# UNIT I

Introduction: History of AI - problem spaces and search- Heuristic Search techniques –Best-first search- Problem reduction-Constraint satisfaction-Means Ends Analysis. Intelligent agents: Agents and environment – structure of agents and its functions

# INTRODUCTION

**ARTIFICIAL INTELLIGENCE**

Artificial Intelligence is a branch of computer science that deals with the creation of computer programs that can provide solutions, otherwise human would have to solve.

Artificial Intelligence definitions are given on the basis of

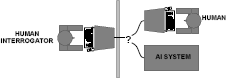
1. Based on thought process and reasoning.
2. Based on the behavior. (iii)Based on human performance. (iv)Based on Rationality.

# Views of AI fall into four categories:

|  |  |
| --- | --- |
| **Thinking humanly**  “The automation of activities that we associate with human thinking, activities such as decision-making,  problem solving, learning…” | **Thinking rationally**  “ The study of mental faculties through the use of computational models” |
| **Acting humanly**  “The study of how to make computers do things at which, at the moment, people are better” | **Acting rationally** “Computational intelligence is the study of the design of intelligent  agents” |

**Acting humanly: Turing Test**

* + - The Turing Test, Proposed by Alan Turing (1950) ,was designed to provide a satisfactory operational definition of Intelligence ."Computing machinery and intelligence":
    - "Can machines think?"  "Can machines behave intelligently?"
    - Operational test for intelligent behavior: The Imitation Game



* Predicted that by 2000, a machine might have a 30% chance of fooling a person for 5 minutes
* Anticipated all major arguments against AI in following 50 years
* Suggested major components of AI: knowledge, reasoning, language understanding, learning.

# Turing test

•Three rooms contain a person, a computer, and an interrogator.

•The interrogator can communicate with the other two by teleprinter.

•The interrogator tries to determine which the person is and which the machine is.

•The machine tries to fool the interrogator into believing that it is the person.

* + If the machine succeeds, then we conclude that the machine can think. The computer would 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;
* ***A*utomated 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.
* **Computer vision** to perceive objects.
* **Robotics** to manipulate objects and move about.

# Thinking humanly: cognitive modeling

* In 1960’s "cognitive revolution": information-processing psychology
* Requires scientific theories of internal activities of the brain
* How to validate? Requires

1. Predicting and testing behaviour of human subjects (top-down)
2. Direct identification from neurological data (bottom-up)

* Both approaches (roughly, Cognitive Science and Cognitive Neuroscience) are now distinct from Artificial Intelligence

Try to understand how the mind works. How do we think?

Two possible routes to find answers:

* + By introspection: We figure it out ourselves!
  + By experiment: Draw upon techniques of psychology to conduct controlled experiments. (Rat in a box!)

# Thinking rationally: "laws of thought"

* Aristotle: what are correct arguments/thought processes?
* Several Greek schools developed various forms of logic: notation and rules of derivation for thoughts; may or may not have proceeded to the idea of mechanization
* Direct line through mathematics and philosophy to modern AI
* Problems:
  + Not all intelligent behaviour is mediated by logical deliberation
  + What is the purpose of thinking? What thoughts should I have?
* Trying to understand how we actually think is one route to AI. But how about how we should think.
* Use logic to capture the laws of rational thought as symbols.
* Reasoning involves shifting symbols according to well-defined rules (like algebra).
* Result is idealized reasoning.

# Acting rationally: rational agent

* Rational behaviour: doing the right thing
* The right thing: that which is expected to maximize goal achievement, given the available information
* Doesn't necessarily involve thinking – e.g., blinking reflex – but thinking should be in the service of rational action

# Rational agents

* An agent is an entity that perceives and acts
* This course is about designing rational agents
* Abstractly, an agent is a function from percept histories to actions: [f: P\*  A]
* For any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.
* Caveat: computational limitations make perfect rationality unachievable.

 Design best program for given machine resources.

# HISTORY OF ARTIFICIAL INTELLIGENCE

* 1. **The gestation of artificial intelligence**
     + Pitts and McCulloch (1943): simplified mathematical model of neurons (resting/firing states) can realize all propositional logic primitives (can compute all Turing computable functions)
     + Allen Turing: Turing machine and Turing test (1950)

They drew on three sources: knowledge of the basic physiology and function of neurons in the brain; a formal analysis of propositional logic. They proposed a model of artificial neurons in which each neuron is characterized as being "on" or "off," with a switch to "on" occurring in response to stimulation by a sufficient number of neighboring neurons. A simple updating rule for modifying the connection strengths between neurons. This rule, now called as Hebbian learning.

* + - Claude Shannon: information theory; possibility of chess playing computers
    - Tracing back to Boole, Aristotle, Euclid (logics, syllogisms)

# The birth of artificial intelligence (1956)

* U.S. researchers interested in automata theory, neural nets, and the study of intelligence. They organized a two-month workshop at **Dartmouth in the summer of 1956.**
* The others had ideas and in some cases programs for particular applications such as checkers.
* They invented a computer program capable of thinking non-numerically, and thereby solved the venerable mind-body problem.

# Early enthusiasm, great expectations (1952-1969)

* Given the primitive computers and programming tools of the time, and the fact that only a few years earlier computers were seen as things that could do arithmetic and no more.
* **Newel and Simon's** early success was followed up with the General Problem Solver.
* Unlike Logic Theorist, this program was designed from the start to imitate human problem-solving protocols. Within the limited class of puzzles it could handle, it turned out that the order in which the program considered subgoals and possible actions was similar to that in which humans approached the same problems

# A dose of reality (1966-1973)

* The first kind of difficulty arose because most early programs contained little or no knowledge of their subject matter; they succeeded by means of simple syntactic manipulations.
* The second kind of difficulty was the intractability of many of the problems that A1 was attempting to solve. Most of the early **A1** programs solved problems by trying out different combinations of steps until the solution was found.
* The illusion of unlimited computational power was not confined to problem-solving programs.
* A third difficulty arose because of some fundamental limitations on the basic structures being used to generate intelligent behavior.

# Knowledge-based systems:

* A general-purpose search mechanism trying to string together elementary reasoning steps to find complete solutions. Such approaches have been called weak methods, because, although general, they do not scale up to large or difficult problem instances. The alternative to weak methods is to use more powerful, domain-specific knowledge that allows larger reasoning steps and can more easily handle.

# AI becomes an industry:

* In 1981, the Japanese announced the "Fifth Generation" project, a 10-year plan to build intelligent computers running Prolog. In response the United States formed the **Microelectronics and Computer Technology Corporation (MCC**) as a research consortium designed to assure national competitiveness. In both cases, A1 was part of a broad effort, including chip design and human-interface research. However, the A1 components of MCC and the Fifth Generation projects never met their ambitious goals.
* Overall, the AI industry boomed from a few million dollars in **1980** to billions of dollars in **1988.**

# The return of neural networks (1986-present)

* Although computer science had largely abandoned the field of neural networks in the late 1970s, work continued in other fields.
* The algorithm was applied to many learning problems in computer science and psychology, and the widespread dissemination of the results in the collection Parallel Distributed Processing.

# AI becomes a science (1987-present)

* Recent years have seen a revolution in both the content and the methodology of work in artificial intelligence.
* It is now more common to build on existing theories than to propose brand new ones, to base claims on rigorous theorems or hard experimental evidence rather than on intuition, and to show relevance to real-world applications rather than toy examples.

