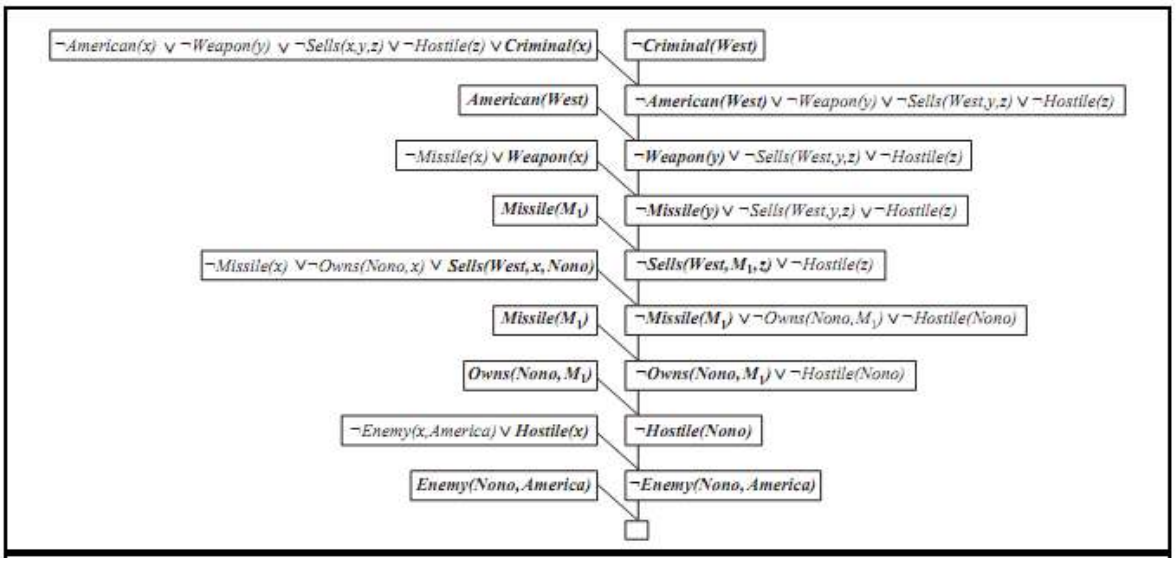


Figure 4.25 A resolution proof that West is a criminal. At each step, the literals that unify are in bold.

The first is the crime example from Section 9.3. The sentences in CNF are

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

This results in a somewhat more complex proof structure. In English, the problem is as follows:

Everyone who loves all animals is loved by someone.
Anyone who kills an animal is loved by no one.
Jack loves all animals.
Either Jack or Curiosity killed the cat, who is named Tuna.

Did Curiosity kill the cat?

First, we express the original sentences, some background knowledge, and the negated goal G in first-order logic:

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

B. $\forall x\ [\exists z\ \ Animal(z) \wedge Kills(x, z)] \Rightarrow [\forall y\ \neg Loves(y, x)]$

C. $\forall x\ \ Animal(x) \Rightarrow Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\forall x\ \ Cat(x) \Rightarrow Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

Now we apply the conversion procedure to convert each sentence to CNF:

A1. $Animal(F(x)) \vee Loves(G(x), x)$

A2. $\neg Loves(x, F(x)) \vee Loves(G(x), x)$

B. $\neg Loves(y, x) \vee \neg Animal(z) \vee \neg Kills(x, z)$

C. $\neg Animal(x) \vee Loves(Jack, x)$

D. $Kills(Jack, Tuna) \vee Kills(Curiosity, Tuna)$

E. $Cat(Tuna)$

F. $\neg Cat(x) \vee Animal(x)$

¬G. $\neg Kills(Curiosity, Tuna)$

The resolution proof that Curiosity killed the cat is given in Figure. In English,, the proof could be paraphrased as follows:

Suppose Curiosity did not kill Tuna.

We know that either Jack or Curiosity did; thus Jack must have. Novu, Tuna is a cat and cats are animals, so Tuna is an animal.

Because anyone who kills an animal is loved by no one, we know that no one loves Jack.

On the other hand, Jack lovles all animals, so someone loves him; so we have a contradiction.

Therefore, Curiosity killed the cat.

The proof answers the question "Did Curiosity kill the cat?" but often we want to pose more general questions, such as "Who killed the cat?"

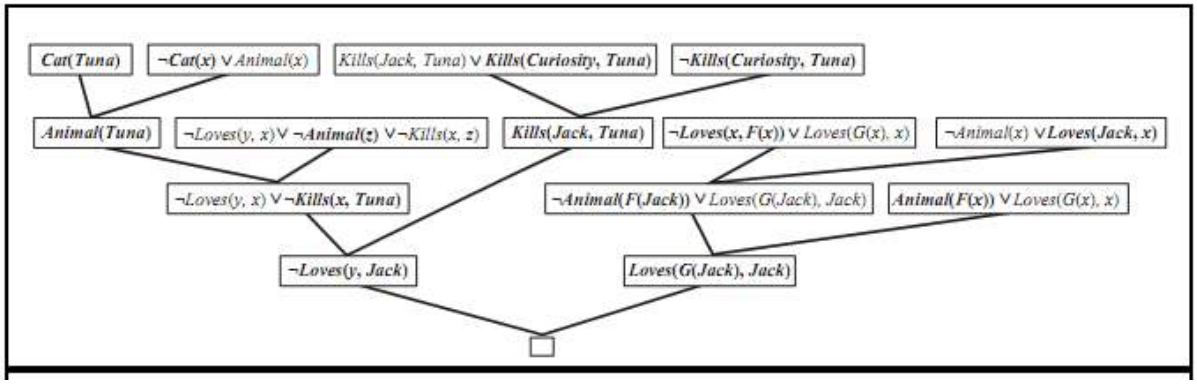


Figure 9.12 A resolution proof that Curiosity killed the cat.

Completeness of resolution

A resolution is **refutation-complete,** which means that if a set of sentences is unsatisfiable, then resolution will always be able to derive a contradiction.

Resolution cannot be used to generate all logical consequences of a set of sentences, but it can be used to establish that a given sentence is entailed by the set of sentences. Hence, it can be used to find all answers to a given question, using the negated-goal method.

Our goal therefore is to prove the following: if S ***is an unsatisjable set of clauses, then the applicationof a finite number of resolution steps to S will yield a contradiction.***

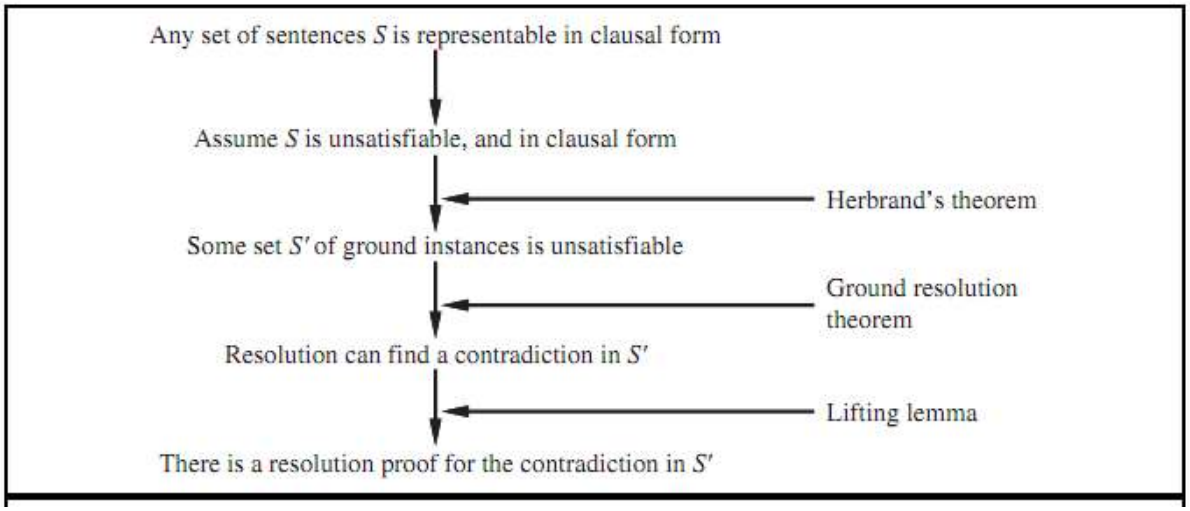


Figure 9.13 Structure of a completeness proof for resolution.

The basic structure of the proof is shown in Figure it proceeds as follows:

1. First, we observe that if S is unsatisfiable, then there exists a particular set of ***ground instances*** of the clauses of S such that this set is also unsatisfiable (Herbrand's theorem).
2. We then appeal to the **ground resolution theorem**, which states that propositional resolution is complete for ground sentences.
3. We then use a **lifting lemma** to show that, for any propositional resolution proof using the set of ground sentences, there is a corresponding first-order resolution proof using the first-order sentences from which the ground sentences were obtained.

