UNIT I

PROBLEM SOLVING

Introduction to AI - AI Applications - Problem solving agents – search algorithms – uninformed search strategies – Heuristic search strategies – Local search and optimization problems – adversarial search constraint satisfaction problems (CSP)

1. INTRODUCTION

INTELLIGENCE	ARTIFICIAL INTELLIGENCE
It is a natural process.	It is programmed by humans.
It is actually hereditary.	It is not hereditary.
Knowledge is required for intelligence.	KB and electricity are required to generate output.
No human is an expert. We may get better solutions from other humans.	Expert systems are made which aggregate many person's experience and ideas.

1.1 DEFINITION

The study of how to make computers do things at which at the moment, people are better.
"Artificial Intelligence is the ability of a computer to act like a human being".

- Systems that think like humans
- Systems that act like humans
- Systems that think rationally. Systems that act rationally.

"The exciting new effort to make computers think ... *machines with minds*, in the full and literal sense" (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning ..." (Bellman, 1978)	"The study of mental faculties through the use of computational models" (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act" (Winston, 1992)
"The art of creating machines that perform functions that require intelligence when performed by people" (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better" (Rich and Knight, 1991)	"A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes" (Schalkoff, 1990) "The branch of computer science that is concerned with the automation of intelligent behavior" (Luger and Stubblefield, 1993)

Figure 1.1 Some definitions of artificial intelligence, organized into four categories

(a) **Intelligence -** Ability to apply knowledge in order to perform better in an environment.

(b) **Artificial Intelligence -** Study and construction of agent programs that perform well in a given environment, for a given agent architecture**.**

(c) **Agent -** An entity that takes action in response to precepts from an environment**.**

(d) **Rationality** - property of a system which does the "right thing" given what it knows.

(e) **Logical Reasoning -** A process of deriving new sentences from old, such that the new sentences are necessarily true if the old ones are true.

Four Approaches of Artificial Intelligence:

- ➢ Acting humanly: The Turing test approach.
- ➢ Thinking humanly: The cognitive modelling approach.
- ➢ Thinking rationally: The laws of thought approach.
- ➢ Acting rationally: The rational agent approach.

1.2 ACTING HUMANLY: THE TURING TEST APPROACH

The Turing Test, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer.

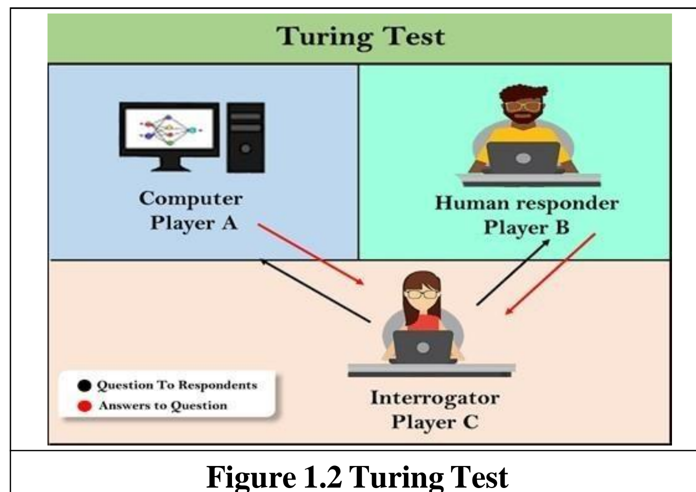


Figure 1.2 Turing Test

- **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.

Total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need

- **computer vision** to perceive objects, and **robotics** to manipulate objects and move about**.**

Thinking humanly: The cognitive modelling approach

Analyse how a given program thinks like a human, we must have some way of determining how humans think. The interdisciplinary field of **cognitive science** brings together computer models from AI and experimental techniques from psychology to try to construct precise and testable theories of the workings of the human mind.

Although cognitive science is a fascinating field in itself, we are not going to be discussing it all that much in this book. We will occasionally comment on similarities or differences between AI techniques and human cognition. Real cognitive science, however, is necessarily based on experimental investigation of actual humans or animals, and we assume that the reader only has access to a computer for experimentation. We will simply note that AI and cognitive science continue to fertilize each other, especially in the areas of vision, natural language, and learning.

Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify ``right thinking," that is, irrefutable reasoning processes. His famous syllogisms provided patterns for argument structures that always gave correct conclusions 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, and initiated the field of ***logic.***

Acting rationally: The rational agent approach

Acting rationally means acting so as to achieve one's goals, given one's beliefs. An agent is just something that perceives and acts.

The right thing: that which is expected to maximize goal achievement, given the available information

Does not necessary involve thinking.

For Example - blinking reflex- but should be in the service of rational action.

1.3 FUTURE OF ARTIFICIAL INTELLIGENCE

- **Transportation:** Although it could take a decade or more to perfect them, autonomous cars will one day ferry us from place to place.
- **Manufacturing:** AI powered robots work alongside humans to perform a limited range of tasks like assembly and stacking, and predictive analysis sensors keep equipment running smoothly.
- **Healthcare:** In the comparatively AI-nascent field of healthcare, diseases are more quickly and accurately diagnosed, drug discovery is sped up and streamlined, virtual nursing assistants monitor patients and big data analysis helps to create a more personalized patient experience.
- **Education:** Textbooks are digitized with the help of AI, early-stage virtual tutors assist human instructors and facial analysis gauges the emotions of students to help determine who's struggling or bored and better tailor the experience to their individualneeds.
- **Media:** Journalism is harnessing AI, too, and will continue to benefit from it. Bloomberg uses Cyborg technology to help make quick sense of complex financial reports. The Associated Press employs the natural language abilities of Automated Insights to produce 3,700 earning reports stories per year — nearly four times more than in the recent past
- **Customer Service:** Last but hardly least, Google is working on an AI assistant that can place human-like calls to make appointments at, say, your neighborhood hair salon. In addition to words, the system understands context and nuance.

1.4 CHARACTERISTICS OF INTELLIGENT AGENTS

Situatedness

The agent receives some form of sensory input from its environment, and it performs some action that changes its environment in some way.

Examples of environments: the physical world and the Internet.

- Autonomy

The agent can act without direct intervention by humans or other agents and that it has control over its own actions and internal state.

- Adaptivity

The agent is capable of

(1) reacting flexibly to changes in its environment;

(2) taking goal-directed initiative (i.e., is pro-active), when appropriate; and

(3) Learning from its own experience, its environment, and interactions with others.

- Sociability

  The agent is capable of interacting in a peer-to-peer manner with other agents or humans

1.5 AGENTS AND ITS TYPES

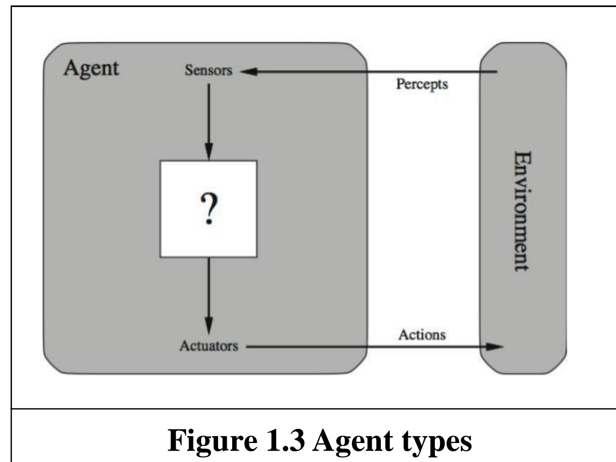


Figure 1.3 Agent types

An agent is anything that can be viewed as perceiving its environment throughsensors and acting upon that environment through actuators.

- Human Sensors:
- Eyes, ears, and other organs for sensors.
- Human Actuators:
- Hands, legs, mouth, and other body parts.
- Robotic Sensors:
- Mic, cameras and infrared range finders for sensors
- Robotic Actuators:
- Motors, Display, speakers etc An agent can be:

Human-Agent: A human agent has eyes, ears, and other organs which work for sensors and hand, legs, vocal tract work for actuators.

Robotic Agent: A robotic agent can have cameras, infrared range finder, NLP for sensors and various motors for actuators.

Software Agent: Software agent can have keystrokes, file contents as sensory input and act on those inputs and display output on the screen.

Hence the world around us is full of agents such as thermostat, cell phone, camera, and even we are also agents. Before moving forward, we should first know about sensors, effectors, and actuators.

Sensor: Sensor is a device which detects the change in the environment and sends the information to other electronic devices. An agent observes its environment through sensors.

Actuators: Actuators are the component of machines that converts energy into motion. The actuators are only responsible for moving and controlling a system. An actuator can be an electric motor, gears, rails, etc.

Effectors: Effectors are the devices which affect the environment. Effectors can be legs, wheels, arms, fingers, wings, fins, and display screen.

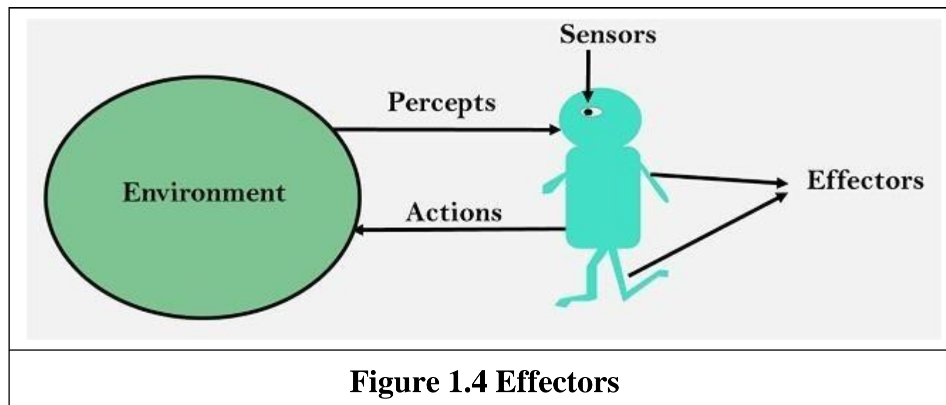


Figure 1.4 Effectors

1.6 PROPERTIES OF ENVIRONMENT

An **environment** is everything in the world which surrounds the agent, but it is not a part of an agent itself. An environment can be described as a situation in which an agent is present.

The environment is where agent lives, operate and provide the agent with something to sense and act upon it.

Fully observable vs Partially Observable:

If an agent sensor can sense or access the complete state of an environment at each point of time then it is **a fully observable** environment, else it is **partially observable**.

A fully observable environment is easy as there is no need to maintain the internal state to keep track history of the world.

An agent with no sensors in all environments then such an environment is called as unobservable.

Example: chess – the board is fully observable, as are opponent's moves. Driving – what is around the next bend is not observable and hence partially observable.

1. Deterministic vs Stochastic

- If an agent's current state and selected action can completely determine the next state of the environment, then such environment is called a deterministic environment.

- A stochastic environment is random in nature and cannot be determined completely by an agent.
- In a deterministic, fully observable environment, agent does not need to worry about uncertainty.

2. **Episodic vs Sequential**

- In an episodic environment, there is a series of one-shot actions, and only the current percept is required for the action.
- However, in Sequential environment, an agent requires memory of past actions to determine the next best actions.

3. **Single-agent vs Multi-agent**

- If only one agent is involved in an environment, and operating by itself then such an environment is called single agent environment.
- However, if multiple agents are operating in an environment, then such an environment is called a multi-agent environment.
- The agent design problems in the multi-agent environment are different from single agent environment.

4. **Static vs Dynamic**

- If the environment can change itself while an agent is deliberating then such environment is called a dynamic environment else it is called a static environment.
- Static environments are easy to deal because an agent does not need to continue looking at the world while deciding for an action.
- However for dynamic environment, agents need to keep looking at the world at each action.
- Taxi driving is an example of a dynamic environment whereas Crossword puzzles are an example of a static environment.

5. **Discrete vs Continuous**

- If in an environment there are a finite number of precepts and actions that can be performed within it, then such an environment is called a discrete environment else it is called continuous environment.
- A chess game comes under discrete environment as there is a finite number of moves that can be performed.
- A self-driving car is an example of a continuous environment.

6. Known vs Unknown

- Known and unknown are not actually a feature of an environment, but it is an agent's state of knowledge to perform an action.
- In a known environment, the results for all actions are known to the agent. While in unknown environment, agent needs to learn how it works in order to perform an action.
- It is quite possible that a known environment to be partially observable and anUnknown environment to be fully observable.

7. Accessible vs. Inaccessible

- If an agent can obtain complete and accurate information about the state's environment, then such an environment is called an Accessible environment else it is called inaccessible.
- An empty room whose state can be defined by its temperature is an example of an accessible environment.
- Information about an event on earth is an example of Inaccessible environment.

Task environments, which are essentially the "problems" to which rational agents are the "solutions."

PEAS: Performance Measure, Environment, Actuators, Sensors

Performance

The output which we get from the agent. All the necessary results that an agent gives after processing comes under its performance.

Environment

All the surrounding things and conditions of an agent fall in this section. It basically consists of all the things under which the agents work.

Actuators

The devices, hardware or software through which the agent performs any actions or processes any information to produce a result are the actuators of the agent.

Sensors

The devices through which the agent observes and perceives its environment are the sensors of the agent.

Agent Type	Performance Measure	Environment	Actuators	Sensors
Medical diagnosis system	Healthy patient, reduced costs	Patient, hospital, staff	Display of 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 of scene categorization	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	Purity, yield, safety	Refinery, operators	Valves, pumps, heaters, displays	Temperature, pressure, chemical sensors
Interactive English tutor	Student's score on test	Set of students, testing agency	Display of exercises, suggestions, corrections	Keyboard entry

Figure 1.5 Examples of agent types and their PEAS descriptions

Rational Agent - A system is rational if it does the "right thing". Given what it knows.

Characteristic of Rational Agent

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

For every possible percept sequence, a rational agent should select an action that is expected to maximize its performance measure, given the evidence provided by the percept sequence and whatever built-in knowledge the agent has.

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.

- **Ideal Rational Agent** precepts and does things. It has a greater performance measure.

  Eg. Crossing road. Here first perception occurs on both sides and then only action. No perception occurs in **Degenerate Agent**.

  Eg. Clock. It does not view the surroundings. No matter what happens outside. The clock works based on inbuilt program.

- **Ideal Agent** describes by ideal mappings. "Specifying which action an agent ought to take in response to any given percept sequence provides a design for ideal agent".

- **Eg.** SQRT function calculation in calculator.
- Doing actions in order to modify future precepts-sometimes called **information gathering**- is an important part of rationality.
- A rational agent should be **autonomous**-it should learn from its own prior knowledge (experience).