# The emergence of intelligent agents (1995-present)

* AI, researchers have also started to look at the "whole agent" problem again.
* One of the most important environments for intelligent agents is the Internet. AS systems have become so common in web-based applications that the "-bot" suffix has entered everyday language. Moreover, AS technologies underlie many Internet tools, such as search engines, recommender systems, and Web site construction systems.
* A second major consequence of the agent perspective is that A1 has been drawn into much closer contact with other fields, such as control theory and economics that also deal with agents.

# APPLICATIONS OF AI Autonomous planning and scheduling

1. Route planning 2. Automated scheduling of actions in spacecrafts

# Game playing

* IBM's Deep Blue defeated G.Kasparov (the human world champion) (1997)
* The program FRITZ running on an *ordinary PC* drawed with V.Kramnik (the human world champion) (2002)

# Autonomous control

* Automated car steering and the Mars mission.

# Diagnosis

* Medical diagnosis programs based on probabilistic analysis have been able to perform at the level of an expert physician in several areas of medicine.
* Literature describes a case where a leading expert was convinced by a computer diagnostic.

# Logistic planning

* + Defence Advanced Research Project Agency stated that this single application more than paid back DARPA's 30-year investment in AI

# Robotics

* Microsurgery and RoboCup. By the year 2050, develop a team of fully autonomous humanoid robots that can win against the human world soccer champion team.

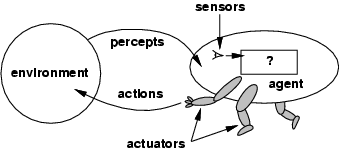
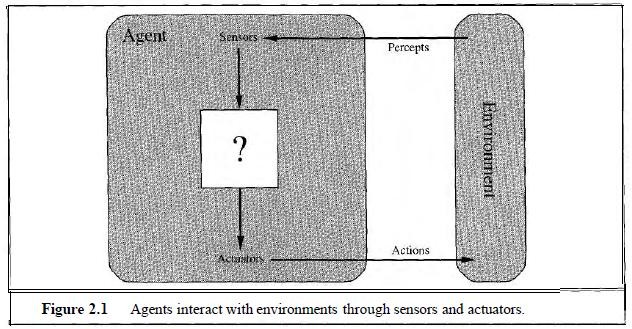
# INTELLIGENT AGENTS AND ITS ENVIRONMENTS

* 1. **AGENTS AND ENVIRONMENTS**

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

* **Human agent:** eyes, ears, and other organs for sensors; hands, legs, mouth, and other body parts for actuators
* **Robotic agent:** cameras and infrared range finders for sensors; 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.
* We use the term **percept** to refer to the agent's perceptual inputs at any given instant. An agent's

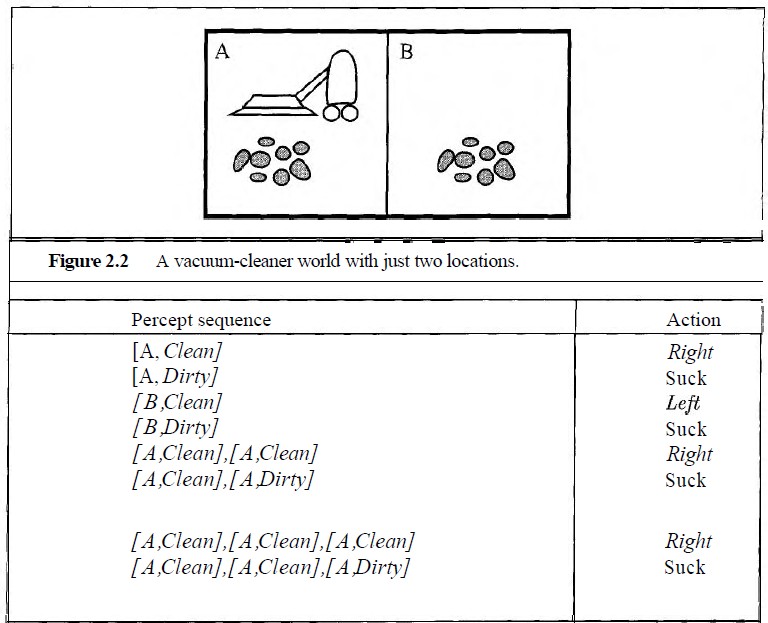
**percept sequence** is the complete history of everything the agent has ever perceived.



* + We can imagine tabulating the agent function that describes any given agent; for most agents, this would be a very large table-infinite, in fact, unless we place a bound on the length of percept sequences we want to consider.
  + Construct this table by trying out all possible percept sequences and recording which actions the agent does in response.
  1. **AGENT PROGRAM**:

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

This world is so simple that we can describe everything that happens; it's also a made-up world, so we can invent many variations. This particular world has just two locations: squares A and B. The vacuum agent perceives which square it is in and1 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.



* 1. **GOOD BEHAVIOUR: THE CONCEPT OF RATIONALITY**

Rational agents

* An agent should strive to "do the right thing", based on what it can perceive and the actions it can perform. The right action is the one that will cause the agent to be most successful.
* Performance measure: An objective criterion for success of an agent's behavior
* E.g., performance measure of a vacuum-cleaner agent could be amount of dirt cleaned up, amount of time taken, amount of electricity consumed, amount of noise generated, etc.
* **Rational Agent:** For each 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.
* Rationality is distinct from omniscience (all-knowing with infinite knowledge)
* Agents can perform actions in order to modify future percepts so as to obtain useful information (information gathering, exploration)
* An agent is autonomous if its behavior is determined by its own experience (with ability to learn and adapt)

# PEAS

* + - PEAS: Performance measure, Environment, Actuators, Sensors
    - Must first specify the setting for intelligent agent design
    - Consider, e.g., the task of designing an automated taxi driver:
      * Performance measure
      * Environment
      * Actuators
      * Sensors
    - Must first specify the setting for intelligent agent design
    - Consider, e.g., the task of designing an automated taxi driver:
      * Performance measure: Safe, fast, legal, comfortable trip, maximize profits
      * Environment: Roads, other traffic, pedestrians, customers
      * Actuators: Steering wheel, accelerator, brake, signal, horn
      * Sensors: Cameras, sonar, speedometer, GPS, odometer, engine sensors, keyboard Agent: Medical diagnosis system
    - Performance measure: Healthy patient, minimize costs, lawsuits
    - Environment: Patient, hospital, staff
    - Actuators: Screen display (questions, tests, diagnoses, treatments, referrals)
    - Sensors: Keyboard (entry of symptoms, findings, patient's answers)

Agent: Part-picking robot

* + - Performance measure: Percentage of parts in correct bins
    - Environment: Conveyor belt with parts, bins
    - Actuators: Jointed arm and hand
    - Sensors: Camera, joint angle sensors Agent: Interactive English tutor
    - Performance measure: Maximize student's score on test
    - Environment: Set of students
    - Actuators: Screen display (exercises, suggestions, corrections)
    - Sensors: Keyboard
  1. **ENVIRONMENT TYPES**
     + **Fully observable (vs. partially observable)**: An agent's sensors give it access to the complete state of the environment at each point in time.
     + **Deterministic (vs. stochastic):** The next state of the environment is completely determined by the current state and the action executed by the agent. (If the environment is deterministic except for the actions of other agents, then the environment is strategic)
     + **Episodic (vs. sequential):** The agent's experience is divided into atomic "episodes" (each episode consists of the agent perceiving and then performing a single action), and the choice of action in each episode depends only on the episode itself.
     + **Static (vs. dynamic):** The environment is unchanged while an agent is deliberating. (The environment is semi dynamic if the environment itself does not change with the passage of time but the agent's performance score does)
     + **Discrete (vs. continuous):** A limited number of distinct, clearly defined percepts and actions***.***
     + **Single agent (vs. multiagent):** An agent operating by itself in an environment.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Chess with a**  **clock** | **Chess without**  **a clock** | **Taxi Driving** |
| Fully | Yes | Yes | No |
| Deterministic | Strategic | Strategic | No |
| Episodic | No | No | No |
| Static | Semi | Yes | No |
| Discrete | Yes | Yes | No |
| Single agent | No | No | No |

* The environment type largely determines the agent design.
* The real world is (of course) partially observable, stochastic, sequential, dynamic, continuous, multi-agent.