To carry out the first step, we need three new concepts:

Herbrand universe:

If S is a set of clauses, then *Hs* ,theHerbrand universe of S, is the set of all ground terms constructible from the following:

a. The function symbols in S, if any.

b. The constant symbols in S, if any; if none, then the constant symbol A.

For example, if S contains just the clause

$$\neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B).$$

Then Hs is the following infinite set of ground terms:

{A, B,F(A,A), F(A, B), F(B, A), F(B, B), F(A, F(A, A)), . . .) .

Saturation:

If S is a set of clauses and P is a set of ground terms, then P(S), the saturation of S with respect to P , is the set of all ground clauses obtained by applying all possible consistent substitutions of ground terms in P with variables in S.

Herbrand base:

The saturation of a set S of clauses with respect to its Herbrand universe is called the Herbrand base of S, written as Hs(S). For example, if *S* contains solely the clause just given, then Hs(S) is the infinite set of clauses

$$\{\neg P(A, F(A, A)) \vee \neg Q(A, A) \vee R(A, B),$$
$$\neg P(B, F(B, A)) \vee \neg Q(B, A) \vee R(B, B),$$
$$\neg P(F(A, A), F(F(A, A), A)) \vee \neg Q(F(A, A), A) \vee R(F(A, A), B),$$
$$\neg P(F(A, B), F(F(A, B), A)) \vee \neg Q(F(A, B), A) \vee R(F(A, B), B), \ldots\}$$

LIFTING LEMMA

Let C1 and C2 be two clauses with no shared variables, and let C1' and C2' be ground instances of C1 and C2. If C' is a resolvent of C1 and C2, then there exists a clause C such that (1) C is a resolvent of Cl and C2 and (2) C' is a ground instance of C. This is called a lifting lemma

We simply illustrate the lemma:

$$C_1 = \neg P(x, F(x, A)) \vee \neg Q(x, A) \vee R(x, B)$$
$$C_2 = \neg N(G(y), z) \vee P(H(y), z)$$
$$C_1' = \neg P(H(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B)$$
$$C_2' = \neg N(G(B), F(H(B), A)) \vee P(H(B), F(H(B), A))$$
$$C' = \neg N(G(B), F(H(B), A)) \vee \neg Q(H(B), A) \vee R(H(B), B)$$
$$C = \neg N(G(y), F(H(y), A)) \vee \neg Q(H(y), A) \vee R(H(y), B) .$$

From this fact, it follows that if the empty clause appears in the resolution closure of *St,* it must also appear in the resolution closure of S. This is because the empty clause cannot be a ground instance of any other clause. To recap: we have shown that if S is unsatisfiable, then there is a finite derivation of the empty clause using the resolution rule.

The lifting of theorem proving from ground clauses to first-order clauses provides a vast increase in power. This increase comes from the fact that the first-order proof need instantiate variables only as far as necessary for the proof, whereas the ground-clause methods.

Dealing with equality

There are three distinct approaches that can be taken.

The first approach is to **axiomatize equality**-to write down sentences about the equality relation in the knowledge base.

We need to say that equality is reflexive, symmetric, and transitive, and we also have to say that we can substitute equals for equals in any predicate or function. So we need three basic axioms, and then one for each predicate and function:

Demodulationdeal with equality with an additional inference rule. The simplest rule, *demodulation,* takes a unit clause *x* = *y* and substitutes *y* for any term that unifies with *x* in some other clause. For any terms *x, y,* and *z,* where *UNIFY(XZ,)* = **Θ** and *mn[zj*is a literal containing *z:*

$$\frac{x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), m_1 \vee \cdots \vee m_n)} .$$

$$\forall x \;\; x = x$$
$$\forall x, y \;\; x = y \;\Rightarrow\; y = x$$
$$\forall x, y, z \;\; x = y \wedge y = z \;\Rightarrow\; x = z$$

$$\forall x, y \;\; x = y \;\Rightarrow\; (P_1(x) \;\Leftrightarrow\; P_1(y))$$
$$\forall x, y \;\; x = y \;\Rightarrow\; (P_2(x) \;\Leftrightarrow\; P_2(y))$$
$$\vdots$$
$$\forall w, x, y, z \;\; w = y \wedge x = z \;\Rightarrow\; (F_1(w, x) = F_1(y, z))$$
$$\forall w, x, y, z \;\; w = y \wedge x = z \;\Rightarrow\; (F_2(w, x) = F_2(y, z))$$
$$\vdots$$

Paramodulation: For any terms *x, y,* and *z,* where *UNIFY(x, z,) = Θ*

$$\frac{\ell_1 \vee \cdots \vee \ell_k \vee x = y, \qquad m_1 \vee \cdots \vee m_n}{\text{SUB}(\text{SUBST}(\theta, x), \text{SUBST}(\theta, y), \text{SUBST}(\theta, \ell_1 \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_n)}.$$

A third approach handles equality reasoning entirely within **an extended unification algorithm.**
That is, terms are unifiable if they are *provably* equal under some substitution, where "provably" allows for some amount of equality reasoning.

Resolution Strategies:

Unit preference

This strategy prefers to1 do resolutions where one of the sentences is a single literal (also known as a **unit clause).** The idea behind the strategy is that we are trying to produce an empty clause, so it might be a good idea to prefer inferences that produce shorler clauses.

Resolving a unit sentence (such as P) with any other sentence (such as ⌐P V ⌐Q V R) always yields a clause (in this case, ⌐Q V R) that is shorter than the other clause.

Unit resolution is a restricted form of resolution in which every resolution step must involve a unit clause.

Set of support

The set-of-support strategy does just that. It starts by identifying a subset of the sentences called the **set of support.** Every resolution combines a sentence from the set of support with another sentence anld adds the resolvent into the set of support. If the set of support is small relative to the whole knowledge base, the search space will be reduced dramatically.

Input resolution

In the **input resolution** strategy, every resolution combines one of the input sentences (from the KB or the query) with some other sentence. The proof uses only input resolutions and has the characteristic shape of a single "spine" with single sentences combining onto the spine.

Subsumption

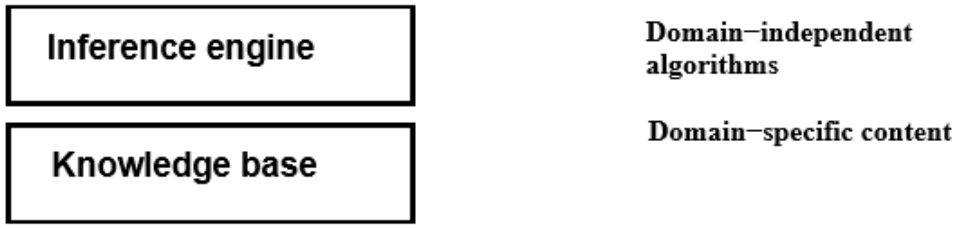
The **subsumption** method eliminates all sentences that are subsumed by (i.e., more specific than) an existing sentence in the KB. For example, if *P(x)* is in the KB, then there is no sense in adding *P(A)* and even less sense in adding *P(A)* V *Q(B).* Subsumption helps keep the KB small, and thus helps keep the search space small.

UNIT V

KNOWLEDGE REPRESENTATION

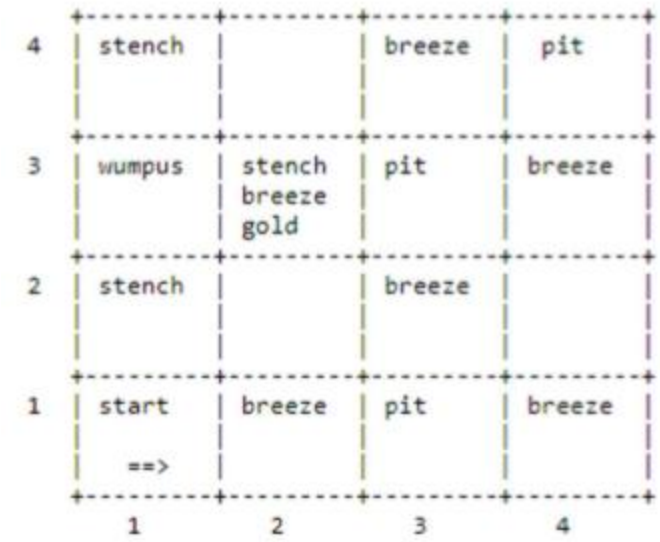
Knowledge Based Agents A knowledge-based agent needs a KB and an inference mechanism. It operates by storing sentences in its knowledge base, inferring new sentences with the inference mechanism, and using them to deduce which actions to take. ... The interpretation of a sentence is the fact to which 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,(A∧(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 Satisfiability is connected to inference via the following: KB|=α iff(KB∧¬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: C∧(B ⇒ A) ∧ (C∧D ⇒ B) Modus Ponens (for Horn Form): complete for Horn KBs