The Structure of Intelligent Agents

Agent = Architecture + Agent Program

Architecture = the machinery that an agent executes on. (Hardware)

Agent Program = an implementation of an agent function. (Algorithm, Logic – Software)

1.7 PROBLEM SOLVING APPROACH TO TYPICAL AI PROBLEMS

Problem-solving agents

In Artificial Intelligence, Search techniques are universal problem-solving methods. **Rational agents** or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem- solving agents are the goal-based agents and use atomic representation. In this topic, wewill learn various problem-solving search algorithms.

- Some of the most popularly used problem solving with the help of artificial intelligence are:

1. Chess.
2. Travelling Salesman Problem.
3. Tower of Hanoi Problem.
4. Water-Jug Problem.
5. N-Queen Problem.

Problem Searching

- In general, searching refers to as finding information one needs.
- Searching is the most commonly used technique of problem solving in artificial intelligence.
- The searching algorithm helps us to search for solution of particular problem.

Problem: Problems are the issues which comes across any system. A solution is needed to solve that particular problem.

Steps : Solve Problem Using Artificial Intelligence

- The process of solving a problem consists of five steps. These are:

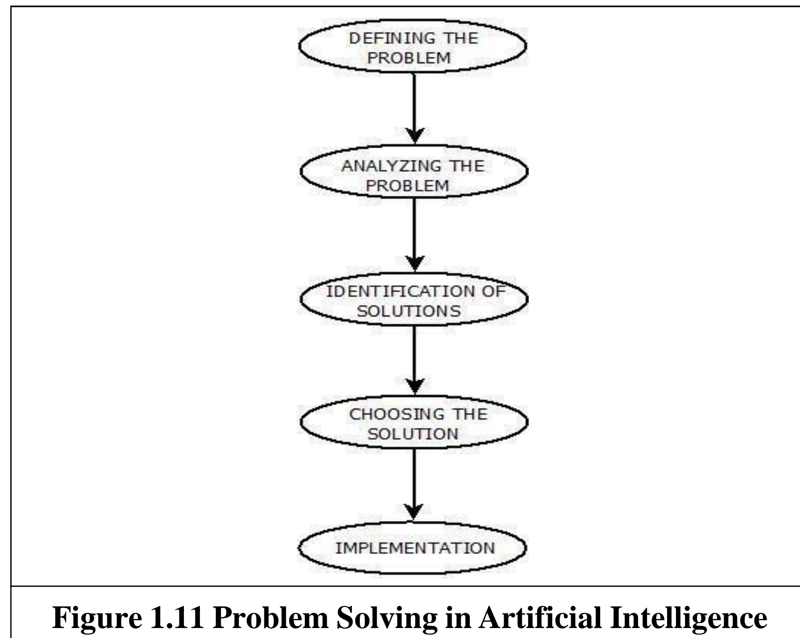


Figure 1.11 Problem Solving in Artificial Intelligence

Defining The Problem: The definition of the problem must be included precisely. It should contain the possible initial as well as final situations which should result in acceptable solution.

1. **Analyzing The Problem**: Analyzing the problem and its requirement must be done as few features can have immense impact on the resulting solution.
2. **Identification Of Solutions**: This phase generates reasonable amount of solutions to the given problem in a particular range.
3. **Choosing a Solution**: From all the identified solutions, the best solution is chosen basis on the results produced by respective solutions.
4. **Implementation**: After choosing the best solution, its implementation is done.

Measuring problem-solving performance

We can evaluate an algorithm's performance 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?

Search Algorithm Terminologies

- Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:
  1. Search Space: Search space represents a set of possible solutions, which a system may have.
  2. Start State: It is a state from where agent begins the search.
  3. Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.
- Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.
- Actions: It gives the description of all the available actions to the agent.
- Transition model: A description of what each action do, can be represented as a transition model.
- Path Cost: It is a function which assigns a numeric cost to each path.
- Solution: It is an action sequence which leads from the start node to the goal node. Optimal Solution: If a solution has the lowest cost among all solutions.

Example Problems

A **Toy Problem** is intended to illustrate or exercise various problem-solving methods. A**real- world problem** is one whose solutions people actually care about.

Toy Problems

Vacuum World

States: The state is determined by both the agent location and the dirt locations. The agent is in one of the 2 locations, each of which might or might not contain dirt. Thus there are 2*2^2=8 possible world states.

Initial state: Any state can be designated as the initial state.

Actions: In this simple environment, each state has just three actions: *Left*, *Right*, and *Suck*. Larger environments might also include *Up* and *Down*.

Transition model: The actions have their expected effects, except that moving *Left* in the leftmost squ are, moving *Right* in the rightmost square, and *Suck*ing in a clean square have no effect. The complete state space is shown in Figure.

Goal test: This checks whether all the squares are clean.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

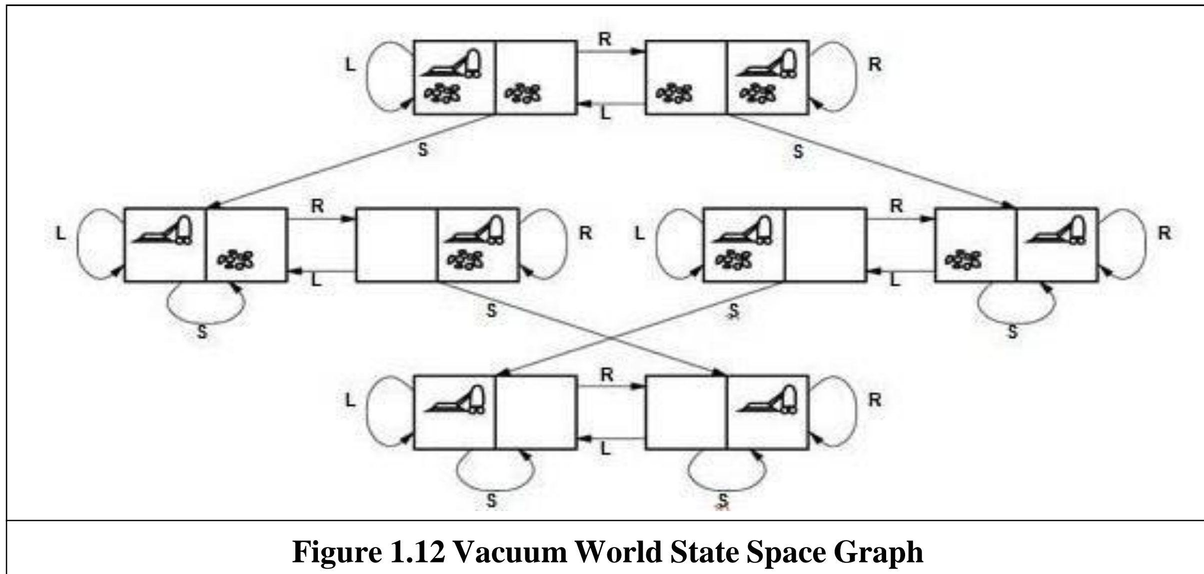


Figure 1.12 Vacuum World State Space Graph

1) 8- Puzzle Problem

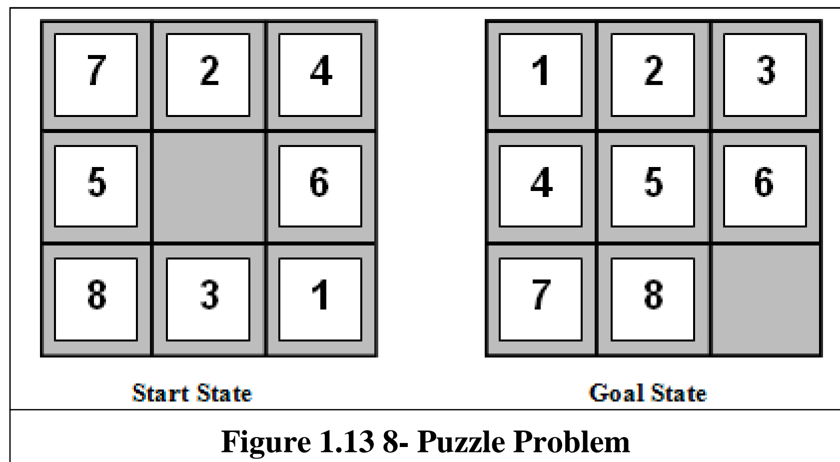


Figure 1.13 8- Puzzle Problem

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. Note that any given goal can be reached from exactly half of the possible initial states.

The simplest formulation defines the actions as movements of the blank space *Left*, *Right*, *Up*, or *Down*. Different subsets of these are possible depending on where the blank is.

Transition model: Given a state and action, this returns the resulting state; forexample, if we apply *Left* to the start state in Figure 3.4, the resulting state has the 5 and the blank switched.

Goal test: This checks whether the state matches the goal configuration shown in Figure. **Path cost:** Each step costs 1, so the path cost is the number of steps in the path.

Queens Problem

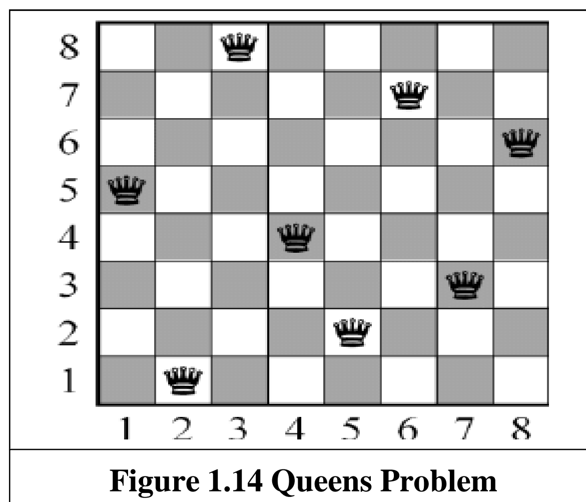


Figure 1.14 Queens Problem

- **States**: Any arrangement of 0 to 8 queens on the board is a state.
- **Initial state**: No queens on the board.
- **Actions**: Add a queen to any empty square.
- **Transition model**: Returns the board with a queen added to the specified square.
- **Goal test**: 8 queens are on the board, none attacked.

Consider the given problem. Describe the operator involved in it. Consider the water jug problem: You are given two jugs, a 4-gallon one and 3-gallon one. Neither has any measuring marker on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallon of water from the 4-gallon jug ?

Explicit Assumptions: A jug can be filled from the pump, water can be poured out of a jug on to the ground, water can be poured from one jug to another and that there are no other measuring devices available.

Here the initial state is (0, 0). The goal state is (2, n) for any value of n.

State Space Representation: we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note that $0 \leq x \leq 4$, and $0 \leq y \leq 3$.

To solve this we have to make some assumptions not mentioned in the problem. They are:

- We can fill a jug from the pump.
- We can pour water out of a jug to the ground.
- We can pour water from one jug to another.
- There is no measuring device available.

Operators - we must define a set of operators that will take us from one state to another.

Table 1.1

Sr.	Current State	Next State	Descriptions
1	(x,y) if x < 4	(4,y)	Fill the 4 gallon jug
2	(x,y) if x < 3	(x,3)	Fill the 3 gallon jug
3	(x,y) if x > 0	(x – d, y)	Pour some water out of the 4 gallon jug
4	(x,y) if y > 0	(x, y – d)	Pour some water out of the 3 gallon jug
5	(x,y) if y > 0	(0, y)	Empty the 4 gallon jug
6	(x,y) if y > 0	(x 0)	Empty the 3 gallon jug on the ground
7	(x,y) if x + y > = 4 and y > 0	(4, y – (4 – x))	Pour water from the 3 gallon jug into the 4 gallon jug until the 4 gallon jug is full
8	(x,y) if x + y > = 3 and x > 0	(x – (3 – x), 3)	Pour water from the 4 gallon jug into the 3 gallon jug until the 3 gallon jug is full
9	(x,y) if x + y < = 4 and y > 0	(x + y, 0)	Pour all the water from the 3 gallon jug into the 4 gallon jug
10	(x,y) if x + y < = 3 and x > 0	(0, x + y)	Pour all the water from the 4 gallon jug into the 3 gallon jug
11	(0, 2)	(2, 0)	Pour the 2 gallons from 3 gallon jug into the 4 gallon jug
12	(2, y)	(0, y)	Empty the 2 gallons in the 4 gallon jug on the ground

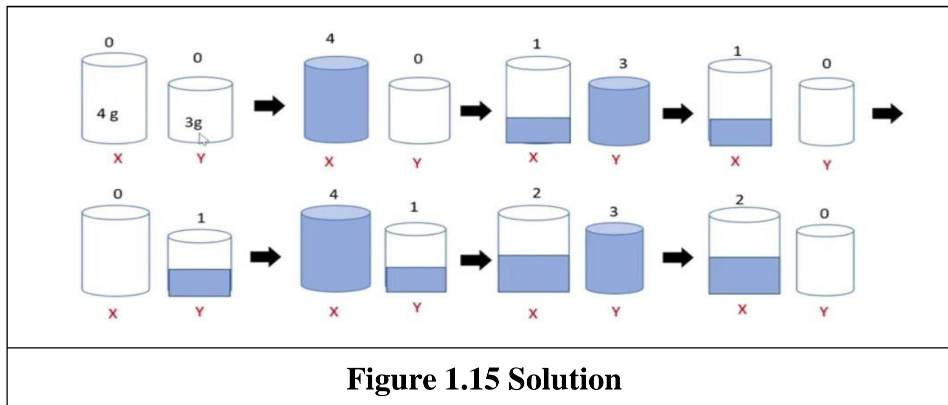


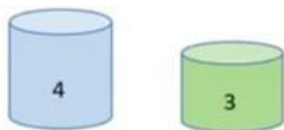
Figure 1.15 Solution

Table 1.2

Solution

S.No.	Gallons in 4-gel jug(x)	Gallons in 3-gel jug (y)	Rule Applied
1.	0	0	Initial state
2..	4	0	1. Fill 4
3	1	3	6. Poor 4 into 3 to fill
4.	1	0	4. Empty 3
5.	0	1	8. Poor all of 4 into 3
6.	4	1	1. Fill 4
7.	2	3	6. Poor 4 into 3

- 4-gallon one and a 3-gallon Jug



- No measuring mark on the jug.
- There is a pump to fill the jugs with water.
- How can you get exactly 2 gallon of water into the 4-gallon jug?

2.1 PROBLEM SOLVING BY SEARCH

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 sequenceof 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

2.2 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.