# STRUCTURE OF AGENTS AND ITS FUNCTIONS

* 1. **AGENT FUNCTIONS AND PROGRAMS**
* An agent is completely specified by the agent function mapping percept sequences to actions.
* One agent function (or a small equivalence class) is rational.
* Aim: find a way to implement the rational agent function concisely. Table-lookup agent and its Drawbacks:
  + Huge table
  + Take a long time to build the table
  + No autonomy
  + Even with learning, need a long time to learn the table entries
  1. **AGENT TYPES**
* Four basic types in order of increasing generality:
* Simple reflex agents
* Model-based reflex agents
* Goal-based agents
* Utility-based agents
* Learning agents

# simple-reflex-agentSimple reflex agents

The simplest kind of agent is the simplex reflex agent. These agents select actions on the basis of the current percept, ignoring the rest of the percept history.

# Example: Agent function for vacuum agent

**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.

[A, clean],[A, clean], [A, clean] Right

[A, clean],[A, clean], [A, dirty] Suck An agent program for this agent is

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

The vacuum agent program is very small. But some processing is done on the visual input to establish the condition-action rule.

**Rectangles:** Current internal state of the agent’s decision process

**Ovals:** Background information used in the process The **agent program** is given below:

Function SIMPLE-REFLEX-AGENT (percept) returns an action Static: rules, a set of condition-action rules

State<- INTERPRET-INPUT (percept) Rule<- RULE-MATCH (state, rules) Action <- RULE-ACTION [rule]

Return action

# Function

INTERPRET-INPUT: generates an abstracted description of the current state from the percept RULE-MATCH: returns the first rule in the set of rules that matches the given state description. This agent will work only if the correct decision can be made on the basis of only the current percept. i.e. only if the environment is fully observable.

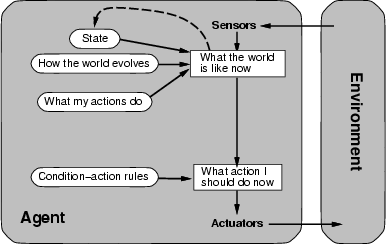
* + 1. **MODEL-BASED REFLEX AGENTS**

To handle partial observability, the agent should maintain some sort of internal state that depends on the percept history and thereby reflects at least some of the unobserved aspects of the current state.

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

* + - * How the world evolves independently of the agent
      * How the agent’s actions affect the world.

This knowledge can be implemented in simple Boolean circuits called model of the world. An agent that uses such a model is called a model-based agent.

The following figure shows the structure of the reflex agent with internal state, showing how the current percept is combined with the old internal state to generate the updated description of the current state.

# The agent program is shown below:

Function REFLEX-AGENT-WITH-STATE(percept)returns an action Static: state, a description of the current world state

Rules, a set of condition-action rules Action, the most recent action, initially none

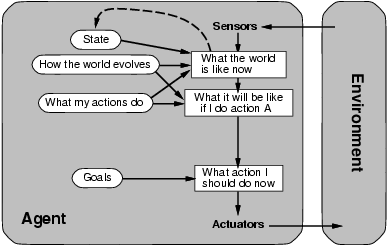
State <- UPDATE-STATE (state, action , percept) Rule<- RULE-MATCH (state, rules)

Action<- RULE-ACTION [rule] Return action

UPDATE-STATE: for creating the new internal state description.

* + 1. **GOAL-BASED AGENTS**

Here, along with current-state description, the agent needs some sort of goal information that describes situations that are desirable – for eg, being at the passenger’s destination.

Goal –based agents structure is shown below:

**A model-based, goal-based agent.**

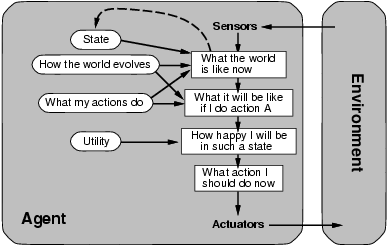
Knowing about the current state of the environment is not always enough to decide what to do. **For example,** at a road junction, the taxi can turn left, turn right, or go straight on. The correct decision depends on where the taxi is trying to get to.

In other words, as well as a current state description, 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 (the same information as was used to update internal state in the reflex agent) in order to choose actions that achieve the goal.

Sometimes **goal-based action selection is straightfor**ward, when goal satisfaction results immediately from a single action. Sometimes it will be more tricky, when the agent has to consider long sequences of twists and turns to find a way to achieve the goal.

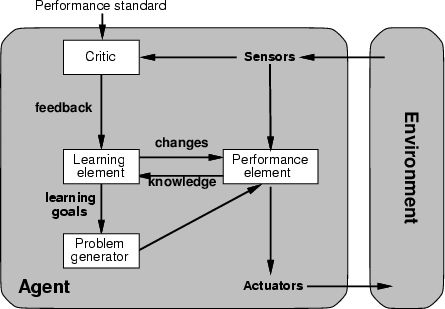
* + 1. **UTILITY-BASED AGENTS**

Goals alone are not enough to generate high-quality behavior in most environments. 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.

A utility function maps a state onto a real number, which describes the associated degree of happiness. The utility-based agent structure appears in the following figure.

***CST71 Artificial Intelligence, Dept of CSE, MVIT***

**LEARNING AGENTS**

It allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow. A learning agent can be divided into four conceptual components, as shown in figure:

**Learning element:** responsible for making improvement.

**Performance element:** responsible for selecting external 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 critic tells the learning element how well the agent is doing with respect to a fixed performance standard. The critic is necessary because the percepts themselves provide no indication of the agent’s success. 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.

# PROBLEM SPACES AND SEARCH Solving problems by searching

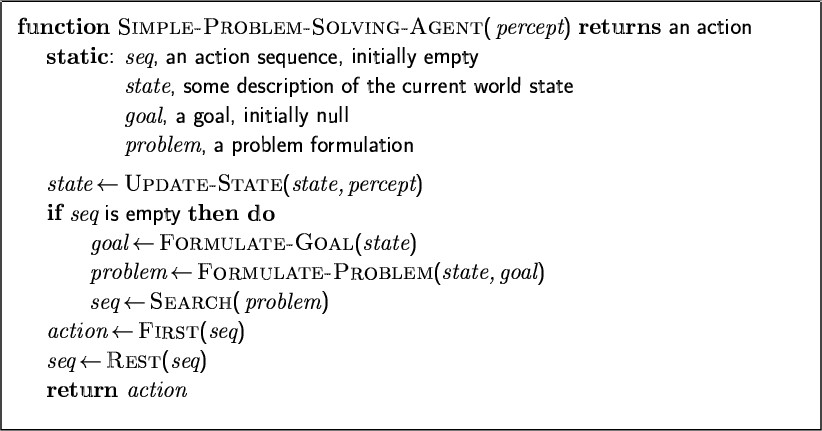
* + *Problem-solving agents*
  + *Problem types*
  + *Problem formulation*
  + *Example problems*
  + *Basic search algorithms*

PROBLEM-SOLVING AGENTS

Problem solving agent is a goal-based agent decides what to do by finding sequences of actions that lead to desirable states. Let us take for an example, an agent in the city of Arad, Romania, enjoying a touring holiday. **Goal formulation**, based on the current situation and the

agent’s performance measure, is the first step in problem solving. We will consider a goal to be a set of world states- exactly those states in which the goal is satisfied.