α1,...,αn,α1∧⋯∧α⇒ β β

Canbeusedwithforwardchaining orbackwardchaining. These algorithms areverynaturalandruninlineartime.,

ForwardChaining

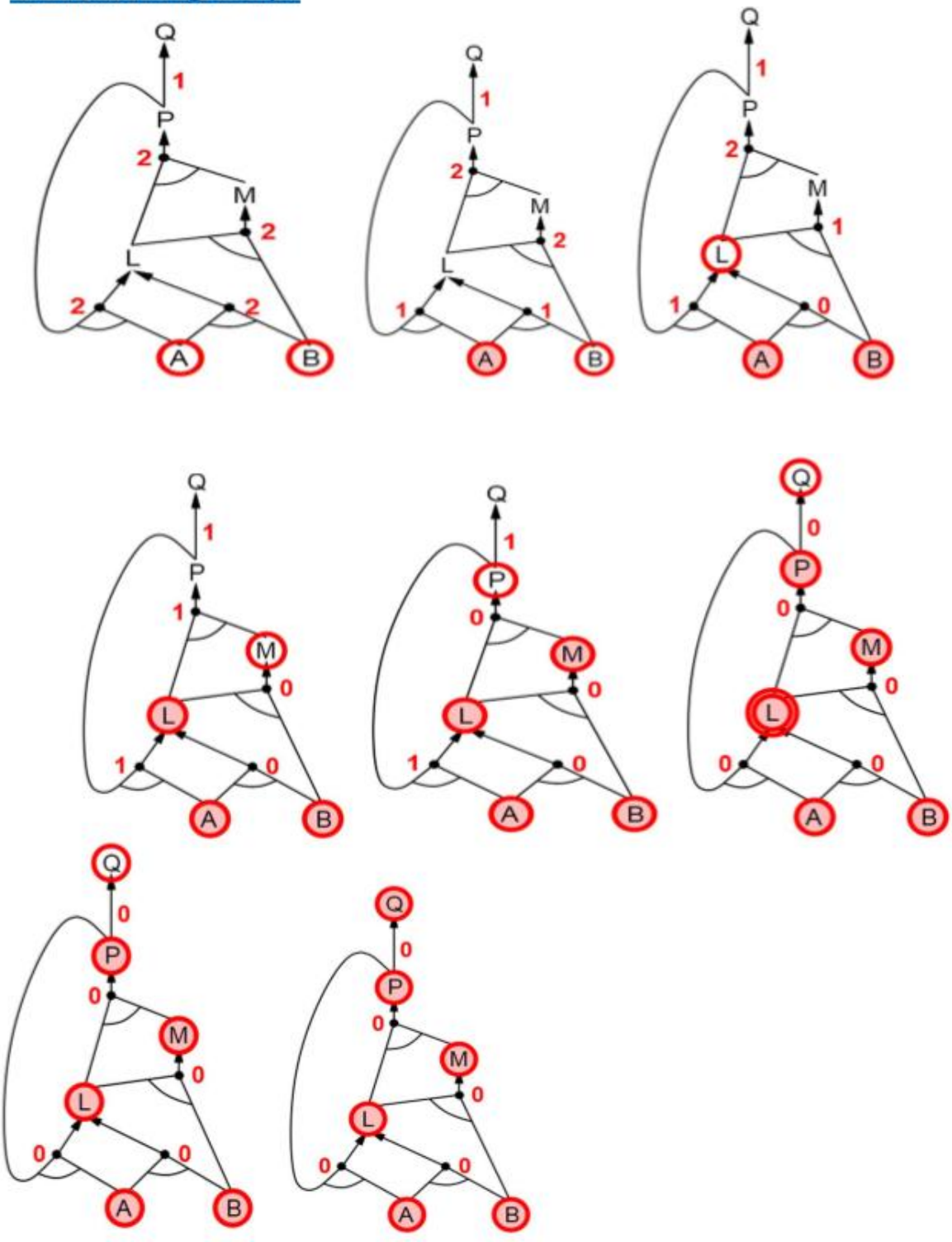
Idea: If anyrulewhosepremisesaresatisfiedintheKB, additsconclusiontotheKB,untilqueryisfound

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

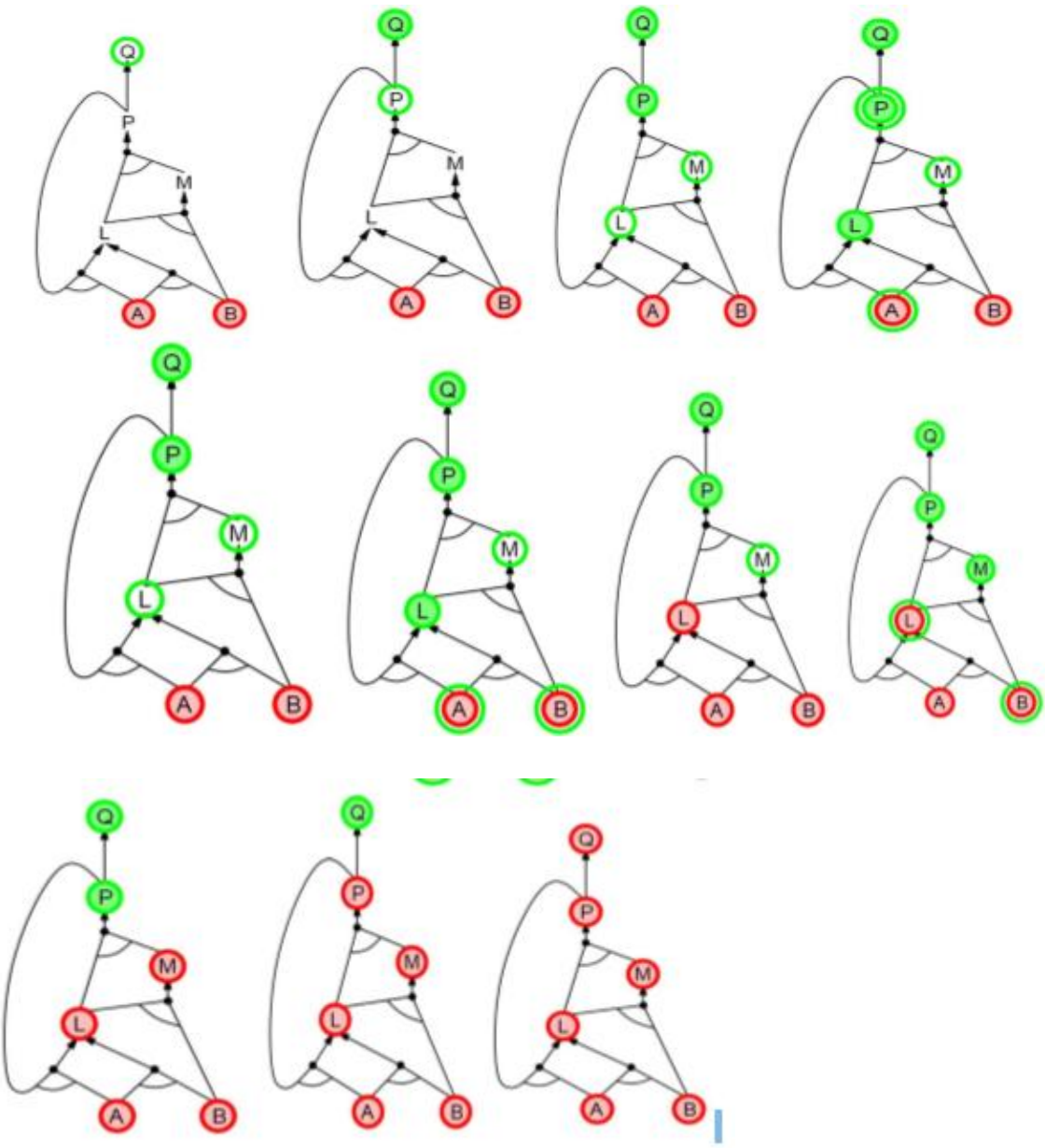
1. FCreachesafixedpointwherenonewatomicsentencesarederived

2. Considerthefinalstateasamodelm,assigningtrue/falsetosymbols

3. Every clause in the original KB is true inm i. Proof:Supposeaclausea1∧...∧ak⇒bisfalsei nm Then a1∧. . . ∧akis 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 “S1,4∧ 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⇔ (P1,2 ∨ 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