Different Search Algorithm

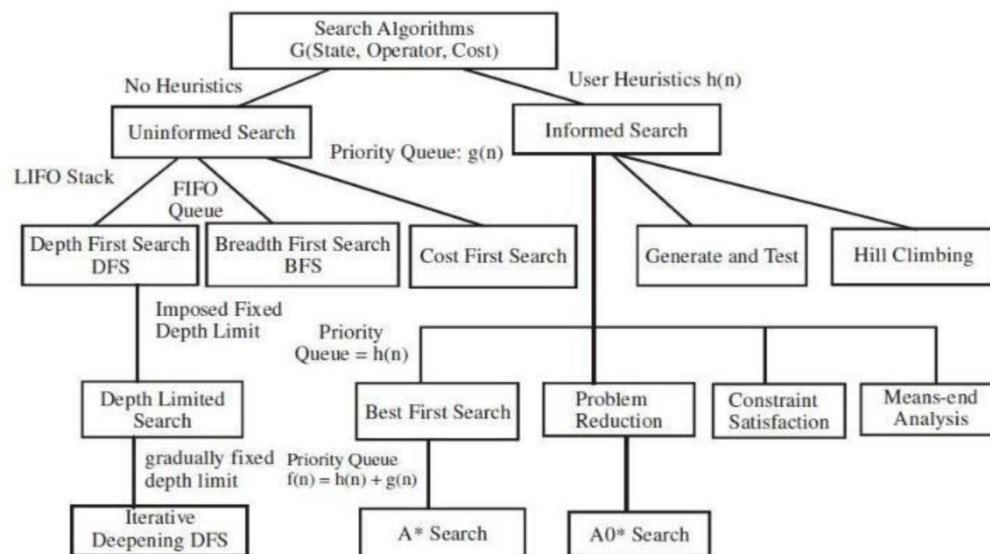


Figure 2.4 Different Search Algorithms

2.3 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.

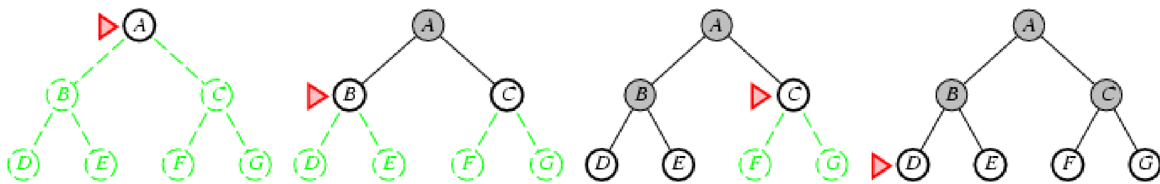


Figure 2.5 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Properties of breadth-first-search

Complete?? Yes (if b is finite)

Time?? $1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$, i.e., exp. in d

Space?? $O(b^{d+1})$ (keeps every node in memory)

Optimal?? No, unless step costs are constant

Space is the big problem; can easily generate nodes at 100MB/sec
so 24hrs = 8640GB.

Figure 2.6 Breadth-first-search properties

Time complexity for BFS

Assume every state has b successors. The root of the search tree generates b nodes at the first level, each of which generates b more nodes, for a total of b^2 at the second level. Each of these generates b more nodes, yielding b^3 nodes at the third level, and so on. Now suppose, that the solution is at depth d. In the worst case, we would expand all but the last node at level d, generating b^{d+1} - b nodes at level d+1.

Then the total number of nodes generated is $b + b^2 + b^3 + \ldots + b^d + (b^{d+1} + b) = O(b^{d+1})$.

Every node that is generated must remain in memory, because it is either part of the fringe or is an ancestor of a fringe node. The space compleity is, therefore, the same as the time complexity

2.4 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.

Expand least-cost unexpanded node

Implementation:
$fringe$ = queue ordered by path cost, lowest first

Equivalent to breadth-first if step costs all equal

Complete?? Yes, if step cost $\geq \epsilon$

Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

2.5 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 1.31. 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.

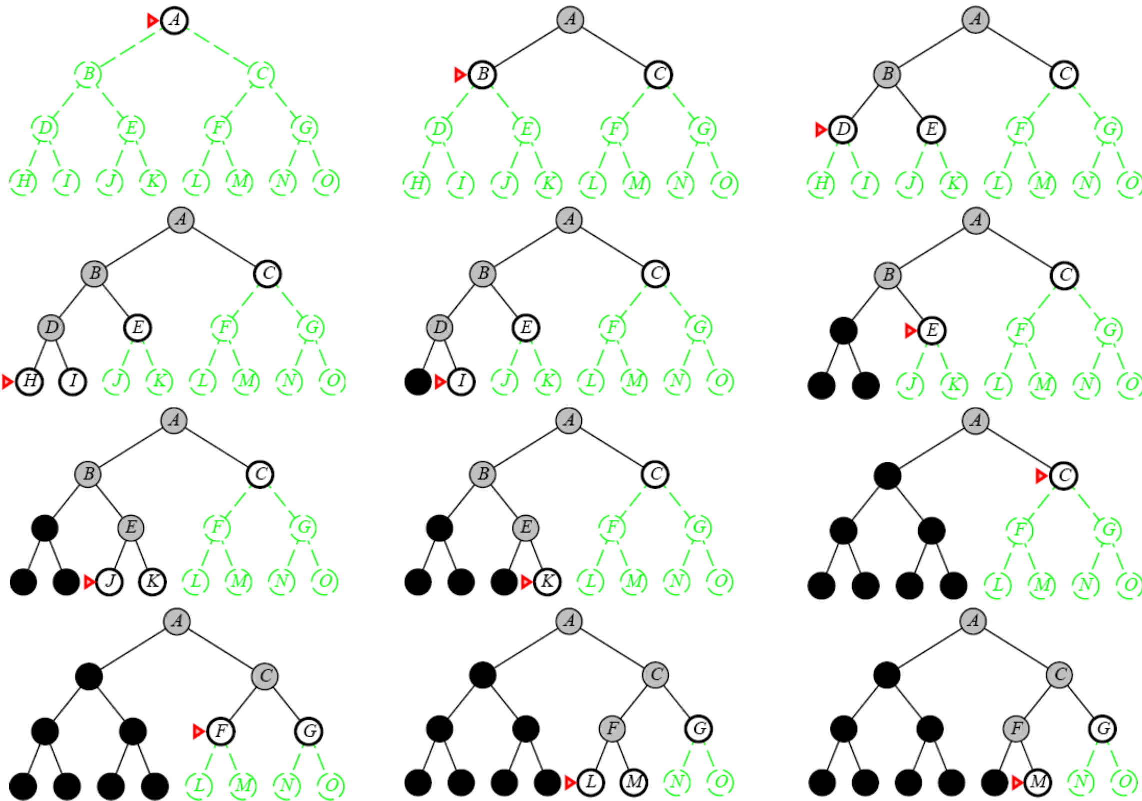


Figure 2.7 Depth-first-search on a binary tree. Nodes that have been expanded and have node scendants in the fringe can be removed from the memory; these are shown in black. Nodes at depth 3 are assumed to have no successors and M is the only goal node.

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

Depth-first-search has very modest memory requirements. 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 (Refer Figure 2.7).

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

Using the same assumptions as Figure, and assuming that nodes at the same depth as the goal node have no successors, we find the depth-first-search would require 118 kilobytes instead of 10 petabytes, a factor of 10 billion times less space.

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.

2.12 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

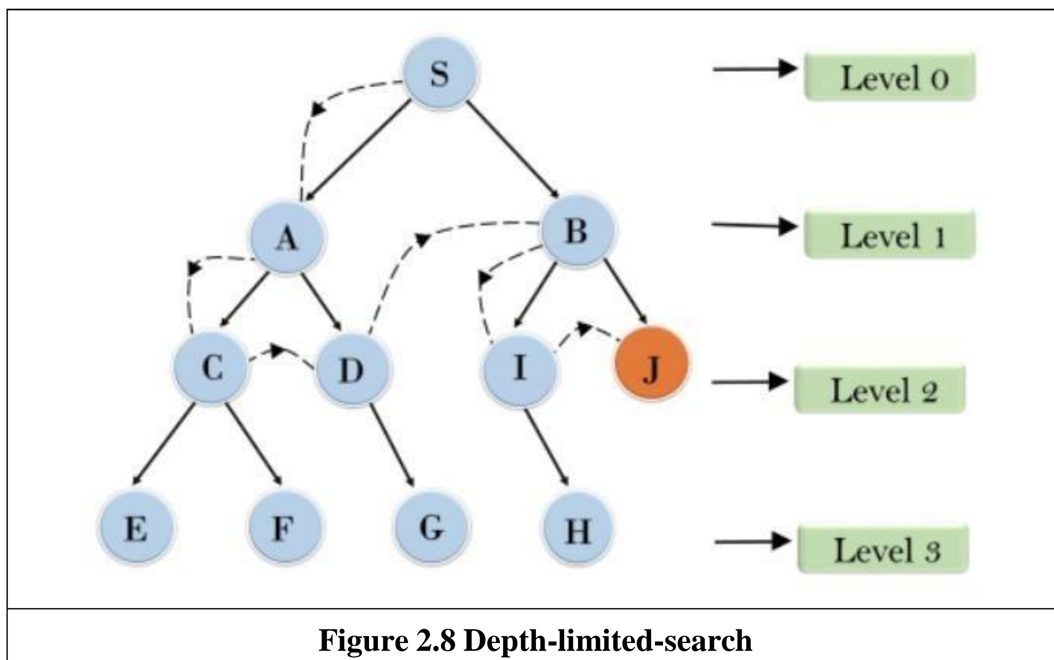


Figure 2.8 Depth-limited-search

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 complete is O(bl). Depth-first-search can be viewed as a special case of depth-limited search with l = oo Sometimes, depth limits can be based on knowledge of the problem. For, example, on the map of Romania there are 20 cities. Therefore, we know that ifthere is a solution, it must be of length 19 at the longest, So l = 10 is a possible choice. However, it can be shown that any city can be reached from any other city in at most 9 steps. This number known as the **diameter** of the state space, gives us a better depth limit.

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.

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 2.9 Recursive implementation of Depth-limited-search

2.13 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.

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 andoptimal 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 2.10 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 returns *failure*, meaning that no solution exists.

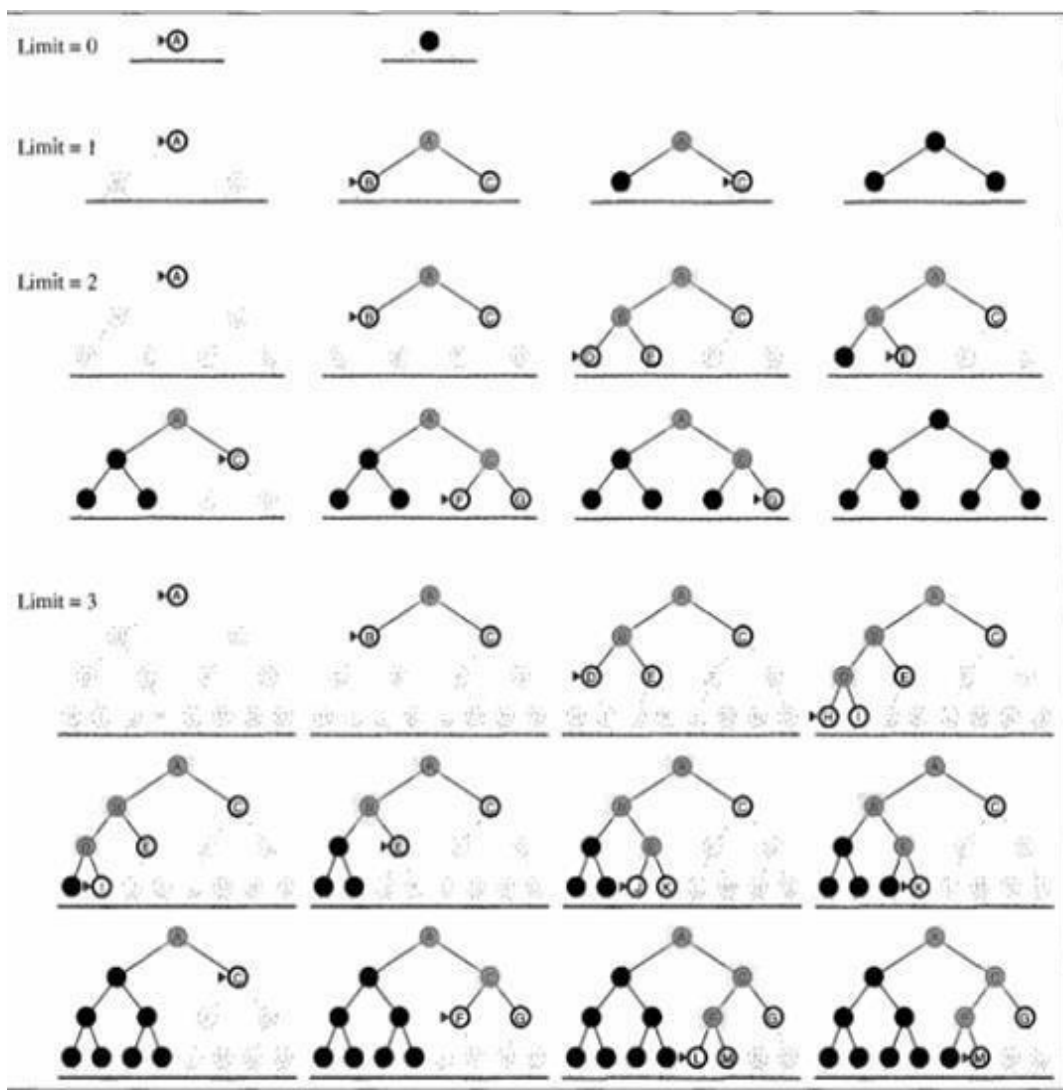


Figure 2.11 Four iterations of iterative deepening search on a binary tree

Iterative search is not as wasteful as it might seem

Iterative deepening

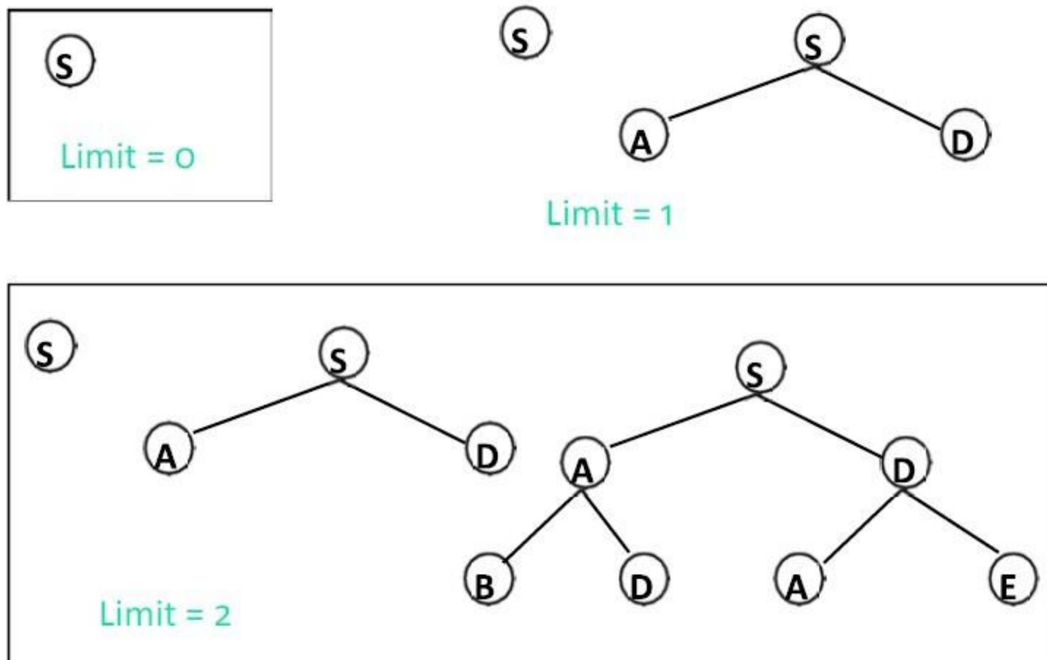


Figure 2.12 Iterative search is not as wasteful as it might seem

Properties of iterative deepening search

Complete?? Yes

Time?? $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$

Space?? $O(bd)$

Optimal?? No, unless step costs are constant

Can be modified to explore uniform-cost tree

Numerical comparison for $b = 10$ and $d = 5$, solution at far right leaf:

$$N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$$
$$N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$$

IDS does better because other nodes at depth d are not expanded

BFS can be modified to apply goal test when a node is generated

Figure 2.13 Properties of iterative deepening search

Bidirectional Search

The idea behind bidirectional search is to run two simultaneous searches – oneforward 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.