**Problem formulation** is the process of deciding what actions and states to consider, given a goal. Let us assume that the agent will consider actions at the level of driving from one major town to another. Our agent has now adopted the goal of driving to Bucharest, and is considering where to go from Arad. There are three roads out of Arad. The agent will not know which of its possible actions is best, because it does not know enough about the state that results from taking each action. If the agent has a map, it provides the agent with information about the states it might get itself into, and the actions it can take.

An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence. The process of looking for such a sequence is called a **search**. A search algorithm takes a problem as input and returns a **solution** in the form of an action sequence. Once a solution is found, the actions it recommends can be carried out.

This is called the **execution phase**. The design for such an agent is shown in the following function:

Example: Romania

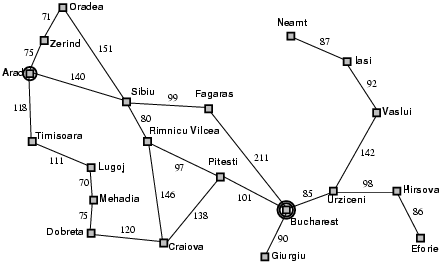
* *On holiday in Romania; currently in Arad.*
* *Flight leaves tomorrow from Bucharest*
* *Formulate goal: be in Bucharest*
* *Formulate problem: states: various cities*

actions: drive between cities

* *Find solution: sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest*

Single-state problem formulation

***A problem is defined by four items:***

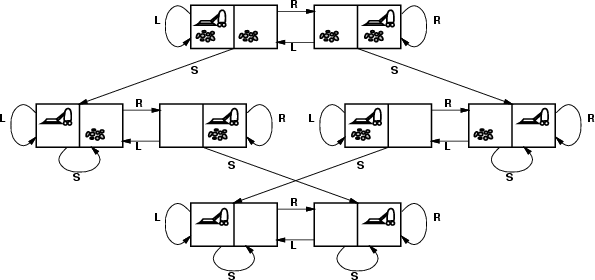
* ***initial state e.g., "at Arad"***
* ***actions or successor function* S(x) *= set of action–state pairs***
  + **e.g., *S(Arad) =* {*<Arad***  ***Zerind, Zerind>, …* }**
* *goal test, can be*
  + explicit, e.g., *x* = "at Bucharest"
  + implicit, e.g., *Checkmate(x)*
* *path cost (additive)*
  + e.g., sum of distances, number of actions executed, etc.
  + *c(x,a,y)* is the step cost, assumed to be ¡Ã 0
* *A solution is a sequence of actions leading from the initial state to a goal state*

# Selecting a state space

* *Real world is absurdly complex*

 state space must be abstracted for problem solving

* *(Abstract) state = set of real states*
* *(Abstract) action = complex combination of real actions*
  + e.g., "Arad  Zerind" represents a complex set of possible routes, detours, rest stops, etc.
* *For guaranteed realizability, any real state "in Arad“ must get to some real state "in Zerind"*
* *(Abstract) solution =*
  + set of real paths that are solutions in the real world
* *Each abstract action should be "easier" than the original problem*

Vacuum world state space graph

## states?

* + ***actions?***

## goal test?

* + ***path cost?***

# Vacuum world state space graph

## states? vacuum2-paths

***integer dirt and robot location***

* ***actions?* Left*,* Right*,* Suck**

## goal test? no dirt at all locations

* ***path cost? 1 per action***

# 8puzzleExample: The 8-puzzle

## states? locations of tiles

* + ***actions? move blank left, right, up, down***

## goal test? = goal state (given)

* + ***path cost? 1 per move***

# stanford-arm+blocksExample: robotic assembly



## states?: real-valued coordinates of robot joint angles parts of the object to be assembled

* + ***a****ctions?: continuous motions of robot joints*
  + *goal test?: complete assembly*
  + *path cost?: time to execute*

# SEARCHING FOR SOLUTIONS:

Solving the formulated problem can be done by a search through the state space. One of the search technique is an explicit **search tree** that is generated by the initial state and the successor function that together define the state space.

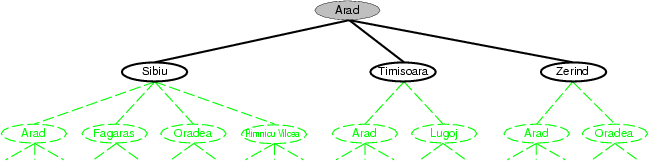
# Tree search algorithms

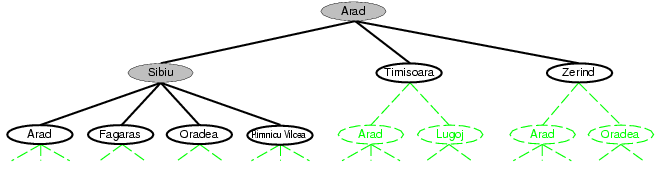
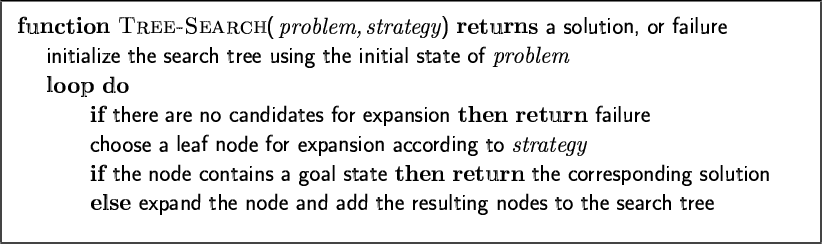
*Basic idea*

– offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

The following figure shows some of the expansions in the search tree for finding a route from Arad to Bucharest. The root of the search tree is a search node corresponding to the initial state, Arad. The first step is to test whether this is a goal state. If this is not the goal state, expand the current state by applying the successor function to the current state, thereby generating a new set of states.

# search-map1cTree search example

The choice of which state to expand is determined by the search strategy. The general tree- search algorithm is given below:



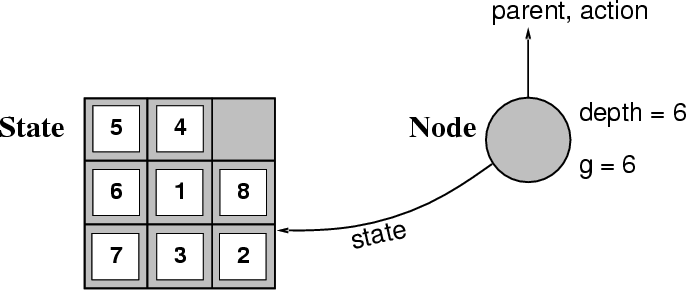
Assume that a node is a data structure with five components:

* STATE: the 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, traditionally denoted by g(n), of the path from the initial state to the node, as indicated by the parent pointers; and
* DEPTH: the number of steps along the path from the initial state.