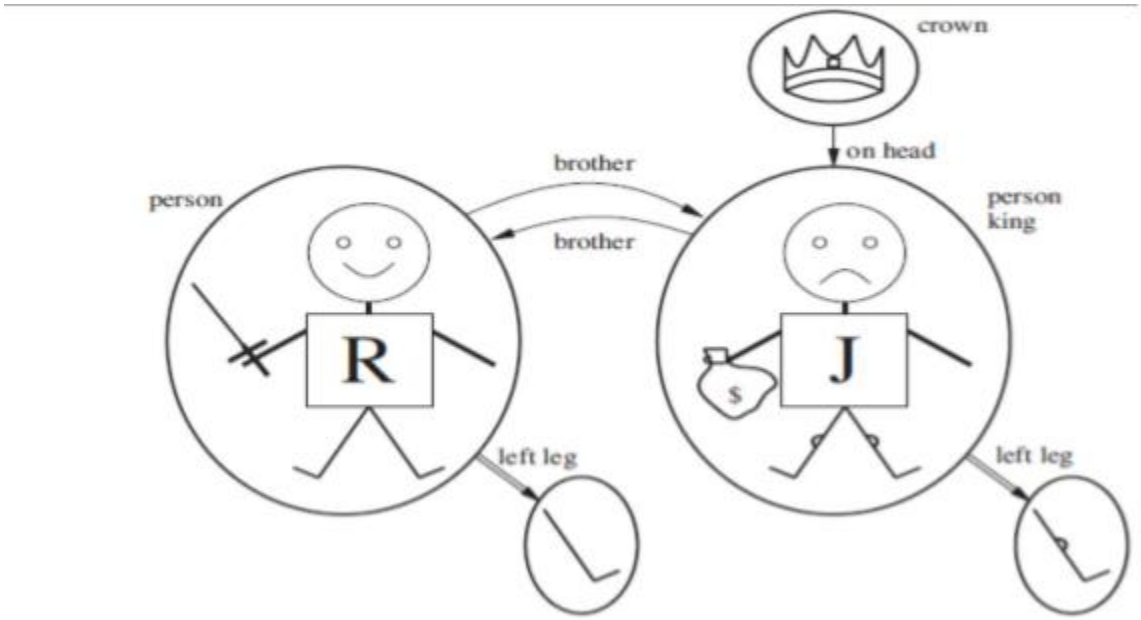
Models for first-order logic :

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 betweenthe 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
∀xKing(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) ∧OnHead(x, John)

Given assertions:

Richard the Lionheart is a crown ∧Richard the Lionheart is on John's head; King John is a crown ∧King John is on John's head; Richard's left leg is a crown ∧Richard's left leg is on John's head; John's left leg is a crown ∧John's left leg is on John's head; The crown is a crown ∧the 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 ∀, ∧is 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) ∧Brother (y,Richard) ∧¬(x=y) .

The sentence

∃x, y Brother (x,Richard) ∧Brother (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.

$$
\begin{aligned}
\textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\
\textit{AtomicSentence} &\rightarrow \textit{Predicate} \mid \textit{Predicate}(\textit{Term}, \ldots) \mid \textit{Term} = \textit{Term} \\
\textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\
&\mid \neg \textit{Sentence} \\
&\mid \textit{Sentence} \wedge \textit{Sentence} \\
&\mid \textit{Sentence} \vee \textit{Sentence} \\
&\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\
&\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \\
&\mid \textit{Quantifier}\ \textit{Variable}, \ldots\ \textit{Sentence} \\
\textit{Term} &\rightarrow \textit{Function}(\textit{Term}, \ldots) \\
&\mid \textit{Constant} \\
&\mid \textit{Variable} \\
\textit{Quantifier} &\rightarrow \forall \mid \exists \\
\textit{Constant} &\rightarrow A \mid X_1 \mid \textit{John} \mid \cdots \\
\textit{Variable} &\rightarrow a \mid x \mid s \mid \cdots \\
\textit{Predicate} &\rightarrow \textit{True} \mid \textit{False} \mid \textit{After} \mid \textit{Loves} \mid \textit{Raining} \mid \cdots \\
\textit{Function} &\rightarrow \textit{Mother} \mid \textit{LeftLeg} \mid \cdots \\
\text{OPERATOR PRECEDENCE} &: \neg, =, \wedge, \vee, \Rightarrow, \Leftrightarrow
\end{aligned}
$$

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) ∧Parent(m, c) .

2. One's husband is one's male spouse: ∀w, h Husband(h,w) ⇔Male(h) ∧Spouse(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) ∧Parent(p, c) .

6. A sibling is another child of one's parents: ∀x, y Sibling(x, y) ⇔x _= y ∧∃p Parent(p, x) ∧Parent(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) . ∀n NatNum(n) ⇒ 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) ∧ 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) ∧ 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 ∪ 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: ∀s Set(s) ⇔(s={}) ∨(∃x, s2 Set(s2) ∧s={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} ∧(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 ∧s2 ⊆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 ∧x∈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 ∪s2) ⇔(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) ,

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 ∧(y = b − 1 ∨y = b + 1)) ∨(y = b ∧(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) ∧At(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) ∧Breeze(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

$\forall s$ Breezy(s) $\Rightarrow \exists r$ Adjacent (r, s)$\wedge$Pit(r) ,

and that if a square is not breezy, no adjacent square contains a pit: $\forall s \neg$Breezy (s) $\Rightarrow \neg \exists r$ Adjacent (r, s) $\wedge$ Pit (,r) .Combining these two, we obtain the biconditional sentence $\forall s$ Breezy (s)$\Leftrightarrow \exists r$ Adjacent(r, s) $\wedge$ 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: $\forall s[\forall r$ Adjacent (r, s) $\Rightarrow \neg$Pit (r)] $\Rightarrow \neg$Breezy (s) .

It is possible to show that these two sentences together are logically equivalent to the biconditional sentence " $\forall s$ Breezy (s)$\Leftrightarrow \exists r$ Adjacent(r, s) $\wedge$ 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 θto 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

$$\exists x \ \ Crown(x) \wedge OnHead(x, John)$$

So, we can infer the sentence

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

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 oldin 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

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

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

2. 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 Generalized 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,

$$\frac{p1',\ p2',\ \ldots,\ pn',\ (p1 \wedge p2 \wedge \ldots \wedge pn \Rightarrow q)}{SUBST(\theta, q)}$$

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:

$$\forall y\ Greedy(y)$$

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**.**

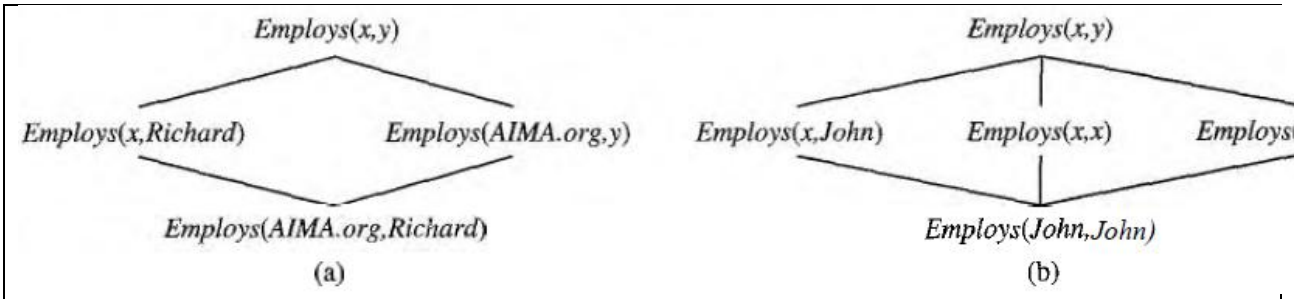


Figure 2.2 *(a) The subsumption lattice whose lowest node is the sentence Employs (AIMA.org, Richard). (b) The subsumption lattice for the sentence Employs (John, John).*

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.