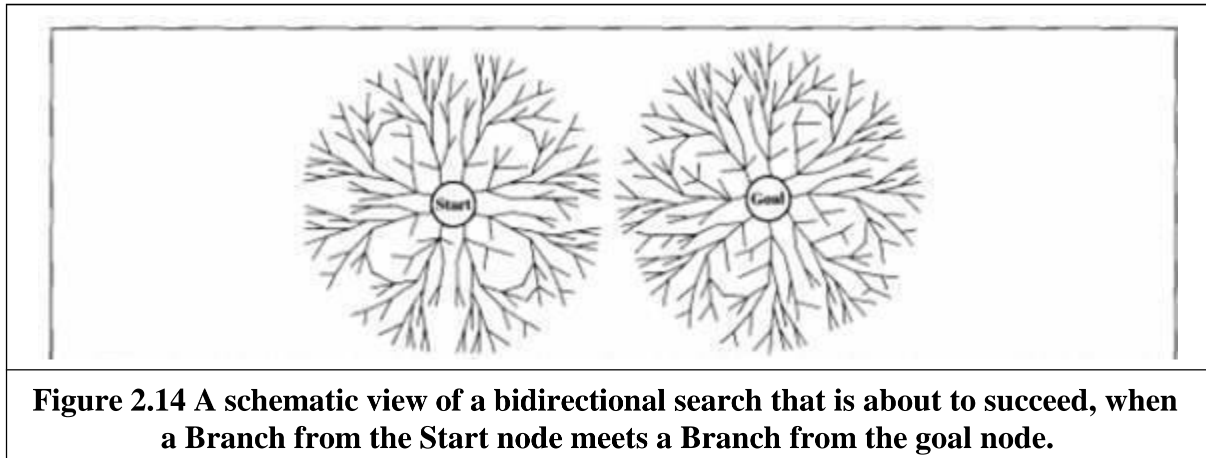


Figure 2.14 A schematic view of a bidirectional search that is about to succeed, when a Branch from the Start node meets a Branch from the goal node.

- Before moving into bidirectional search let’s first understand a few terms.
- Forward Search: Looking in-front of the end from start.
- Backward Search: Looking from end to the start back-wards.
- So Bidirectional Search as the name suggests is a combination of forwarding and backward search. Basically, if the average branching factor going out of node / fan- out, if fan-out is less, prefer forward search. Else if the average branching factor is going into a node/fan in is less (i.e. fan-out is more), prefer backward search.
- We must traverse the tree from the start node and the goal node and wherever they meet the path from the start node to the goal through the intersection is the optimal solution. The BS Algorithm is applicable when generating predecessors is easy in bothforward and backward directions and there exist only 1 or fewer goal states.

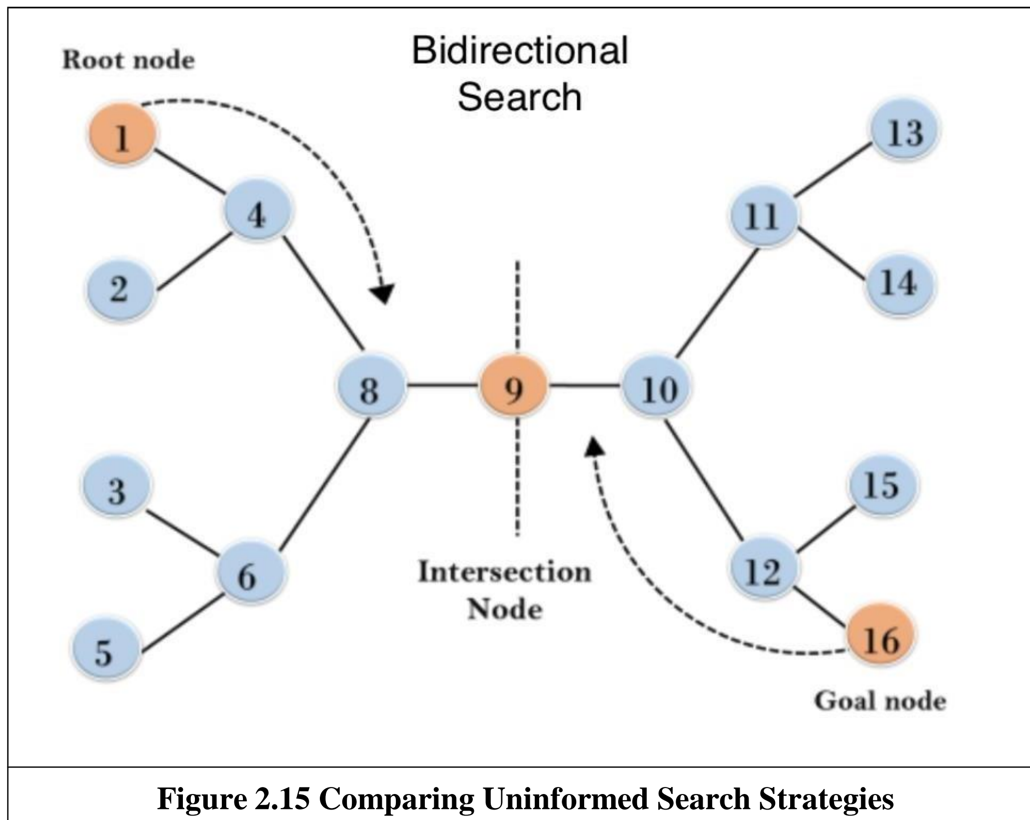


Figure 2.15 Comparing Uninformed Search Strategies

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 2.16 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. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs >= E for positive E; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

2.14 SEARCHING WITH PARTIAL INFORMATION

- Different types of incompleteness lead to three distinct problem types:
- **Sensorless problems** (conformant): If the agent has no sensors at all
- **Contingency problem**: if the environment if partially observable or if action are uncertain (adversarial)
- **Exploration problems**: When the states and actions of the environment are unknown.

- No sensor
- Initial State(1,2,3,4,5,6,7,8)
- After action [Right] the state (2,4,6,8)
- After action [Suck] the state (4, 8)
- After action [Left] the state (3,7)
- After action [Suck] the state (8)
- Answer : [Right, Suck, Left, Suck] coerce the world into state 7 without any sensor
- Belief State: Such state that agent belief to be there

Partial knowledge of states and actions:

- *sensorless or conformant problem*
  - Agent may have no idea where it is; solution (if any) is a sequence.
- *contingency problem*
  - Percepts provide *new* information about current state; solution is a tree or policy; often interleave search and execution.
  - If uncertainty is caused by actions of another agent: *adversarial problem*
- *exploration problem*
  - When states and actions of the environment are unknown.

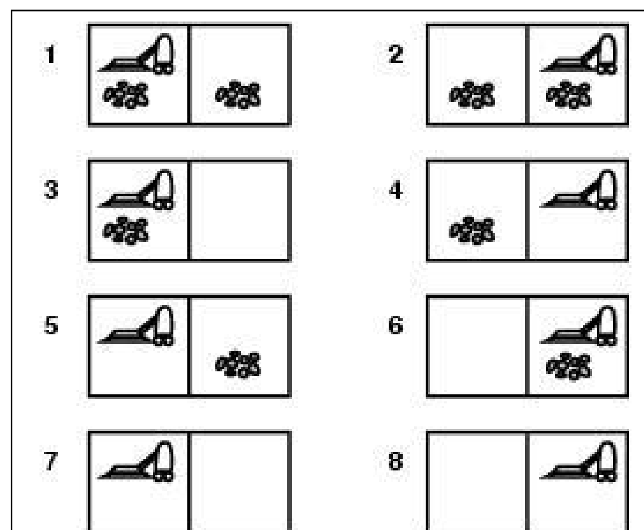


Figure 2.17 states and actions of the environment are unknown

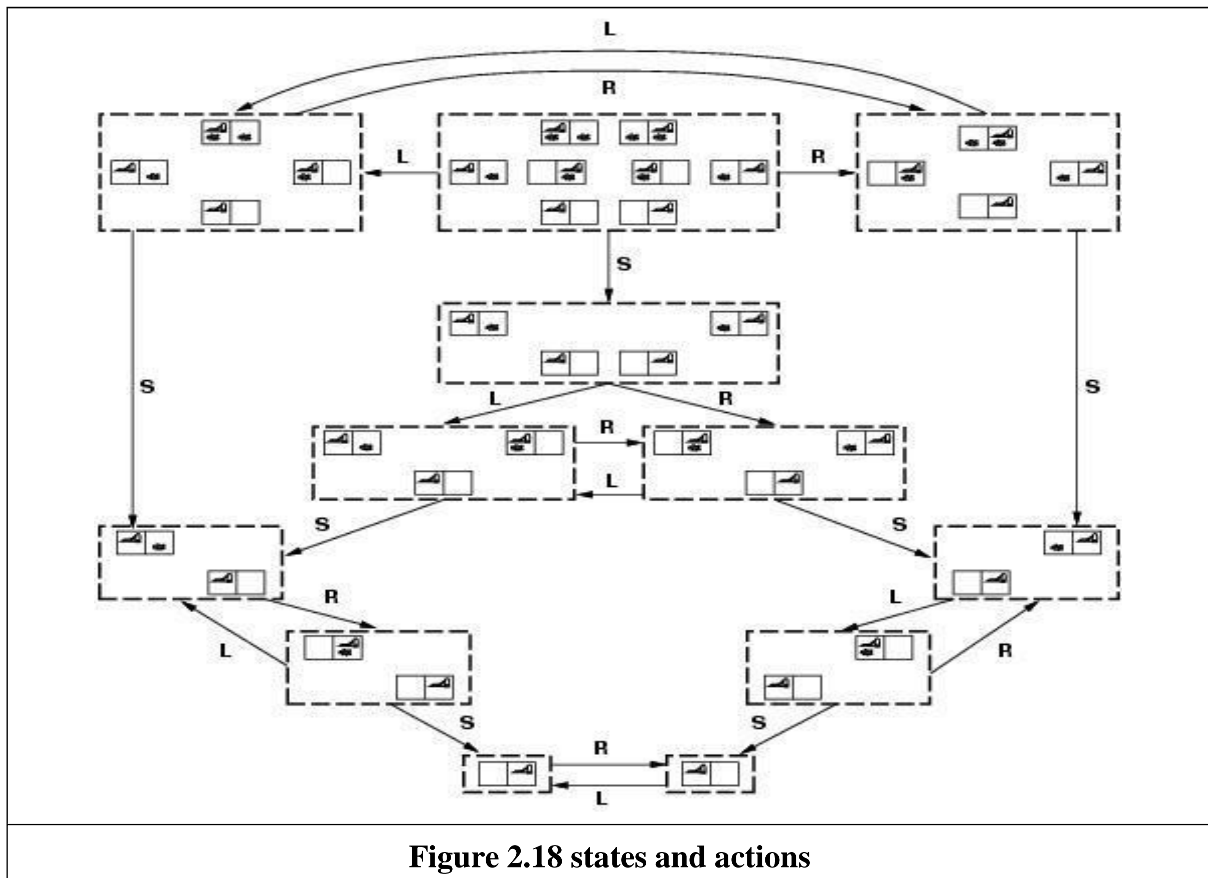


Figure 2.18 states and actions

Contingency, start in {1,3}.

Murphy's law, Suck *can* dirty a clean carpet. Local sensing: dirt, location only.

- Percept = [L,Dirty] ={1,3}
- [Suck] = {5,7}
- [Right] ={6,8}
- [Suck] in {6}={8} (Success)
- BUT [Suck] in {8} = failure Solution??
- Belief-state: no fixed action sequence guarantees solution

Relax requirement:

- [*Suck*, *Right*, if *[R,dirty]* then *Suck*]
- Select actions based on contingencies arising during execution.

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space.

In AI, where the graph is represented implicitly by the initial state and successor function, the complexity is expressed in terms of three quantities:

b, the **branching factor** or maximum number of successors of any node;

d, the **depth** of the **shallowest goal node**; and

m, the **maximum length** of any path in the state space.

Search-cost - typically depends upon the time complexity but can also include the term for memory usage.

Total–cost – It combines the search-cost and the path cost of the solution found.

2.15 INFORMED SEARCH AND EXPLORATION

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.

2.16 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) = estimated 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 2.19).

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. For example, the initial state is In(Arad),and the straight line distance heuristic hSLD (In(Arad)) is found to be 366.

Using the **straight-line distance** heuristic **hSLD,** the goal state can be reached faster.