# Implementation: states vs. nodes

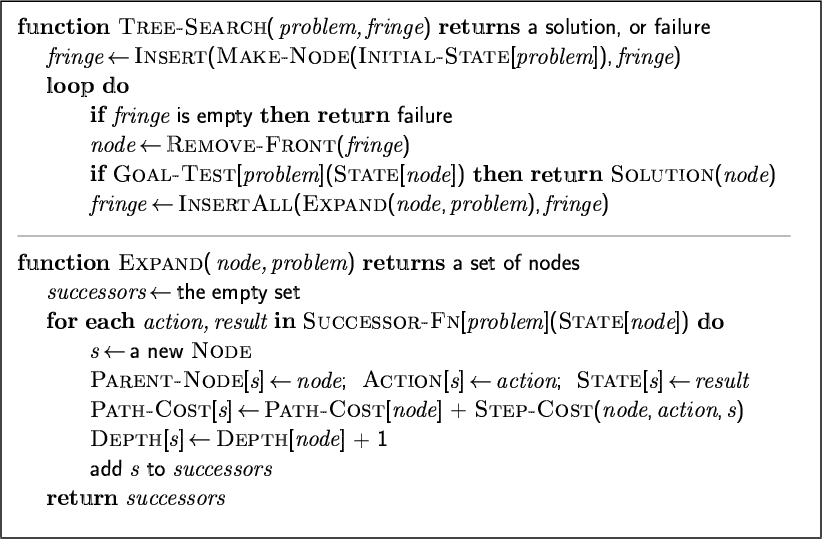
* *A state is a (representation of) a physical configuration*
* *A node is a data structure constituting part of a search tree includes state, parent node, action, path cost* g(x)*, depth*

The node data structure is depicted in the following figure:



The collection of nodes is implemented as a queue. The operations 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 the first element of the queue
* REMOVE-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.

With these definitions, the more formal version of the general tree search algorithm is shown below:

* *The Expand function creates new nodes, filling in the various fields and using the SuccessorFn of the problem to create the corresponding states.*

# Measuring problem-solving performance

* *A search strategy is defined by picking the order of node expansion*
* *Strategies are evaluated along the following dimensions:*
  + completeness: does it always find a solution if one exists?
  + time complexity: number of nodes generated
  + space complexity: maximum number of nodes in memory
  + optimality: does it always find a least-cost solution?

## Time and space complexity are measured in terms of

* + *b:* maximum branching factor of the search tree
  + *d:* depth of the least-cost solution
  + *m*: maximum depth of the state space

# UNINFORMED SEARCH STRATEGIES

* Uninformed or blind search strategies use only the information available 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.
  + 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 the successors of the root node are expanded next, then their succesors, and so on. All the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded. **Algorithm:**

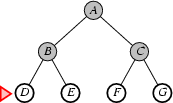
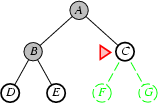
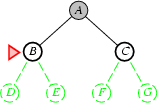
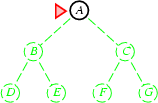
1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element from the queue and place all the children at the end of the queue in any order.
5. Return to step 2.

## Implementation:

* + By calling TREE-SEARCH with an empty fringe

# *fringe* is a FIFO queue, i.e., new successors go at end

The following figure shows the progress of the search on a simple binary tree.



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

Properties of breadth-first search

## Complete? Yes (if b is finite)

* If the shallowest goal node is at some finite depth d, BFS will eventually find it after expanding all shallower nodes (b is a branching factor)

– ***Time?* 1+b+b2+b3*+… +*bd *+* b(bd-1*) = O(bd+1)***

## Space? O(bd+1) (keeps every node in memory)

* We consider a hypothetical state space where 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 b2 at the second level, and so on. Now suppose that the solution is at depth d.

## Optimal? Yes (if cost = 1 per step)

* BFS is optimal if the path cost is a nondecreasing funcion of the depth of the node.
* ***Space is the bigger problem (more than time)***

# Uniform-cost search

BFS is optimal when all step costs are equal, because it always expands the shallowest unexpanded node. Instead of expanding the shallowest node, Uniform-cost search expands the node n with the lowest path cost.

## Implementation:

* *fringe* = queue ordered by path cost
  + *Equivalent to breadth-first if step costs all equal*
* *Complete? Yes, if step cost >=* 
* *Time? # of nodes with* g <= *cost of optimal solution,* O(bceiling(C\*/)) *where* C*\* is the cost of the optimal solution*
* *Space? # of nodes with* g *<= cost of optimal solution,* O(bceiling(C\*/ ))
* *Optimal? Yes – nodes expanded in increasing order of* g(n)

# Depth-first search

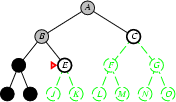
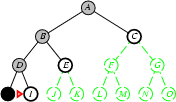
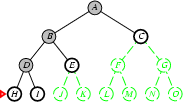
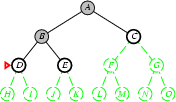
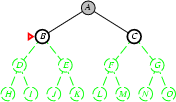
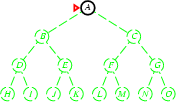
* *Expand deepest unexpanded node*
* *Implementation:*

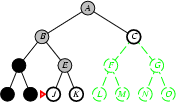
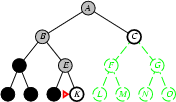
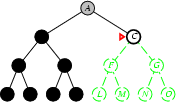
– *fringe* = LIFO queue, i.e., put successors at front

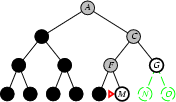
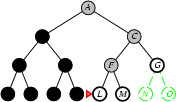
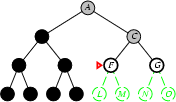
# Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove and expand the first element , and place the children at the front of the queue (in any order).
5. Return to step 2.

The progress of the search is illustrated in the following figure:





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

Properties of depth-first search

* ***Complete?*** *No: fails in infinite-depth spaces, spaces with loops*

– Modify to avoid repeated states along path

 complete in finite spaces

* *Time?* O(bm)*: terrible if* m *is much larger than* d

– but if solutions are dense, may be much faster than breadth-first

* *Space?* O(bm), *i.e., linear space!*
* *Optimal? No*

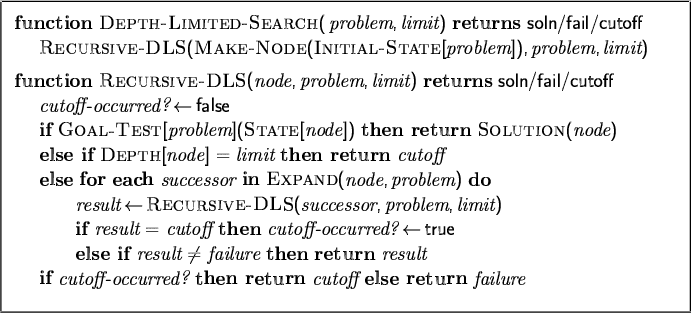
# Depth-limited search

The problem of unbounded trees can be alleviated by supplying DFS with a pre-determined depth limit.

***=*** *depth-first search with depth limit* l*, i.e., nodes at depth* l *have no successors*

Depth-limited search will also be nonoptimal if we choose l<d. Its time complexity is O(bl) and its space complexity is O(bl).