3. 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, α) returns a substitution or false
  inputs: KB, the knowledge base, a set of first-order definite clauses
          α, 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 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-APART(r)
          for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ 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', α)
                  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.*

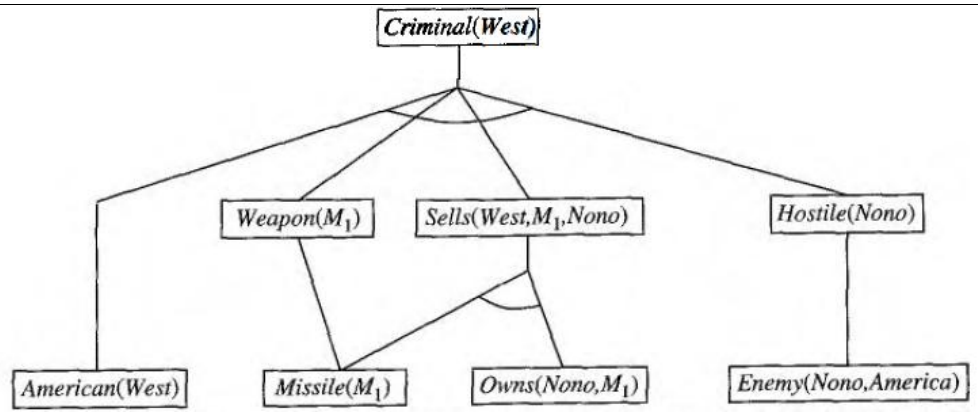


Figure 3.2 *The proof tree generated by forward chaining on the crime example. The initial facts appear at the bottom level, facts inferred on the first iteration in the middle level, and facts inferred on the second iteration at the top level.*