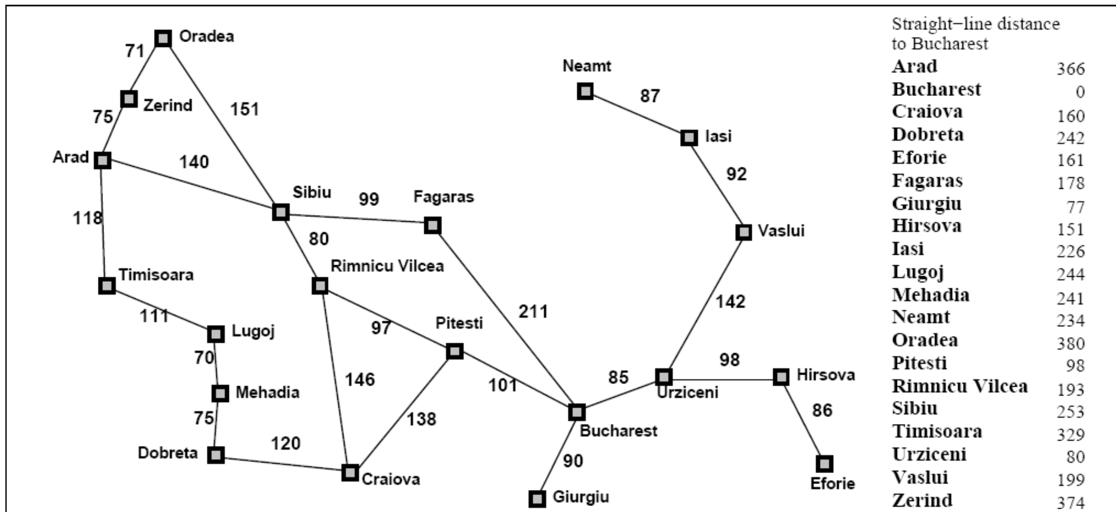


Figure 2.19 Values of h_{SLD} - straight line distances to Bucharest

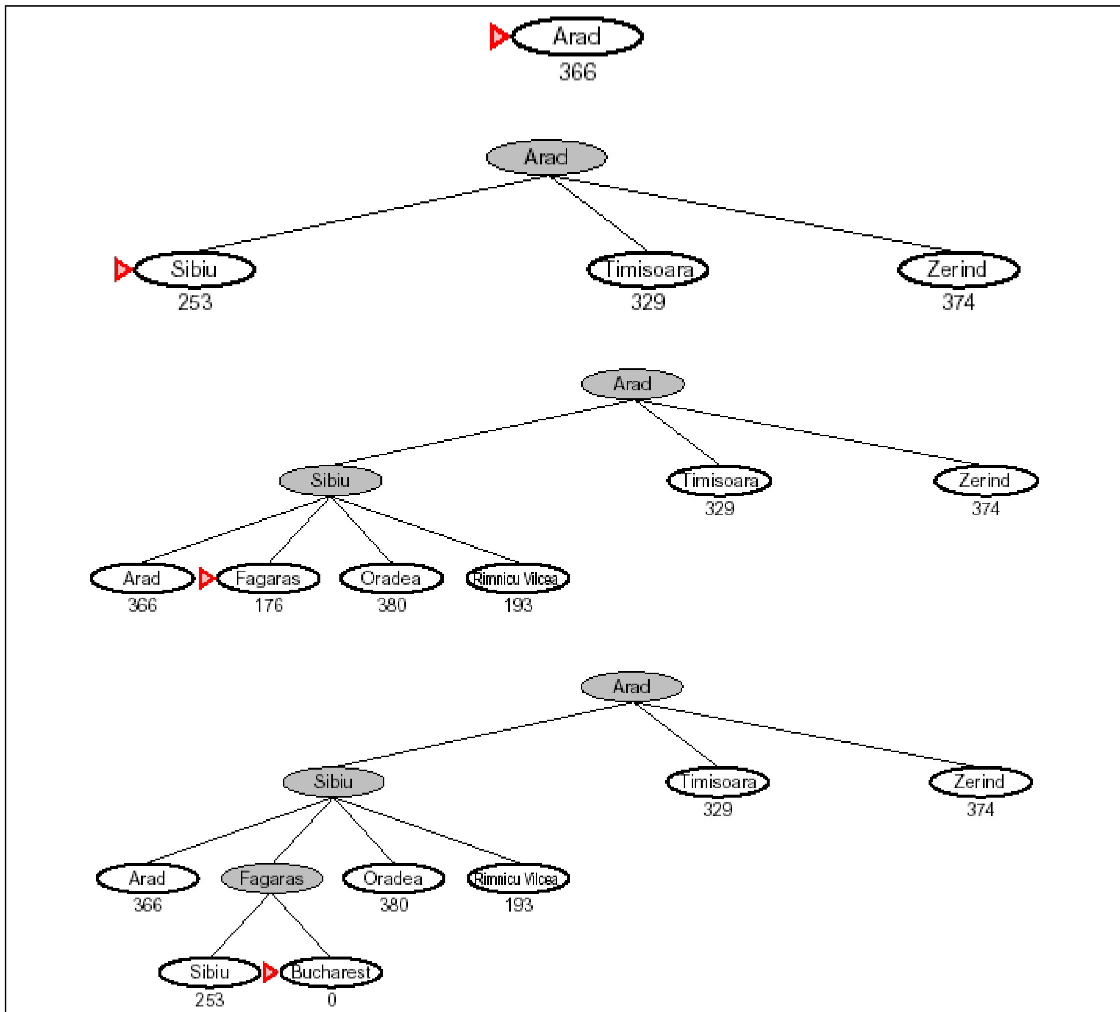


Figure 2.20 progress of greedy best-first search

Figure shows the progress of greedy best-first search using hSLD 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.

Properties of greedy search

- **Complete:** No–can get stuck in loops, e.g., Iasi !Neamt !Iasi !Neamt !

  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.

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** :

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

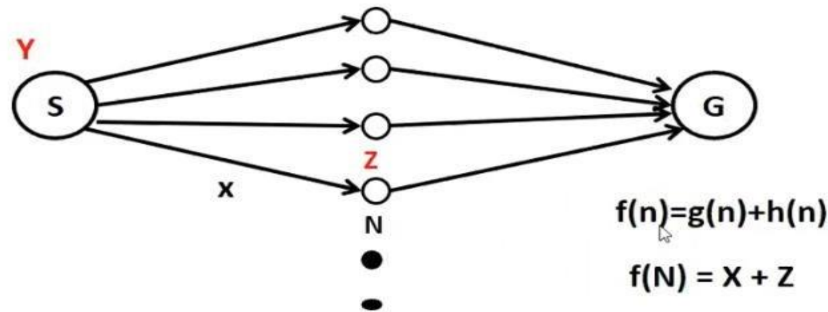
A^* Search is both optimal and complete. A^* is optimal if h(n) is an admissibleheuristic. The obvious example of admissible heuristic is the straight-line distance hSLD. It cannot be an overestimate.

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 hSLD 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). Also the values of hSLD are given in Figure

from Start node to reach Goal node there are in between nodes. Let us say there are n nodes in between which might take us to goal state



to reach from S to N , then g(n) is the actual cost from s to n. It's a cost which takes us to intermediate node N. And h(n) is the estimation cost from Node N to Goal node.

Figure 2.21 A* Search

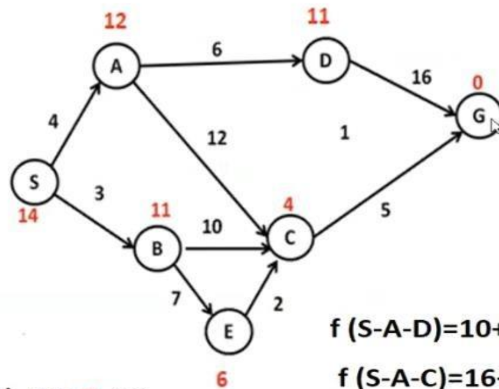


Figure 2.22 Example A* Search

2.17 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** (rather than multiple paths) 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.

2.18 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.

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**.

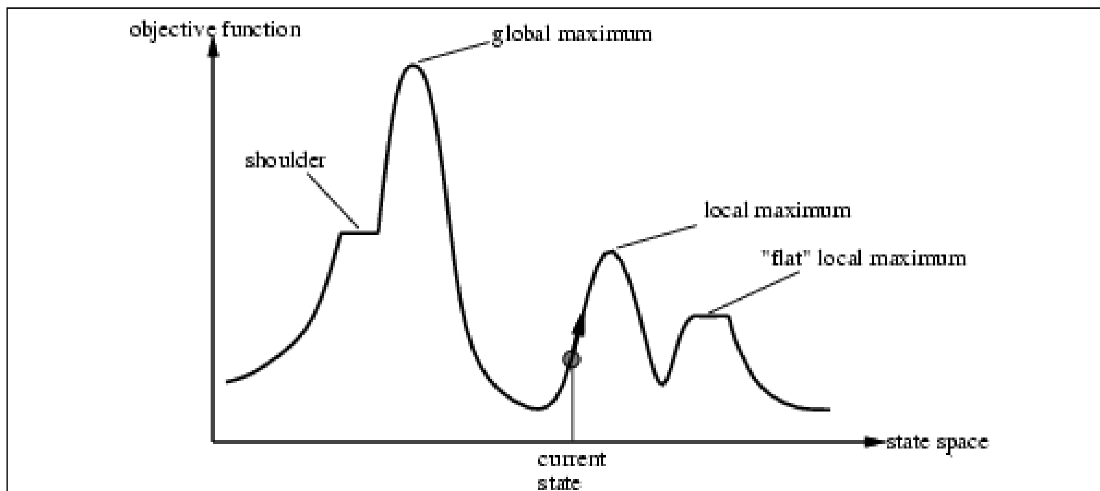


Figure 2.23 A one dimensional state space landscape in which elevation corresponds to the objective function. The aim is to find the global maximum. Hill climbing search modifies the current state to try to improve it, as shown by the arrow. The various topographic features are defined in the text

Hill-climbing search

The **hill-climbing** search algorithm as shown in figure, 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 2.24 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.

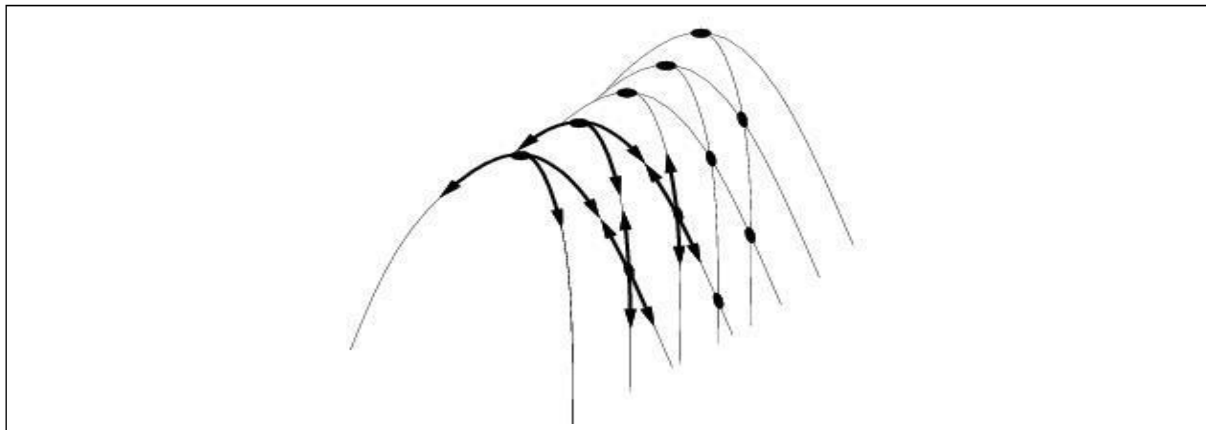


Figure 2.25 Illustration of why ridges cause difficulties for hill-climbing. The grid of states(dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available options point 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 choosenuniformly 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.

Figure shows simulated annealing algorithm. It is quite similar to hill climbing.Instead of picking the best move, however, it picks the random move. If the move improves the situation, it is always accepted. Otherwise, the algorithm accepts the move with some probability less than 1. The probability decreases exponentially with the "badness" of the move – the amount E by which the evaluation is worsened.

Simulated annealing was first used extensively to solve VLSI layout problems in the early 1980s. It has been applied widely to factory scheduling and other large-scale optimization tasks.

```
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 2.26 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 a column of 8 squares, and so requires 8 x log2 8 = 24 bits.

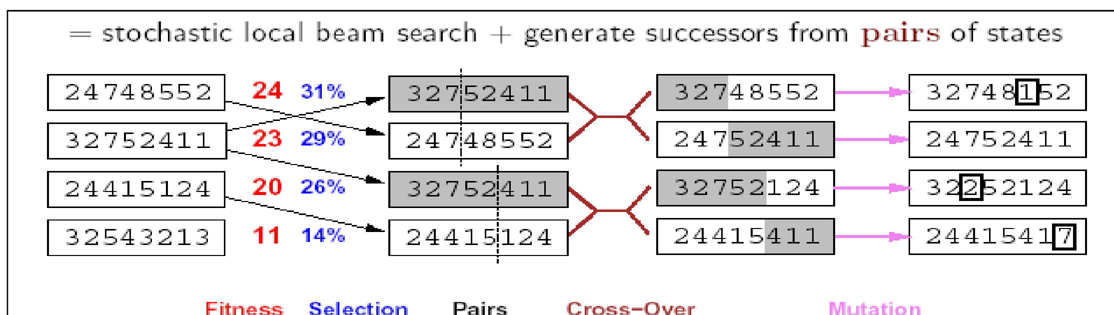


Figure 2.27 Genetic algorithm

Figure shows a population of four 8-digit strings representing 8-queen states. The production of the next generation of states is shown in Figure

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).

Figure describes the algorithm that implements all these steps.

```
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 ifrom 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 2.28 A genetic algorithm.

2.29 CONSTRAINT SATISFACTION PROBLEMS(CSP)

A **Constraint Satisfaction Problem**(or CSP) is defined by a set of **variables,**X1,X2, ….Xn, and a set of constraints C1,C2,…,Cm. Each variable Xi has a nonempty **domain** D,of possible **values**.

Each constraint Ci 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,{Xi = vi,Xj = vj,…}. An assignment that does not violate any constraints is called a **consistent** or **legal assignment.** A complete assignment is one in which every variable is mentioned, and a **solution** to a CSP is a complete assignment that satisfies all the constraints.

Some CSPs also require a solution that maximizes an **objective function**.

Example for Constraint Satisfaction Problem

Figure shows the map of Australia showing each of its states and territories. We are 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}.

It is helpful to visualize a CSP as a constraint graph, as shown in Figure 2.29. The nodes of the graph corresponds to variables of the problem and the arcs correspond to constraints.