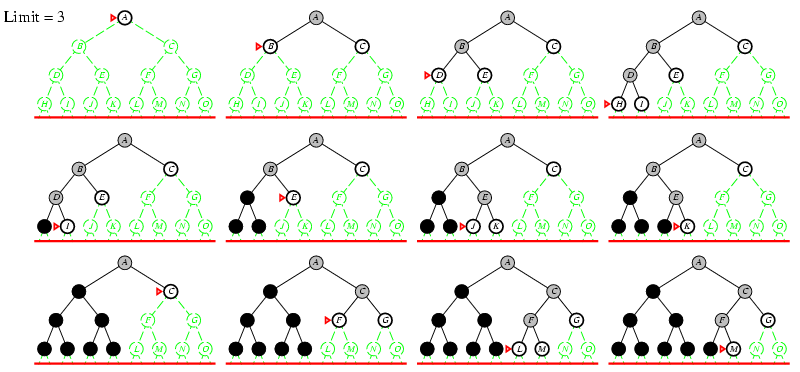
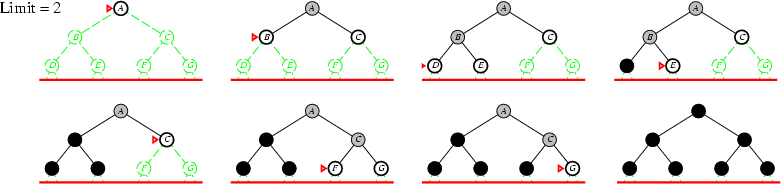
## Recursive implementation:

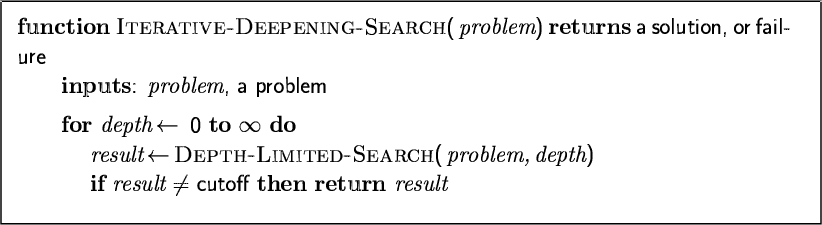


Depth-limited search can terminate with two kinds of failure: the standard failure value indicates no solution; the cutoff value indicates no solution within the depth limit.

# Iterative deepening search

Iterative deepening search is a strategy used in combination with DFS, that finds the best 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 below:



Iterative deepening combines the benefits of DFS and BFS. Like DFS, its memory requirements are very modest: O(bd). Like BFS, 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.

ids-progress1cids-progress2cThe following figure shows four iterations of ITERATIVE-DEEPENING SEARCH on a binary search tree, where the solution is found on the fourth iteration.

***CST71 Artificial Intelligence, Dept of CSE, MVIT***

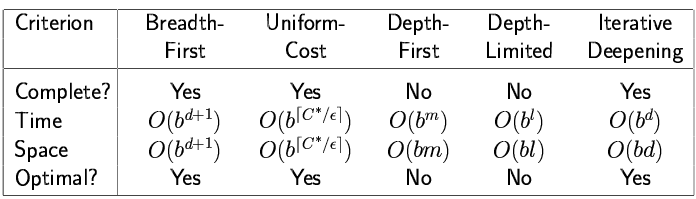
* + *Number of nodes generated in a depth-limited search to depth* d *with branching factor* b*:* NDLS = b0 + b1 + b2 + … + bd-2 + bd-1 + bd
  + *Number of nodes generated in an iterative deepening search to depth* d *with branching factor* b*: NIDS = (d+1)b0 + d b^1 + (d-1)b^2 + … + 3bd-2 +2bd-1 + 1bd*

Properties of iterative deepening search

* *Complete? Yes*

– *Time?* (d+1)b0 + d b1 + (d-1)b2 + … + bd = O(bd)

* *Space?* O(bd)
* *Optimal? Yes, if step cost = 1*

Summary of algorithms

**Bidirectional Search**

The idea behind bi-directional 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.

Bidirectional search is implemented by having one or both of the searches check each node before it is expanded to see if it is in the fringe of the other search tree; if so, a solution has been found. Checking a node for membership in the other search tree can be done in constant time with a hash table, so the time complexity of bi-directional search is O(bd/2).

Atleast one of the search trees must be kept in memory so that the membership check can be done, hence the space complexity is O(bd/2) which is the weakness of the algorithm. The algorithm is complete and optimal if both searches are breadth-first;

# HEURISTIC SEARCH TECHNIQUES - BEST-FIRST SEARCH

INFORMED SEARCH ALGORITHMS

Strategies that know whether one non-goal state is “more promising” than another are called **informed search or heuristic search strategies** Informed search strategy is the one that uses problem-specific knowledge beyond the definition of the problem itself.

* Best-first search
* Greedy best-first search
* A\* search
* Heuristics
* Local search algorithms
* Hill-climbing search
* Simulated annealing search
* Local beam search
* Genetic algorithms

# Review: Tree search

* A search strategy is defined by picking the order of node expansion

# Best-first search

Best first search is an instance of the 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 the lowest evaluation is selected for expansion, because the evaluation measures distance to the goal. It can be implemented using a priority queue, a data structure that will maintain the fringe in ascending order of f – values.

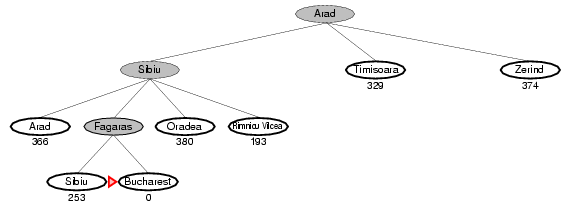
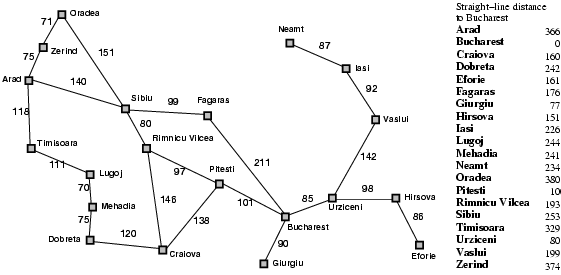
# Algorithm:

1. Place the starting node s on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove the first element from the queue, expand it and compute the estimated goal distances for each child. Place the children on the queue(at either end) and arrange all queue elements in ascending order corresponding to goal distance from the front of the queue.
5. Return to step 2.

Best-first search uses different evaluation functions. A key component of these algorithms is a heuristic function, denoted 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 the straight-line distance from Arad to Bucharest which is shown below:

Romania with step costs in km

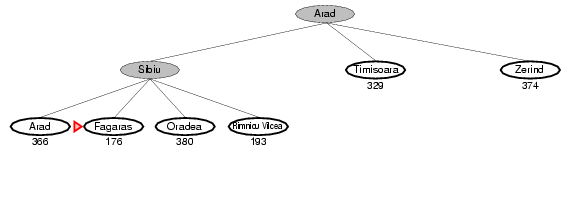
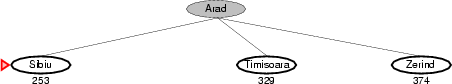
**Greedy best-first search**

* ***Evaluation function* f(n) = h(n) *(heuristic)***

 *= estimate of cost from* n *to* goal

* *e.g.,* hSLD(n) *= straight-line distance from* n *to Bucharest*
* *Greedy best-first search expands the node that appears to be closest to goal*

greedy-progress01cThe progress of a greedy best-first search using hSLD to find a path from Arad to Bucharest is shown in the following figure:



***CST71 Artificial Intelligence, Dept of CSE, MVIT***

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. Greedy best-first search using hSLD finds a solution without ever expanding a node that is not on the solution path; hence its search cost is minimal.