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 everyfinite-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 leastone 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 {θ}
  q' ← SUBST(θ, FIRST(goals))
  for each sentence r in KB where STANDARDIZE-APART(r) = (p1 ∧ ... ∧ pn ⇒ q')
        and θ' ← UNIFY(q, q') succeeds
      new-goals ← [p1, ..., pn | REST(goals)]
      answers ← FOL-BC-ASK(KB, new-goals, COMPOSE(θ', θ)) ∪ answers
  return answers
```

Figure 4.1*A simple backward-chaining algorithm.*

.

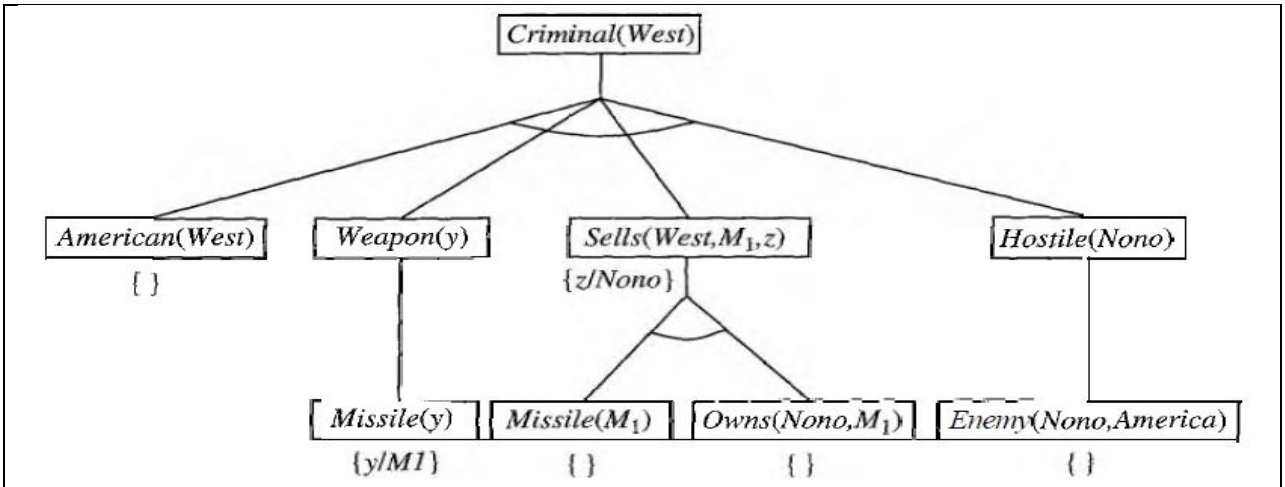


Figure 4.2 *Proof tree constructed by backward chaining to prove that West is a criminal. The tree should be read depth first, left to right. To prove Criminal (West), we have to prove the four conjuncts below it. Some of these are in the knowledge base, and others require further backward chaining. 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'scontinuation 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

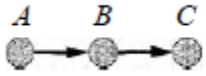
1. The first, called **OR-parallelism,** comes from the possibility of a goal unifying with many different clauses in the knowledge base. Each gives rise to an independent branch in the search space that can lead to a potential solution, and all such branches can be solved in parallel.
2. The second,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

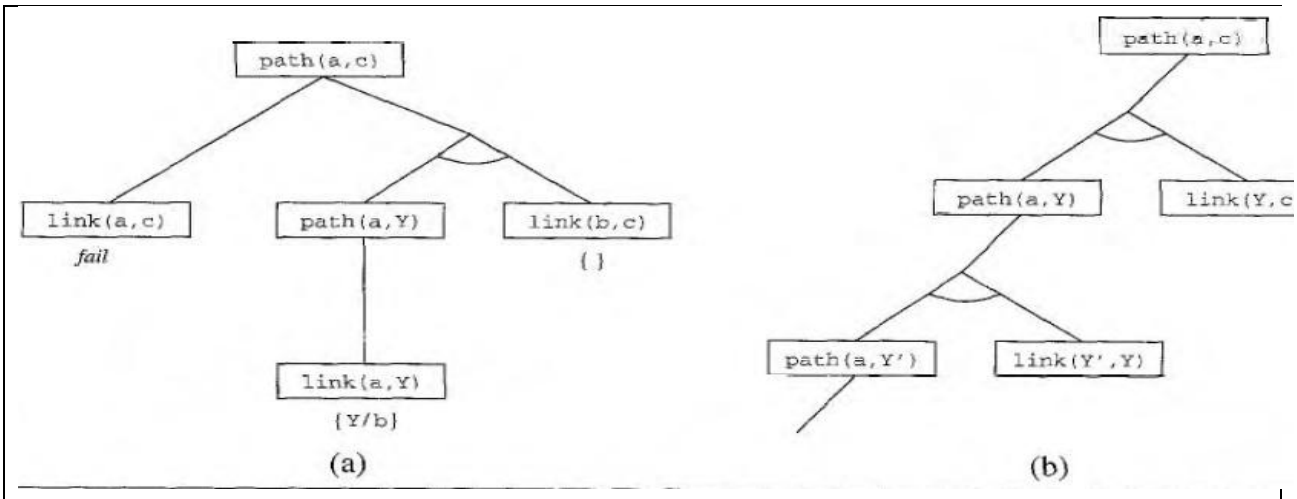


Figure 4.3 (a) Proof that a path exists from A to C. (b) Infinite proof tree generated when the clauses are in the "wrong" order.

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 ofliterals.

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:

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

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

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

Our sentence goes through the following transformations:

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

- Standardize variables: For sentences like $(\forall x \ P(x)) \ V \ (3x \ Q(x))$ 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

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

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:*

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

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;

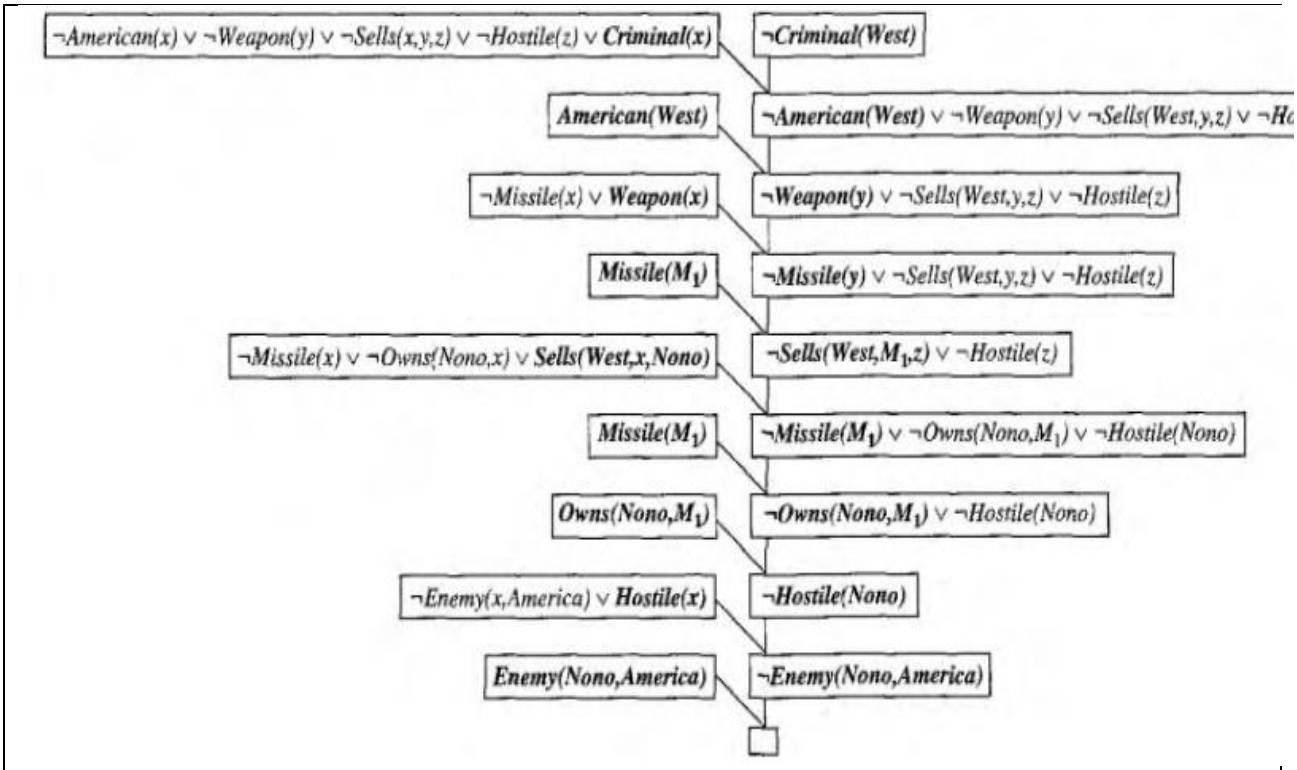


Figure 5.1 A resolution proof that West is a criminal.

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.

Planning Classical Planning: AI as the study of rational action, which means that planning—devising a plan of action to achieve one's goals—is a critical part of AI. We have seen two examples of planning agents so far the search-based problem-solving agent.

DEFINITION OF CLASSICAL PLANNING: The problem-solving agent 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 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,whichmeansthatitmaybeswampedwhentherearemanyactionsandstates.For example,intheworld,thesimpleactionofmovingastepforwardhadtoberepeated for all four agent orientations, T time steps, and n^2 currentlocations.

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 $4Tn^2$ 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. 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 flaunts 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) ∧ At(Truck 2, Sydney). Database semantics is used: the

closed-world assumption means that any flaunts that are not mentioned are false, and the unique names assumption means that Truck 1 and Truck 2 are distinct.

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.

A set of action schemas serves as a definition of a planning domain. A specific problem within the domain is defined with the addition of an initial state and a goal.

state is a conjunction of ground atoms. (As with all states, the closed-world assumption is used, which means that any atoms that are not mentioned are false.) The goal is just like a precondition: a conjunction of literals (positive or negative) that may contain variables, such as At(p, SFO) ∧ Plane(p). Any variables are treated as existentially quantified, so this goal is to have any plane at SFO. The problem is solved when we can find a sequence of actions that end in a states that entails the goal.

Example: Air cargo transport

An air cargo transport problem involving loading and unloading cargo and flying it from place to place. The problem can be defined with three actions: Load , Unload , and Fly . The actions affect two predicates: In(c, p) means that cargo c is inside plane p, and At(x, a) means that object x (either plane or cargo) is at airport a. Note that some care must be taken to make sure the At predicates are maintained properly. When a plane flies from one airport to another, all the cargo inside the plane goes with it. In first-order logic it would be easy to quantify over all objects that are inside the plane. But basic PDDL does not have a universal quantifier, so we need a different solution. The approach we use is to say that a piece of cargo ceases to be At anywhere when it is In a plane; the cargo only becomes At the new airport when it is unloaded. So At really means “available for use at a given location.”

The complexity of classical planning :

We consider the theoretical complexity of planning and distinguish two decision problems. PlanSAT is the question of whether there exists any plan that solves a planning problem. Bounded PlanSAT asks whether there is a solution of length k or less; this can be used to find an optimal plan.

The first result is that both decision problems are decidable for classical planning. The proof follows from the fact that the number of states is finite. But if we add function symbols to the language, then the number of states becomes infinite, and PlanSAT becomes only semi decidable: an algorithm exists that will terminate with the correct answer for any solvable problem, but may not terminate on unsolvable problems. The Bounded PlanSAT problem remains decidable even in the presence of function symbols.

Both PlanSAT and Bounded PlanSAT are in the complexity class PSPACE, a class that is larger (and hence more difficult) than NP and refers to problems that can be solved by a deterministic Turing machine with a polynomial amount of space. Even if we make some rather severe restrictions, the problems remain quite difficult.

Algorithms for Planning with State Space Search

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 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 nodes.

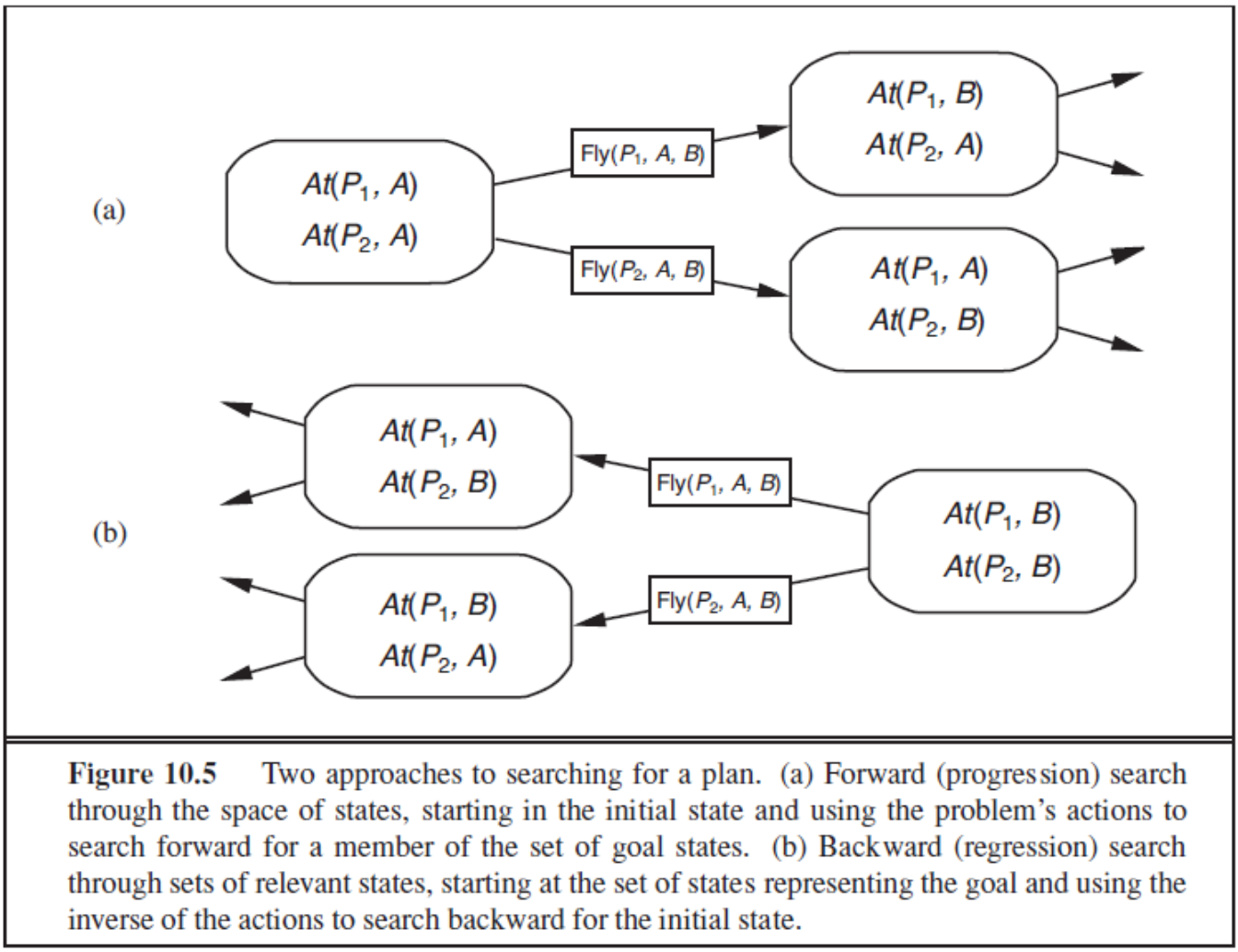


Figure 10.5 Two approaches to searching for a plan. (a) Forward (progression) search through the space of states, starting in the initial state and using the problem's actions to search forward for a member of the set of goal states. (b) Backward (regression) search through sets of relevant states, starting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

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 ∧ 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 flaunts in a domain, then there are 2n ground states (each fluent can be true or false), but 3n 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.

The final issue is deciding which actions are candidates to regress over. In the forward direction 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.

Heuristics for planning:

Neither forward nor backward search is efficient without a good heuristic function. Recall from Chapter 3 that a heuristic function h(s) estimates the distance from a state s to the goal and that if we can derive an admissible heuristic for this distance—one that does not overestimate—then we can use A∗ search to find optimal solutions. An admissible heuristic can be derived by defining a relaxed problem that is easier to solve. The exact cost of a solution to this easier problem then becomes the heuristic for the original problem.

By definition, there is no way to analyze an atomic state, and thus it it requires some ingenuity by a human analyst to define good domain-specific heuristics for search problems with atomic states. Planning uses a factored representation for states and action schemas. That makes it possible to define good domain-independent heuristics and for programs to automatically apply a good domain-independent heuristic for a given problem.

Planning Graphs:

All of the heuristics we have suggested can suffer from inaccuracies. This section shows how a special data structure called a planning graph can be used to give better heuristic estimates. These heuristics can be applied to any of the search techniques we have seen so far. Alternatively, we can search for a solution over the space formed by the planning graph, using an algorithm called GRAPHPLAN.

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 successors, and so on. If we indexed this tree appropriately, we could answer the planning question “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.

We now define mutex links for both actions and literals. A mutex relation holds between two actions at a given level if any of the following three conditions holds:

- Inconsistent effects: one action negates an effect of the other. For example, Eat(Cake) and the persistence of Have(Cake) have inconsistent effects because they disagree on the effect Have(Cake).
- Interference: one of the effects of one action is the negation of a precondition of the other. For example Eat(Cake) interferes with the persistence of Have(Cake) by its precondition.
- Competing needs: one of the preconditions of one action is mutually exclusive with a precondition of the other. For example, Bake(Cake) and Eat(Cake) are mutex because they compete on the value of the Have(Cake) precondition.