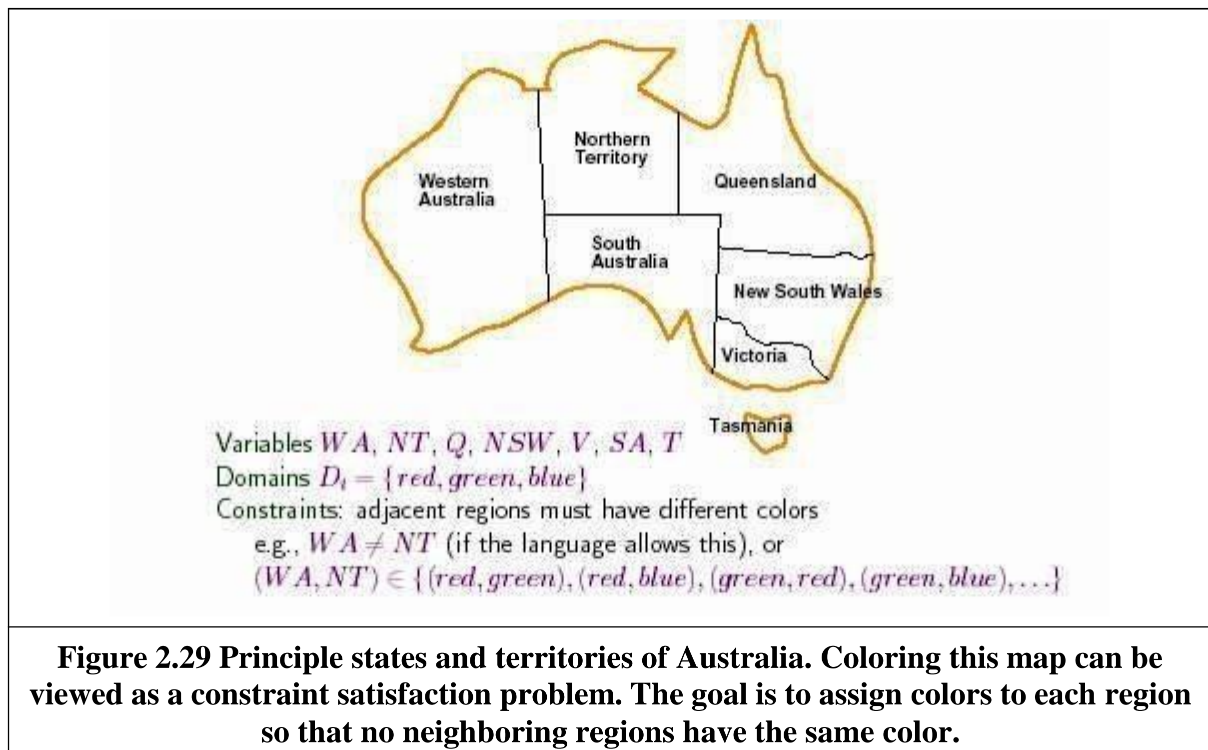


Figure 2.29 Principle states and territories of Australia. Coloring this map can be viewed as a constraint satisfaction problem. The goal is to assign colors to each region so that no neighboring regions have the same color.

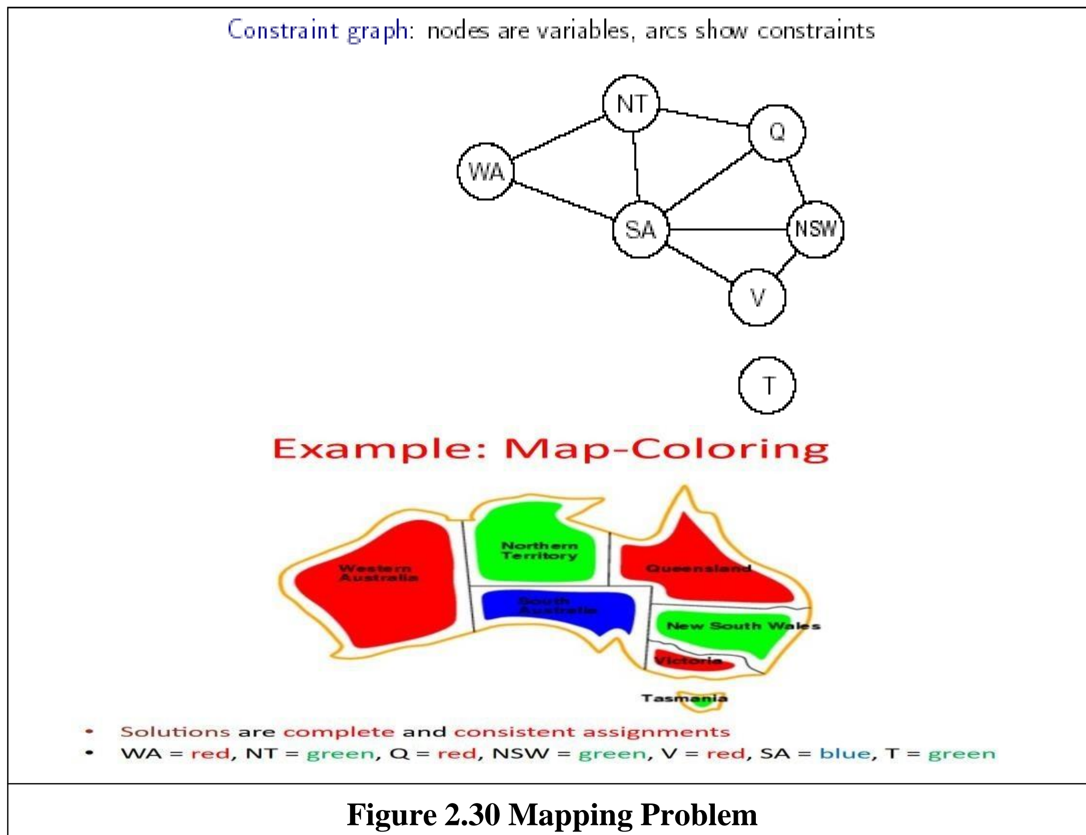


Figure 2.30 Mapping Problem

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 Q1,Q2,…..Q8 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 Startjob1 + 5 <= Startjob3.

(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 manoeuvre 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.

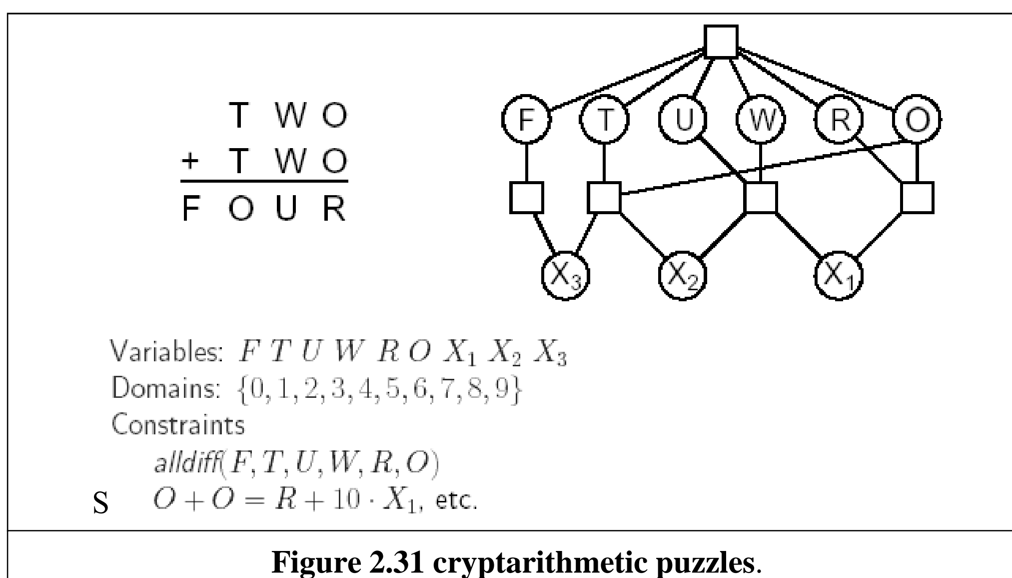


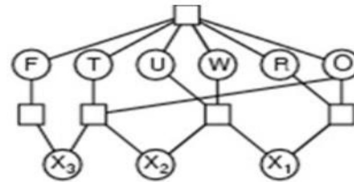
Figure 2.31 cryptarithmetic puzzles.

Example: Cryptarithmetic

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 zeroes are allowed

```
   T W O
 + T W O
 -------
 F O U R
```

- Variables: $F\ T\ U\ W\ R\ O\ X_1\ X_2 X_3$
- Domains: {0,1,2,3,4,5,6,7,8,9}
- Constraints: *Alldiff (F,T,U,W,R,O)*
- where XI, X2, and X3 are auxiliary variables representing the digit (0 or 1) carried over into the next column
  - $O + O = R + 10 \cdot X_1$
  - $X_1 + W + W = U + 10 \cdot X_2$
  - $X_2 + T + T = O + 10 \cdot X_3$
  - $X_3 = F,\ T \neq 0,\ F \neq 0$



The constraint hyper graph for the cryptarithmetic problem, showing the *Alldzff constraint as well as the column addition* constraints

Figure 2.32 Cryptarithmetic puzzles-Solution

- **Each puzzle may has one or many solutions or no solution**

```
   T W O
 + T W O
 -------
 F O U R
```

has 7 solutions:

```
1.  734   F=1 O=4 R=8 T=7 U=6 W=3
   +734
   --------
   1468

2.  765   F=1 O=5 R=0 T=7 U=3 W=6
   +765
   --------
   1530

3.  836   F=1 O=6 R=2 T=8 U=7 W=3
   +836
   ---------
   1672

4.  846   F=1 O=6 R=2 T=8 U=9 W=4
   +846
   ---------
   1692

5.  867   F=1 O=7 R=4 T=8 U=3 W=6
   +867
   ---------
   1734

6.  928   F=1 O=8 R=6 T=9 U=5 W=2
   +928
   ----------
   1856

7.  938   F=1 O=8 R=6 T=9 U=7 W=3
   +938
   -----------
   1876
```

Figure 2.33 Numerical Solution

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

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

Backtracking example

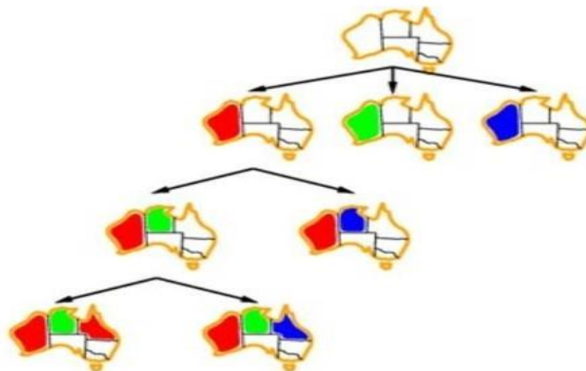


Figure 2.34 Part of the search tree generated by simple backtracking for the map-coloring problem

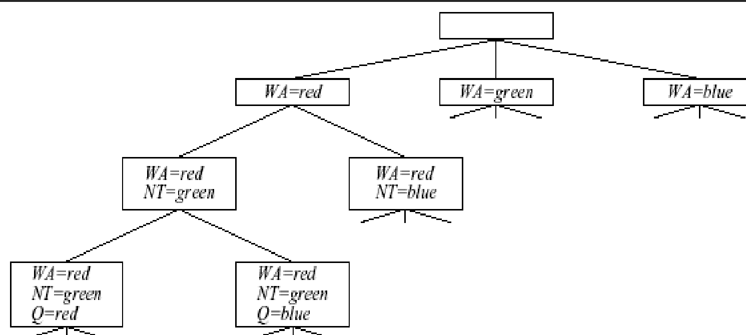


Figure 2.35 Part of search tree generated by simple backtracking for the map coloring problem.

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.

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	(R)	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	(R)	B	(G)	R B	R G B	B	R G B
After *V*=*blue*	(R)	B	(G)	R	(B)		R G B

Figure 2.36 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 domain of *NT, SA*, and *NSW*. After *V* = *blue, blue,* is deleted from the domains of *NSW* and *SA*, leaving *SA* with no legal values.

Constraint propagation

Although forward checking detects many inconsistencies, it does not detect all of them.

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

Arc Consistency

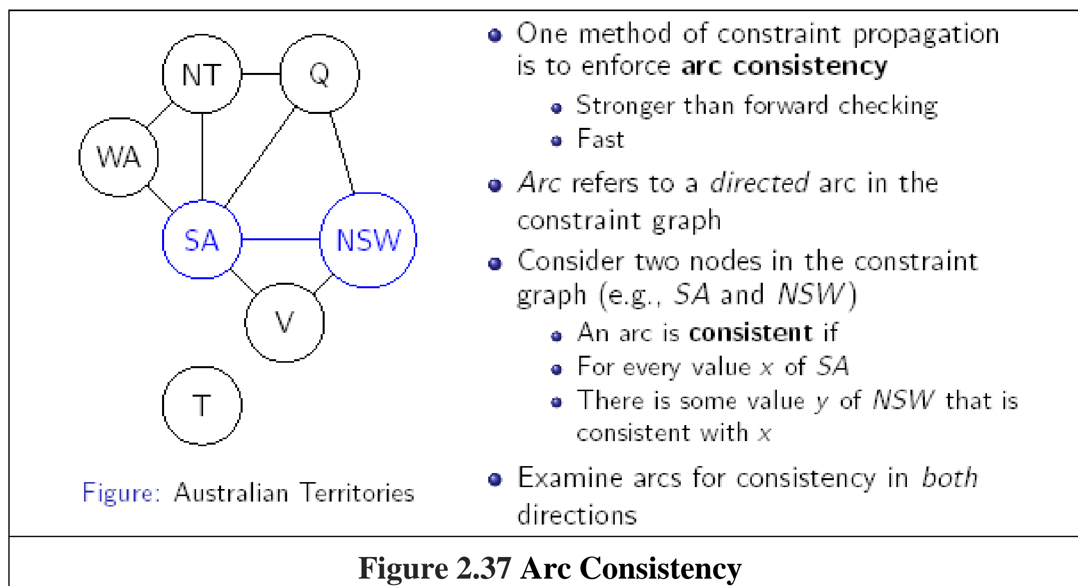


Figure 2.37 Arc Consistency

- 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

Figure 2.38 Arc Consistency –CSP

k-Consistency

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

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

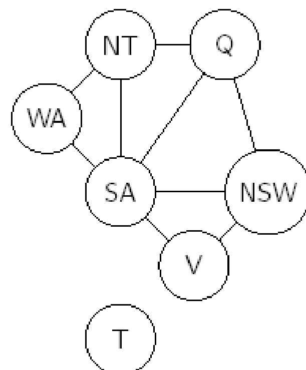


Figure: Australian Territories

- 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

Figure 2.39 Independent Subproblems

Tree-Structured CSPs

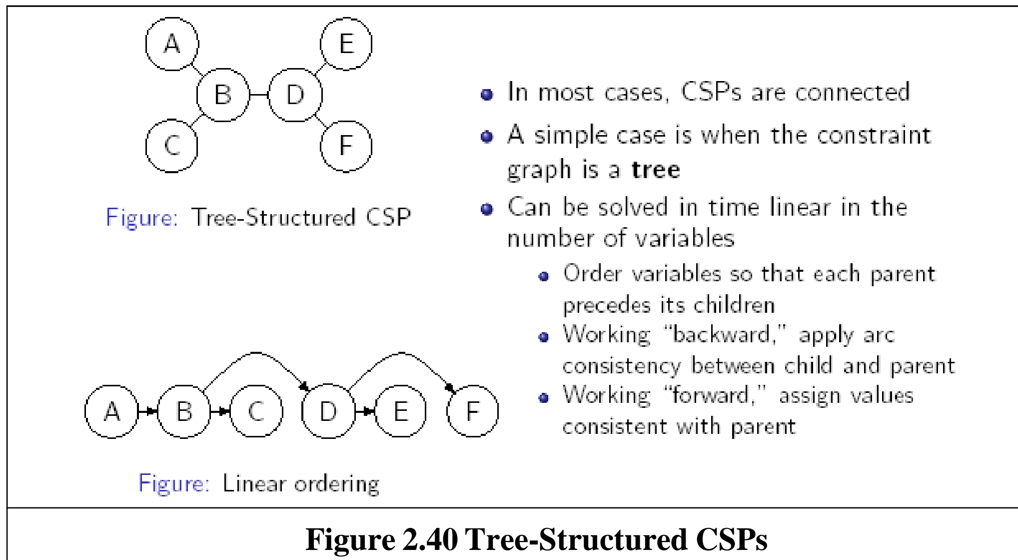


Figure 2.40 Tree-Structured CSPs

2.30 ADVERSARIAL SEARCH

Competitive environments, in which the agent's goals are in conflict, give rise to **adversarial search** problems – often known as **games.**