Properties of greedy best-first search

* *Complete? No – can get stuck in loops, e.g., Iasi*  *Neamt*  *Iasi*  *Neamt* 
* *Time?* O(bm)*, but a good heuristic can give dramatic improvement*
* *Space?* O(bm) *-- keeps all nodes in memory*
* *Optimal? No*

# A\* search: Minimizing the total estimated solution cost

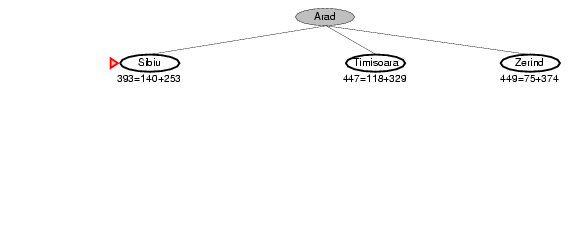
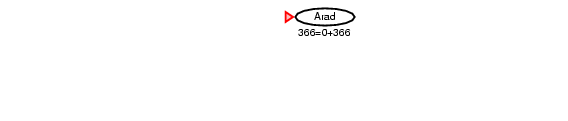
* *Idea: avoid expanding paths that are already expensive*
* *Evaluation function* f(n) = g(n) + h(n)
* g(n) *= cost so far to reach* n
* h(n) *= estimated cost from* n *to goal*
* f(n) *= estimated total cost of path through* n *to goal*

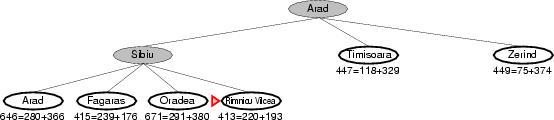
# Algorithm:

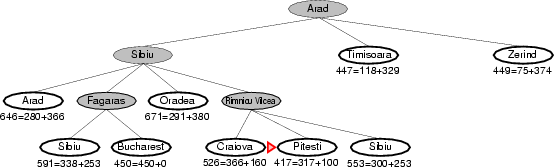
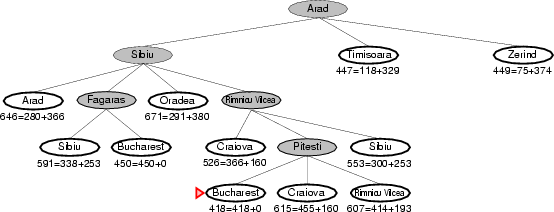
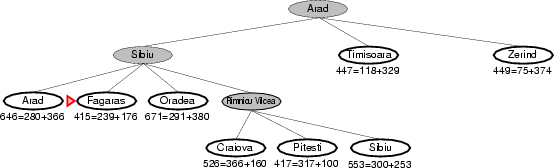
1. Place the starting node s on open.
2. If open is empty, stop and return failure.
3. Remove from open the node n that has the smallest value of f\*(n). If the node is a goal node, return success and stop. Otherwise,
4. Expand n, generating all of its successors n’ and place n on closed. For every successor n’, if n’ is not already on open or closed attach a back-pointer to n, compute f\*(n’) and place it on open.
5. Each n’ that is already on open or closed should be attached to back-pointers which reflect the lowest g\*( n’) path. If n’ was on closed and its pointer was changed, remove it and place it on open.
6. Return to step 2.

The following figure shows an A\* tree search for Bucharest.

***CST71 Artificial Intelligence, Dept of CSE, MVIT***





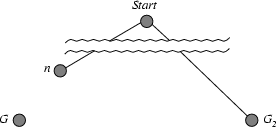


Admissible heuristics

* A heuristic h(n) is admissible if for every node n,

***CST71 Artificial Intelligence, Dept of CSE, MVIT***

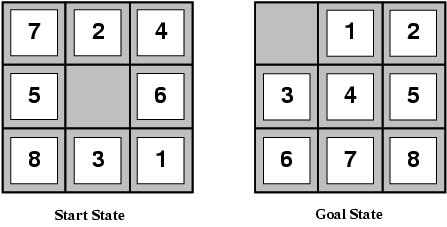
h(n) <= h\*(n), where h\*(n) is the true cost to reach the goal state from n.

* An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic
* Example: hSLD(n) (never overestimates the actual road distance)
* Theorem: If h(n) is admissible, A\* using TREE-SEARCH is optimal Optimality of A\* (proof)
  + Suppose some suboptimal goal G2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.
  + f(G2) = g(G2) since h(G2) = 0
  + g(G2) > g(G) since G2 is suboptimal
  + f(G) = g(G) since h(G) = 0
  + f(G2) > f(G) from above
    - Suppose some suboptimal goal G2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

Properties of A\*

* Complete? Yes (unless there are infinitely many nodes with f <= f(G) )
* Time? Exponential
* Space? Keeps all nodes in memory
* Optimal? Yes **HEURISTIC FUNCTIONS** E.g., for the 8-puzzle:
* h1(n) = number of misplaced tiles
* h2(n) = the sum of the distances of the tiles from their goal positions. This is sometimes called the city block distance or Manhattan distance

(i.e., no. of squares from desired location of each tile)

***CST71 Artificial Intelligence, Dept of CSE, MVIT***

* + h1(S) = ? 8, 8 tiles are out of position, so the start state would have h1=8. h1 is an admissible heuristic, because it is clear that any tile that is out of place must be moved at least once.
* h2(S) = ? 3+1+2+2+2+3+3+2 = 18 . h2 is also admissible, because all any move can do is move one tile one step closer to the goal.

Relaxed problems

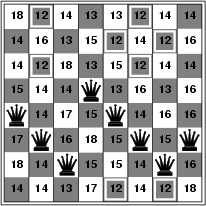
* A problem with fewer restrictions on the actions is called a relaxed problem
* The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
* If the rules of the 8-puzzle are relaxed so that a tile can move anywhere, then h1(n) gives the shortest solution
* If the rules are relaxed so that a tile can move to any adjacent square, then h2(n) gives the shortest solution

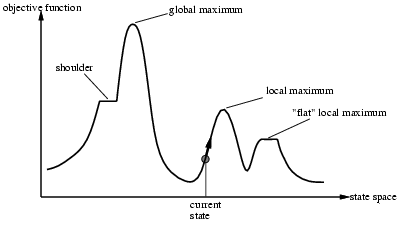
# Local search algorithms

* In many optimization problems, the path to the goal is irrelevant; the goal state itself is the solution
* State space = set of "complete" configurations
* Find configuration satisfying constraints, e.g., n-queens
* In such cases, we can use local search algorithms
* keep a single "current" state, try to improve it Example: n-queens
* Put n queens on an n × n board with no two queens on the same row, column, or diagonal



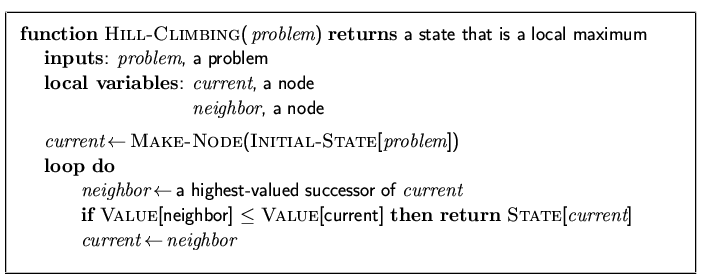
To understand local search, we will find it very useful to consider the state space landscape as shown in the following figure:



A landscape has both “location” and “elevation”. If elevation corresponds to an objective function, then the aim is to find the highest peak - a global maximum. A complete local search algorithm always finds a goal if one exists; an optimal algorithm always finds a global minimum/maximum.

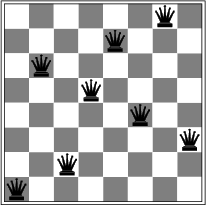
**HILL-CLIMBING SEARCH**

## – "Like climbing Everest in thick fog with amnesia"

The hill-climbing search algorithm is shown in the following function. It 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.

***Problem: depending on initial state, can get stuck in local maxima***

# Hill-climbing search: 8-queens problem

* + *h = number of pairs of queens that are attacking each other, either directly or indirectly*
* *h = 17 for the above state*

A local minimum in the 8-queens state space; the state has h=1 but every successor has a higher cost.