A mutex relation holds between two literals at the same level if one is the negation of the other or if each possible pair of actions that could achieve the two literals is mutually exclusive. This condition is called inconsistent support. For example, Have(Cake) and Eaten(Cake) are mutex in S1 because the only way of achieving Have(Cake), the persistence action, is mutex with the only way of achieving Eaten(Cake), namely Eat(Cake). In S2 the two literals are not mutex, because there are new ways of achieving them, such as Bake(Cake) and the persistence of Eaten(Cake), that are not mutex.

other Classical Planning Approaches:

Currently the most popular and effective approaches to fully automated planning are:

- Translating to a Boolean satisfiability (SAT) problem
- Forward state-space search with carefully crafted heuristics
- Search using a planning graph (Section 10.3)

These three approaches are not the only ones tried in the 40-year history of automated planning. Figure 10.11 shows some of the top systems in the International Planning Competitions, which have been held every even year since 1998. In this section we first describe the translation to a satisfiability problem and then describe three other influential approaches: planning as first-order logical deduction; as constraint satisfaction; and as plan refinement.

Classical planning as Boolean satisfiability :

we saw how SATPLAN solves planning problems that are expressed in propositional logic. Here we show how to translate a PDDL description into a form that can be processed by SATPLAN. The translation is a series of straightforward steps:

• Proposition Alize the actions: replace each action schema with a set of ground actions formed by substituting constants for each of the variables. These ground actions are not part of the translation, but will be used in subsequent steps.

• Define the initial state: assert F 0 for every fluent F in the problem's initial state, and ¬F for every fluent not mentioned in the initial state.

• Proposition Alize the goal: for every variable in the goal, replace the literals that contain the variable with a disjunction over constants. For example, the goal of having block A on another block, On(A, x) ∧ Block (x) in a world with objects A, B and C, would be replaced by the goal

(On(A, A) ∧ Block (A)) ∨ (On(A, B) ∧ Block (B)) ∨ (On(A, C) ∧ Block (C)) .

• Add successor-state axioms: For each fluent F , add an axiom of the form

F t+1 ⇔ ActionCausesF t ∨ (F t ∧ ¬ActionCausesNotF t) ,

where Action CausesF is a disjunction of all the ground actions that have F in their add list, and Action CausesNotF is a disjunction of all the ground actions that have F in their delete list.

Analysis of Planning approaches:

Planning combines the two major areas of AI we have covered so far: search and logic. A planner can be seen either as a program that searches for a solution or as one that (constructively) proves the existence of a solution. The cross-fertilization of ideas from the two areas has led both to improvements in performance amounting to several orders of magnitude in the last decade and to an increased use of planners in industrial applications. Unfortunately, we do not yet have a clear understanding of which techniques work best on which kinds of problems. Quite possibly, new techniques will emerge that dominate existing methods.

Planning is foremost an exercise in controlling combinatorial explosion. If there are n propositions in a domain, then there are 2n states. As we have seen, planning is PSPACE- hard. Against such pessimism,

the identification of independent sub problems can be a powerful weapon. In the best case—full decomposability of the problem—we get an exponential speedup.

Decomposability is destroyed, however, by negative interactions between actions. GRAPHPLAN records mutexes to point out where the difficult interactions are. SATPLAN rep- resents a similar range of mutex relations, but does so by using the general CNF form rather than a specific data structure. Forward search addresses the problem heuristically by trying to find patterns (subsets of propositions) that cover the independent sub problems. Since this approach is heuristic, it can work even when the sub problems are not completely independent.

Sometimes it is possible to solve a problem efficiently by recognizing that negative interactions can be ruled out. We say that a problem has serializable sub goals if there exists an order of sub goals such that the planner can achieve them in that order without having to undo any of the previously achieved sub goals. For example, in the blocks world, if the goal is to build a tower (e.g., A on B, which in turn is on C, which in turn is on the Table, as in Figure 10.4 on page 371), then the sub goals are serializable bottom to top: if we first achieve C on Table, we will never have to undo it while we are achieving the other sub goals. Planners such as GRAPHPLAN, SATPLAN, and FF have moved the field of planning forward, by raising the level of performance of planning systems.

Planning and Acting in the Real World:

This allows human experts to communicate to the planner what they know about how to solve the problem. Hierarchy also lends itself to efficient plan construction because the planner can solve a problem at an abstract level before delving into details. Presents agent architectures that can handle uncertain environments and interleave deliberation with execution, and gives some examples of real-world systems.

Time, Schedules, and Resources:

The classical planning representation talks about what to do, and in what order, but the representation cannot talk about time: how long an action takes and when it occurs. For example, the planners of Chapter 10 could produce a schedule for an airline that says which planes are assigned to which flights, but we really need to know departure and arrival times as well. This is the subject matter of scheduling. The real world also imposes many resource constraints; for example, an airline has a limited number of staff—and staff who are on one flight cannot be on another at the same time. This section covers methods for representing and solving planning problems that include temporal and resource constraints.

The approach we take in this section is “plan first, schedule later”: that is, we divide the overall problem into a planning phase in which actions are selected, with some ordering constraints, to meet the goals of the problem, and a later scheduling phase, in which temporal information is added to the plan to ensure that it meets resource and deadline constraints.

$$Jobs(\{AddEngine1 \prec AddWheels1 \prec Inspect1\}, \{AddEngine2 \prec AddWheels2 \prec Inspect2\})$$

$$Resources(EngineHoists(1), WheelStations(1), Inspectors(2), LugNuts(500))$$

$$Action(AddEngine1, \text{DURATION:}30, \text{USE:}EngineHoists(1))$$
$$Action(AddEngine2, \text{DURATION:}60, \text{USE:}EngineHoists(1))$$
$$Action(AddWheels1, \text{DURATION:}30, \text{CONSUME:}LugNuts(20), \text{USE:}WheelStations(1))$$
$$Action(AddWheels2, \text{DURATION:}15, \text{CONSUME:}LugNuts(20), \text{USE:}WheelStations(1))$$
$$Action(Inspect_i, \text{DURATION:}10, \text{USE:}Inspectors(1))$$

Figure 11.1 A job-shop scheduling problem for assembling two cars, with resource constraints. The notation $A \prec B$ means that action A must precede action B.

This approach is common in real-world manufacturing and logistical settings, where the planning phase is often performed by human experts. The automated methods of Chapter 10 can also be used for the planning phase, provided that they produce plans with just the minimal ordering constraints required for correctness. G RAPHPLAN (Section 10.3), SATPLAN (Section 10.4.1), and partial-order planners (Section 10.4.4) can do this; search-based methods (Section 10.2) produce totally ordered plans, but these can easily be converted to plans with minimal ordering constraints.

Hierarchical Planning :

The problem-solving and planning methods of the preceding chapters all operate with a fixed set of atomic actions. Actions can be strung together into sequences or branching networks; state-of-the-art algorithms can generate solutions containing thousands of actions.

For plans executed by the human brain, atomic actions are muscle activations. In very round numbers, we have about 103 muscles to activate (639, by some counts, but many of them have multiple subunits); we can modulate their activation perhaps 10 times per second; and we are alive and awake for about 109 seconds in all. Thus, a human life contains about 1013 actions, give or take one or two orders of

magnitude. Even if we restrict ourselves to planning over much shorter time horizons—for example, a two-week vacation in Hawaii—a detailed motor plan would contain around 1010 actions. This is a lot more than 1000.

To bridge this gap, AI systems will probably have to do what humans appear to do: plan at higher levels of abstraction. A reasonable plan for the Hawaii vacation might be "Go to San Francisco airport; take Hawaiian Airlines flight 11 to Honolulu; do vacation stuff for two weeks; take Hawaiian Airlines flight 12 back to San Francisco; go home." Given such a plan, the action "Go to San Francisco airport" can be viewed as a planning task in itself, with a solution such as "Drive to the long-term parking lot; park; take the shuttle to the terminal." Each of these actions, in turn, can be decomposed further, until we reach the level of actions that can be executed without deliberation to generate the required motor control sequence.

Planning and Acting in Nondeterministic Domains: While the basic concepts are the same as in Chapter 4, there are also significant differences. These arise because planners deal with factored representations rather than atomic representations. This affects the way we represent the agent's capability for action and observation and the way we represent belief states—the sets of possible physical states the agent might be in—for unobservable and partially observable environments. We can also take ad- vantage of many of the domain-independent methods given in Chapter 10 for calculating search heuristics.