Games

Mathematical **Game Theory**, a branch of economics, views any **multiagent environment** as a **game** provided that the impact of each agent on the other is "significant", regardless of whether the agents are cooperative or competitive. In, **AI**, "games" are deterministic, turn-taking, two-player, zero-sum games of perfect information. This means deterministic, fully observable environments in which there are two agents whose actions must alternate and in which the **utility values** at the end of the game are always equal and opposite. For example, if one player wins the game of chess(+1),the other player necessarily loses(-1). It is this opposition between the agents' utility functions that makes the situation **adversarial.**

Formal Definition of Game

We will consider games with two players, whom we will call **MAX** and **MIN**. MAX moves first, and then they take turns moving until the game is over. At the end of the game, points are awarded to the winning player and penalties are given to the loser. A **game** can be formally defined as a **search problem** with the following components:

- The **initial state**, which includes the board position and identifies the player to move.
- A **successor function**, which returns a list of (*move, state*) pairs, each indicating a legal move and the resulting state.

- o A **terminal test**, which describes when the game is over. States where the game has ended are called **terminal states**.

- o A **utility function** (also called an objective function or payoff function), which give a numeric value for the terminal states. In chess, the outcome is a win, loss, or draw, with values+1,-1, or 0. he payoffs in backgammon range from +192 to -192.

Game Tree

The **initial state** and **legal moves** for each side define the **game tree** for the game. Figure 2.18 shows the part of the game tree for tic-tac-toe (noughts and crosses). From the initial state, MAX has nine possible moves. Play alternates between MAX's placing an X and MIN's placing a 0 until we reach leaf nodes corresponding to the terminal states such that one player has three in a row or all the squares are filled. He number on each leaf node indicates the utility value of the terminal state from the point of view of MAX; high values are assumed to be good for MAX and bad for MIN. It is the MAX's job to use the search tree (particularly the utility of terminal states) to determine the best move.

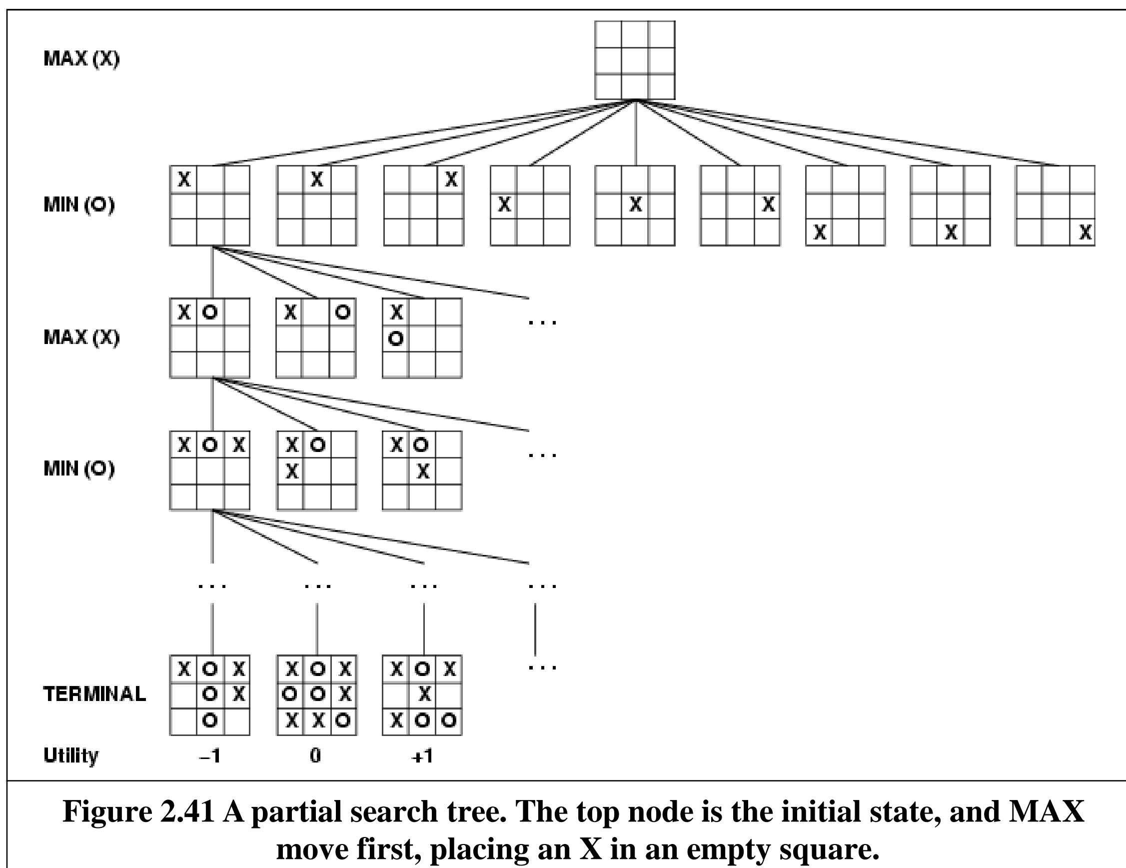


Figure 2.41 A partial search tree. The top node is the initial state, and MAX move first, placing an X in an empty square.

Optimal Decisions in Games

In normal search problem, the **optimal solution** would be a sequence of move leading to a **goal state** – a terminal state that is a win. In a game, on the other hand, MIN has something

to say about it, MAX therefore must find a contingent **strategy**, which specifies MAX's move in the **initial state**, then MAX's moves in the states resulting from every possible response by MIN, then MAX's moves in the states resulting from every possible response by MIN those moves, and so on. An **optimal strategy** leads to outcomes at least as good as any other strategy when one is playing an infallible opponent.

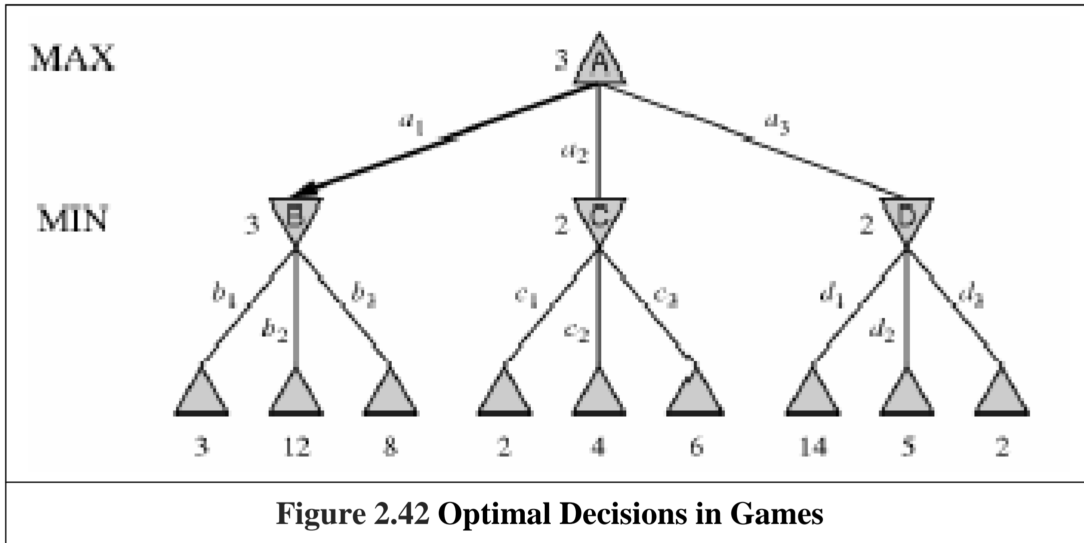


Figure 2.42 Optimal Decisions in Games

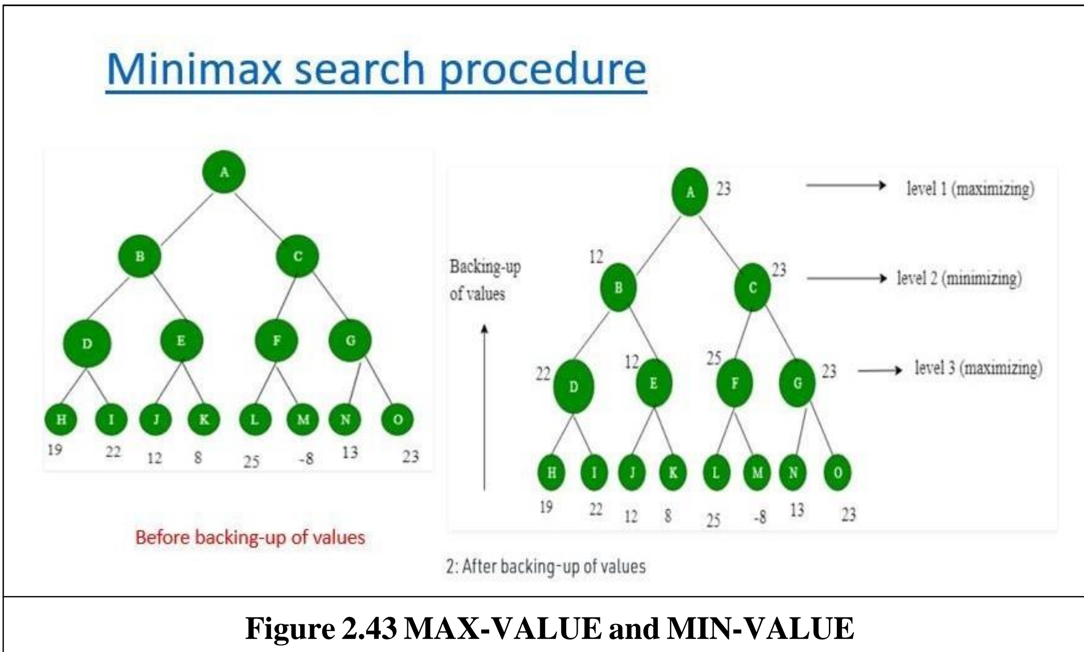


Figure 2.43 MAX-VALUE and MIN-VALUE

Minimax Search: Algorithm

```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game

   v ← MAX-VALUE(state)
   return the action in SUCCESSORS(state) with value v

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

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

For MAX Node

For MIN Node

Figure 2.44 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 minimax Algorithm

The minimax algorithm computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds**.** For example in Figure 2.19,the algorithm first recourses down to the three bottom left nodes, and uses the utility function on them to discover that their values are 3, 12, and 8 respectively. Then it takes the minimum of these values, 3, and returns it as the backed-up value of node B. A similar process gives the backed up values of 2 for C and 2 for D. Finally, we take the maximum of 3, 2, and 2 to get the backed-up value of 3 at the root node. The minimax algorithm performs a complete depth-first exploration of the game tree. If the maximum depth of the tree is *m*, and there are b legal moves at each point, then the time complexity of the minimax algorithm is $O(b^m)$. The space complexity is O(bm) for analgorithm that generates successors at once.

Alpha-Beta Pruning

The problem with minimax search is that the number of game states it has to examine is **exponential** in the number of moves. Unfortunately, we can’t eliminate the exponent, but

we can effectively cut it in half. By performing **pruning,** we can eliminate large part of the tree from consideration. We can apply the technique known as **alpha beta pruning**, when applied to a minimax tree, it returns the same move as **minimax** would, but **prunes away** branches that cannot possibly influence the final decision.

Alpha Beta pruning gets its name from the following two parameters that describe bounds on the backed-up values that appear anywhere along the path:

- α : the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path of MAX.
- β: the value of best (i.e., lowest-value) choice we have found so far at any choice point along the path of MIN.

Alpha Beta search updates the values of α and β as it goes along and prunes the remaining branches at anode(i.e., terminates the recursive call) as soon as the value of the current node is known to be worse than the current α and β value for MAX and MIN, respectively. The complete algorithm is given in Figure. The effectiveness of alpha-beta pruning is highly dependent on the order in which the successors are examined. It might be worthwhile to try to examine first the successors that are likely to be the best. In such case, it turns out that alpha-beta needs to examine only $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ for minimax. This means that the effective branching factor becomes sqrt(b) instead of b – for chess,6 instead of 35. Put an other way alpha-beta cab look ahead roughly twice as far as minimax in the same amount of time.

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

function MAX-VALUE(state, α, β) returns a utility value
   inputs: state, current state in game
           α, the value of the best alternative for MAX along the path to state
           β, the value of the best alternative for MIN along the path to state
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do
      v ← MAX(v, MIN-VALUE(s, α, β))
      if v ≥ β then return v
      α ← MAX(α, v)
   return v
```

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

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

Figure 2.45 The alpha beta search algorithm. These routines are the same as the minimax routines in figure 2.20,except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain α and β

Key points in Alpha-beta Pruning

- Alpha: Alpha is the best choice or the highest value that we have found at any instance along the path of Maximizer. The initial value for alpha is $-\infty$.
- Beta: Beta is the best choice or the lowest value that we have found at any instance along the path of Minimizer. The initial value for alpha is $+\infty$.
- The condition for Alpha-beta Pruning is that $\alpha >= \beta$.
- Each node has to keep track of its alpha and beta values. Alpha can be updated only when it's MAX's turn and, similarly, beta can be updated only when it's MIN's chance.
- MAX will update only alpha values and MIN player will update only beta values.
- The node values will be passed to upper nodes instead of values of alpha and beta during go into reverse of tree.
- Alpha and Beta values only be passed to child nodes.

Working of Alpha-beta Pruning

1. We will first start with the initial move. We will initially define the alpha and beta values as the worst case i.e. $\alpha = -\infty$ and $\beta = +\infty$. We will prune the node only when alpha becomes greater than or equal to beta.

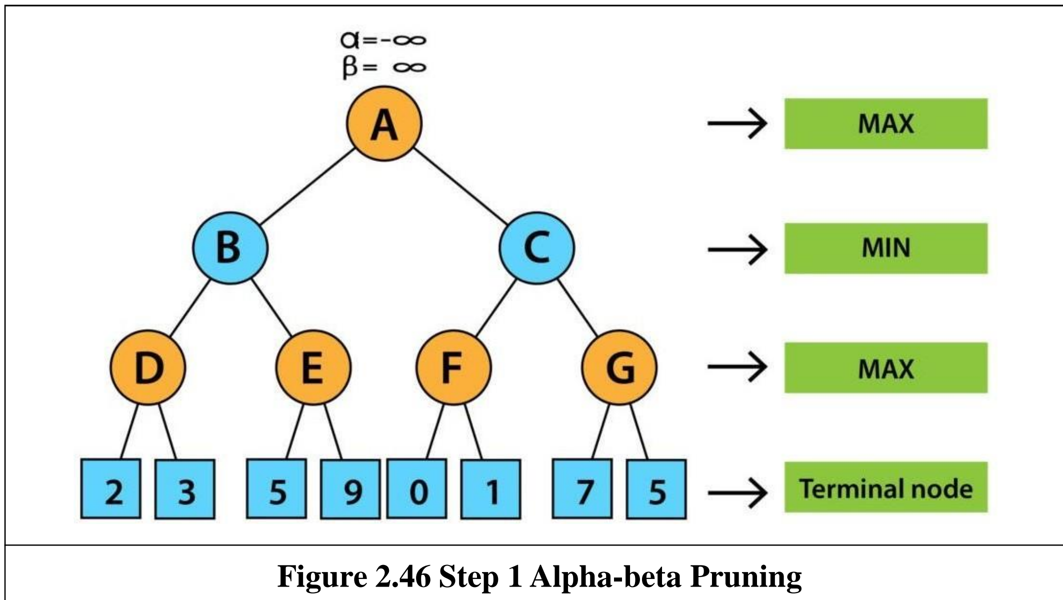


Figure 2.46 Step 1 Alpha-beta Pruning

2. Since the initial value of alpha is less than beta so we didn't prune it. Now it's turn for MAX. So, at node D, value of alpha will be calculated. The value of alpha at node D will be max (2, 3). So, value of alpha at node D will be 3.

3. Now the next move will be on node B and its turn for MIN now. So, at node B, the value of alpha beta will be min (3, ∞). So, at node B values will be alpha= – ∞ and beta will be 3.

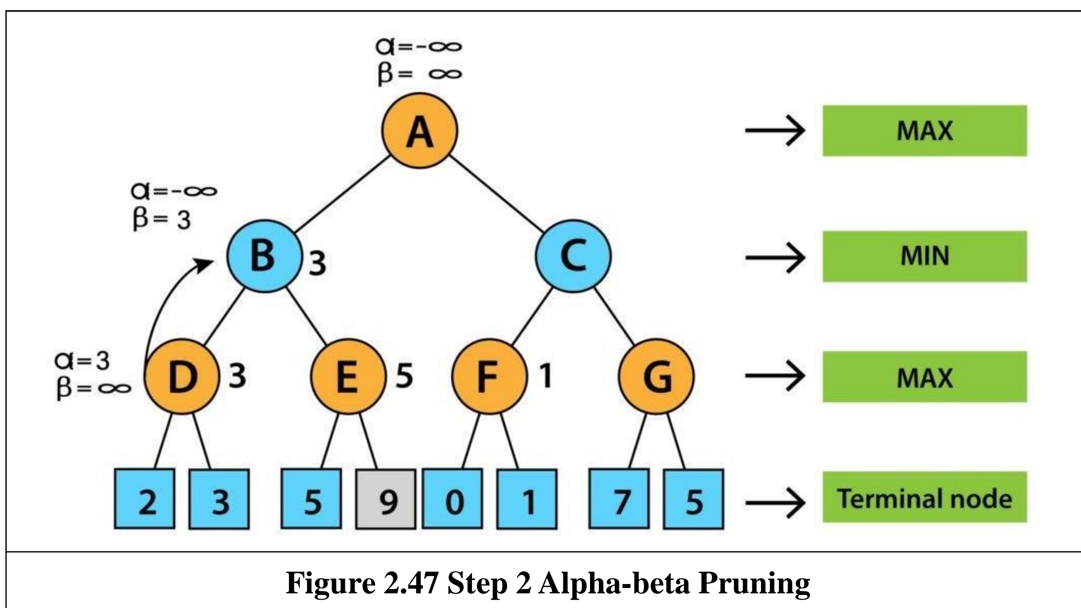


Figure 2.47 Step 2 Alpha-beta Pruning

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

4. Now it's turn for MAX. So, at node E we will look for MAX. The current value of alpha at E is – ∞ and it will be compared with 5. So, MAX (- ∞, 5) will be 5. So, at node E, alpha = 5, Beta = 5. Now as we can see that alpha is greater than beta which is satisfying the pruning condition so we can prune the right successor of node E and algorithm will not be traversed and the value at node E will be 5.

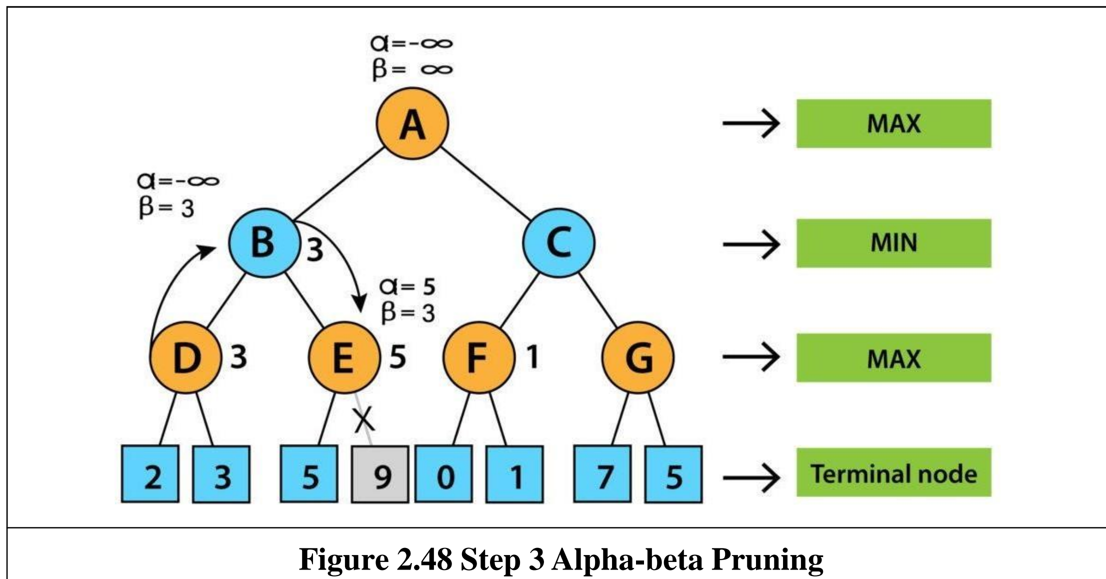


Figure 2.48 Step 3 Alpha-beta Pruning

6. In the next step the algorithm again comes to node A from node B. At node A alpha will be changed to maximum value as MAX (- ∞, 3). So now the value of alpha and beta at node A will be (3, + ∞) respectively and will be transferred to node C. These same values will be transferred to node F.

7. At node F the value of alpha will be compared to the left branch which is 0. So, MAX (0, 3) will be 3 and then compared with the right child which is 1, and MAX (3,1) = 3 still α remains 3, but the node value of F will become 1.

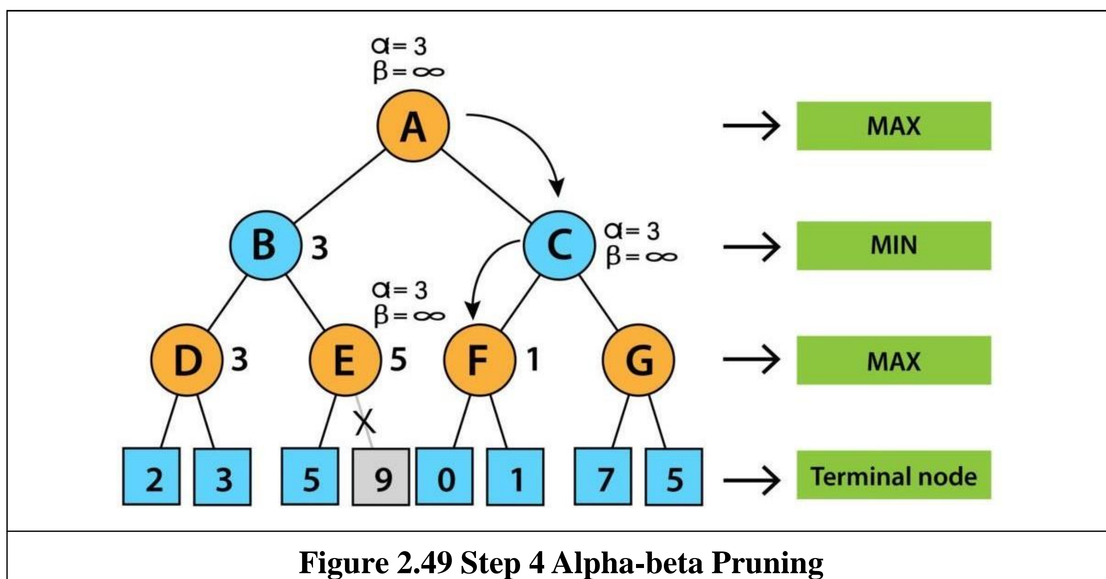


Figure 2.49 Step 4 Alpha-beta Pruning

8. Now node F will return the node value 1 to C and will compare to beta value at C. Now its turn for MIN. So, MIN (+ ∞, 1) will be 1. Now at node C, α= 3, and β= 1 and alpha is greater than beta which again satisfies the pruning condition. So, the next successor of node C i.e. G will be pruned and the algorithm didn't compute the entire subtree G.

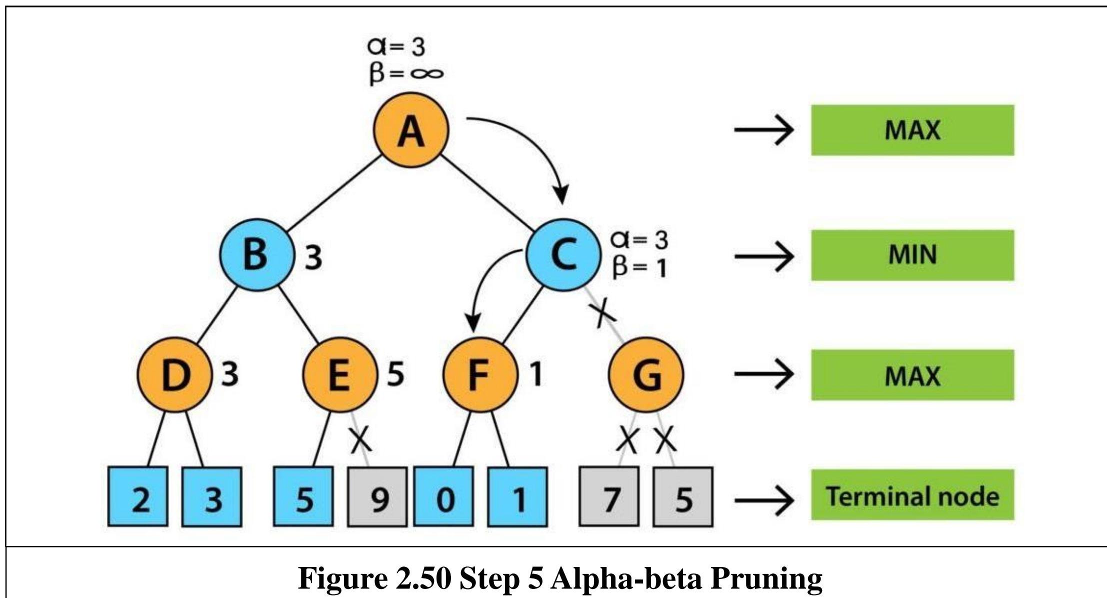


Figure 2.50 Step 5 Alpha-beta Pruning

Now, C will return the node value to A and the best value of A will be MAX (1, 3) will be 3.

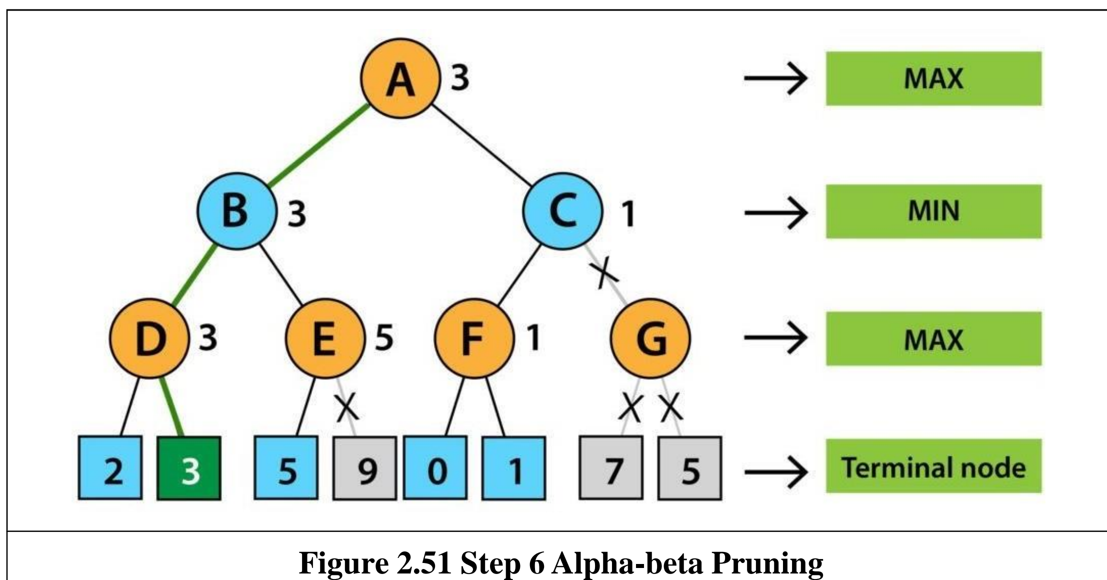


Figure 2.51 Step 6 Alpha-beta Pruning

The above represented tree is the final tree which is showing the nodes which are computed and the nodes which are not computed. So, for this example the optimal value of the maximizer will be 3.