Consider this problem: given a chair and a table, the goal is to have them match—have the same color. In the initial state we have two cans of paint, but the colors of the paint and the furniture are unknown. Only the table is initially in the agent's field of view:

Init(Object(Table) ∧ Object(Chair) ∧ Can(C1) ∧ Can(C2) ∧ InView (Table)) Goal (Color (Chair , c) ∧ Color (Table, c))

There are two actions: removing the lid from a paint can and painting an object using the paint from an open can. The action schemas are straightforward, with one exception: we now allow preconditions and effects to contain variables that are not part of the action's variable list. That is, Paint(x, can) does not mention the variable c, representing the color of the paint in the can. In the fully observable case, this is not allowed—we would have to name the action Paint(x, can, c). But in the partially observable case, we might or might not know what color is in the can. (The variable c is universally quantified, just like all the other variables in an action schema.)

Action(RemoveLid (can),

PRECOND:Can(can)

EFFECT:Open(can))

Action(Paint(x , can),
PRECOND:Object(x) ∧ Can(can) ∧ Color (can, c) ∧ Open(can)
EFFECT:Color (x, c))

To solve a partially observable problem, the agent will have to reason about the percepts it will obtain when it is executing the plan. The percept will be supplied by the agent's sensors when it is actually acting, but when it is planning it will need a model of its sensors. In Chapter 4, this model was given by a function, PERCEPT(s). For planning, we augment PDDL with a new type of schema, the percept schema:

Multi agent Planning:

we have assumed that only one agent is doing the sensing, planning, and acting. When there are multiple agents in the environment, each agent faces a multi agent planning problem in which it tries to achieve its own goals with the help or hindrance of others.

Between the purely single-agent and truly multi agent cases is a wide spectrum of problems that exhibit various degrees of decomposition of the monolithic agent. An agent with multiple effectors that can operate concurrently—for example, a human who can type and speak at the same time—needs to do multi effector planning to manage each effector while handling positive and negative interactions among the effectors. When the effectors are physically decoupled into detached units—as in a fleet of delivery robots in a factory— multi effector planning becomes multibody planning. A multibody problem is still a "standard" single-agent problem as long as the relevant sensor information collected by each body can be pooled—either centrally or within each body—to form a common estimate of the world state that then informs the execution of the overall plan; in this case, the multiple bodies act as a single body.

When a single entity is doing the planning, there is really only one goal, which all the bodies necessarily share. When the bodies are distinct agents that do their own planning, they may still share identical goals; for example, two human tennis players who form a doubles team share the goal of winning the match. Even with shared goals, however, the multibody and multi agent cases are quite different. In a multibody robotic doubles team, a single plan dictates which body will go where on the court and which body will hit the ball. In a multi- agent doubles team, on the other hand, each agent decides what to do; without some method for coordination, both agents may decide to cover the same part of the court and each may leave the ball for the other to hit.

Planning with multiple simultaneous actions

For the time being, we will treat the multi effector, multibody, and multi agent settings in the same way, labeling them generically as **multi actor** settings, using the generic term **actor** to cove cover effectors, bodies, and agents. The goal of this section is to work out how to define transition models, correct plans, and efficient planning algorithms for the multi actor setting.

A correct plan is one that, if executed by the actors, achieves the goal. (In the true multi agent setting, of course, the agents may not agree to execute any particular plan, but at least they will know what plans would work if they did agree to execute them.) For simplicity, we assume perfect synchronization: each action takes the same amount of time and actions at each point in the joint plan are simultaneous.

$Actors(A, B)$
$Init(At(A, LeftBaseline) \wedge At(B, RightNet) \wedge$
$\quad Approaching(Ball, RightBaseline)) \wedge Partner(A, B) \wedge Partner(B, A)$
$Goal(Returned(Ball) \wedge (At(a, RightNet) \vee At(a, LeftNet))$
$Action(Hit(actor, Ball),$
$\quad$ PRECOND: $Approaching(Ball, loc) \wedge At(actor, loc)$
$\quad$ EFFECT: $Returned(Ball))$
$Action(Go(actor, to),$
$\quad$ PRECOND: $At(actor, loc) \wedge to \neq loc,$
$\quad$ EFFECT: $At(actor, to) \wedge \neg At(actor, loc))$

Figure 11.10 The doubles tennis problem. Two actors A and B are playing together and can be in one of four locations: $LeftBaseline$, $RightBaseline$, $LeftNet$, and $RightNet$. The ball can be returned only if a player is in the right place. Note that each action must include the actor as an argument.

Having put the actors together into a multi actor system with a huge branching factor, the principal focus of research on multi actor planning has been to decouple the actors to the extent possible, so that the complexity of the problem grows linearly with n rather than exponentially. If the actors have no interaction with one another—for example, n actors each playing a game of solitaire—then we can simply solve n separate problems. If the actors are loosely coupled, can we attain something close to this exponential improvement? This is, of course, a central question in many areas of AI.

The standard approach to loosely coupled problems is to pretend the problems are completely decoupled and then fix up the interactions. For the transition model, this means writing action schemas as if the actors acted independently. Let's see how this works for the doubles tennis problem. Let's

suppose that at one point in the game, the team has the goal of returning the ball that has been hit to them and ensuring that at least one of them is covering the net.

Planning with multiple agents Cooperation andcoordination:

Now let us consider the true multi agent setting in which each agent makes its own plan. To start with, let us assume that the goals and knowledge base are shared. One might think that this reduces to the multibody case—each agent simply computes the joint solution and executes its own part of that solution. Alas, the "the" in "the joint solution" is misleading. For our doubles team, more than one joint solution exists:

If both agents can agree on either plan 1 or plan 2, the goal will be achieved. But if A chooses plan 2 and B chooses plan 1, then nobody will return the ball. Conversely, if A chooses 1 and B chooses 2, then they will both try to hit the ball.

One option is to adopt a convention before engaging in joint activity. A convention is any constraint on the selection of joint plans. For example, the convention "stick to your side of the court" would rule out plan 1, causing the doubles partners to select plan 2. Drivers on a road face the problem of not colliding with each other; this is (partially) solved by adopting the convention "stay on the right side of the road" in most countries; the alternative, "stay on the left side," works equally well as long as all agents in an environment agree. Similar considerations apply to the development of human language, where the important thing is not which language each individual should speak, but the fact that a community all speaks the same language. When conventions are widespread, they are called social laws.

Conventions can also arise through evolutionary processes. For example, seed-eating harvester ants are social creatures that evolved from the less social wasps. Colonies of ants execute very elaborate joint plans without any centralized control—the queen's job is to re- produce, not to do centralized planning—and with very limited computation,

Communication, and memory capabilities in each ant (Gordon, 2000, 2007). The colony has many roles, including interior workers, patrollers, and foragers. Each ant chooses to perform a role ac- cording to the local conditions it observes. One final example of cooperative multi agent behavior appears in the flocking behavior of birds.

We can obtain a reasonable simulation of a flock if each bird agent (sometimes called a boid) observes the positions of its nearest neighbors and then chooses the heading and acceleration that maximizes the weighted sum of these three components.

(a) (b) (c)

Figure 11.11 (a) A simulated flock of birds, using Reynold's boids model. Image courtesy Giuseppe Randazzo, novastructura.net. (b) An actual flock of starlings. Image by Eduardo (pastaboy sleeps on flickr). (c) Two competitive teams of agents attempting to capture the towers in the NERO game. Image courtesy Risto Miikkulainen.

1. Cohesion: a positive score for getting closer to the average position of the neighbors
2. Separation: a negative score for getting too close to any one neighbor
3. Alignment: a positive score for getting closer to the average heading of the neighbors

If all the boids execute this policy, the flock exhibits the emergent behavior of flying as a pseudo rigid body with roughly constant density that does not disperse over time, and that occasionally makes sudden swooping motions. You can see a still images in Figure 11.11(a) and compare it to an actual flock in (b). As with ants, there is no need for each agent to possess a joint plan that models the actions of other agents. The most difficult multi agent problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control.