Hill climbing is sometimes called greedy local search because it grabs a good neighbour state without thinking ahead about where to go next. Hill climbing often gets stuck for the following reasons:

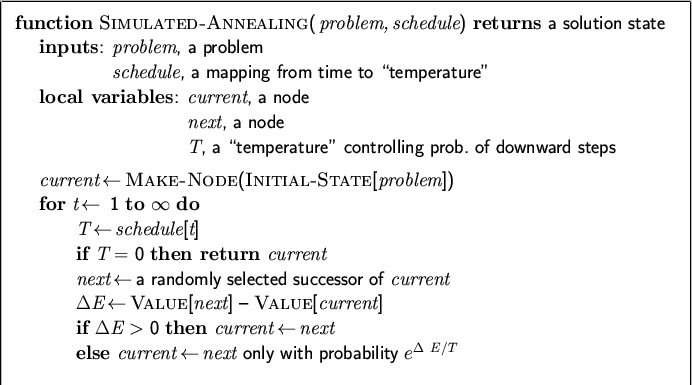
* + **Local Maxima**: a local maximum is a peak that is higher than each of its neighboring states, but lower than the global maximum.
  + **Ridges**: Ridges result in a sequence of local maxima that is very difficult for greedy algorithms to navigate.
  + **Plateaux**: a plateau is an area of the state space landscape where the evaluation funcion 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.

# Simulated annealing search

## Idea: escape local maxima by allowing some "bad" moves but gradually decrease their frequency

A hill climbing algorithm that never makes “downhill” moves towards states with lower value is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk – that is, moving to a successor chosen uniformly at random from the set of successors – is complete, but extremely inefficient.Simulated annealing is the combination of hill climbing with a random walk.

The innermost loop of the simulated-annealing algorithm shown below is quite similar to hill climbing.



# Instead of picking the best move, however, it picks a random move.

Properties of simulated annealing search

* One can prove: If T decreases slowly enough, then simulated annealing search will find a global optimum with probability approaching 1
* Widely used in VLSI layout, airline scheduling, etc

1. **PROBLEM REDUCTION WITH AO\* ALGORITHM.**

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND are may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A\* algorithm can not search AND - OR graphs efficiently. This can be understand from the give figure.

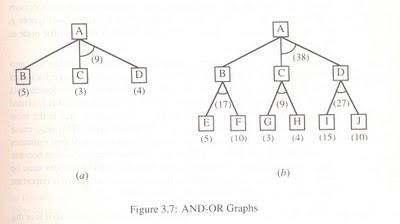


FIGURE : AND - OR graph

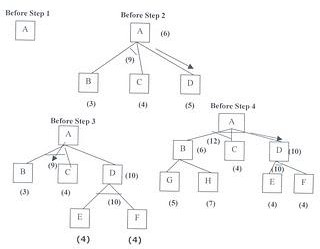
In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D . the numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components.

With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.
3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A\* algorithm. This is because in AO\* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO\* algorithm is illustrated in figure as follows:



**Referring the figure**. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better . it is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A\* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A\* algorithm which used two lists OPEN and CLOSED, the AO\* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes.

The cost of getting from the start nodes to the current node "g" is not stored as in the A\* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO\* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.

For representing above graphs AO\* algorithm is as follows

# AO\* ALGORITHM:

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).
2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.
3. Trace the marked arcs from INIT and select an unbounded node NODE.
4. Generate the successors of NODE . if there are no successors then assign FUTILITY as h' (NODE). This means that NODE is not solvable. If there are successors then for each

one

called SUCCESSOR, that is not also an ancester of NODE do the following

1. add SUCCESSOR to graph G
2. if successor is not a terminal node, mark it solved and assign zero to its h ' value.
3. If successor is not a terminal node, compute it h' value.
4. propagate the newly discovered information up the graph by doing the following . let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;
   1. select a node from S call if CURRENT and remove it from S.
   2. compute h' of each of the arcs emerging from CURRENT , Assign minimum h' to CURRENT.
   3. Mark the minimum cost path a s the best out of CURRENT.
   4. Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED.
   5. If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph . hence all the ancestors of CURRENT are added to S.

# CONSTRAINT SATISFACTION

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}.

|  |
| --- |
|  |
| **Figure:** 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

1. **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(dn) – 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.

# 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 :

1. **unary constraints** involve a single variable.

Example : SA # green

1. Binary constraints involve paris of variables. Example : SA # WA
2. Higher order constraints involve 3 or more variables. Example : cryptarithmetic puzzles.

*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.

|  |
| --- |
|  |
| ***Figure 2.17 A simple backtracking algorithm for constraint satisfaction problem. The***  ***algorithm is modeled on the recursive depth-first search*** |

# MEANS ENDS ANALYSIS

Most of the search strategies either reason forward of backward however, often a mixture o the two directions is appropriate. Such mixed strategy would make it possible to solve the major parts of problem first and solve the smaller problems the arise when combining them together. Such a technique is called "Means - Ends Analysis".

The means -ends analysis process centers around finding the difference between current state and goal state. The problem space of means - ends analysis has an initial state and one or more goal state, a set of operate with a set of preconditions their application and difference functions that computes the difference between two state a(i) and s(j). A problem is solved using means - ends analysis by

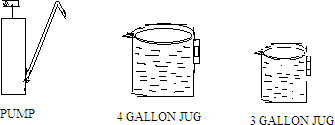
1. Computing the current state s1 to a goal state s2 and computing their difference D12.
2. Satisfy the preconditions for some recommended operator op is selected, then to reduce the difference D12.
3. The operator OP is applied if possible. If not the current state is solved a goal is created and means- ends analysis is applied recursively to reduce the sub goal.
4. If the sub goal is solved state is restored and work resumed on the original problem.

means- ends analysis I useful for many human planning activities. Consider the example of planing for an office worker. Suppose we have a different table of three rules:

1. If in out current state we are hungry , and in our goal state we are not hungry , then either the "visit hotel" or "visit Canteen " operator is recommended.
2. If our current state we do not have money , and if in your goal state we have money, then the "Visit our bank" operator or the "Visit secretary" operator is recommended.
3. If our current state we do not know where something is , need in our goal state we do know, then either the "visit office enquiry" , "visit secretary" or "visit co worker " operator is recommended.

# OTHER IMPORTANT EXAMPLE PROBLEMS IN UNIT:1

* 1. **Water jug problem:**



# Problem statement:

You are given two jugs, 4 gallon one and 3 gallon one. Neither has nay measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water on 4 gallon jug?

# Solution:

**Defining and analyzing the problem:**

* 1 galloon = 3.78541 litre
* The state space for the problem can be described as set of ordered pair of integer (x.y) Such that x= 0,1,2,3 or 4.

y= 0,1,2 or 3.

x= no of gallons of water in the 4 gallon jug.

y= no of gallons of water in the 3 gallon jug.

* The initial state = (0,0) and the Goal state = (2,n) for any value of n (where value of n). how many gallons need to be in 3 gallons jug is not specified in problem.

# Assumptions:

* We need to take some assumptions not mentioned in the problem statement.
* We can fill a jug from pump that we can pour water out of a jug onto the ground.
* We can pour water from one jug to another. Since there are no measuring devices are available.
* To solve the water jug problem all we need is addition to the problem description given above is a control structure that loops through a simple cycle in which some rule whose left side matches the current state is described on right side.

TABLE: WATER JUG PROBLEM

|  |  |  |  |
| --- | --- | --- | --- |
| **Steps** | **Description** | **4 Gallon Jug** | **3 Gallon jug** |
| Step-1 | Initially empty | 0 | 0 |
| Step-2 | pour water on 3 gallon jug. | 0 | 3 |
| Step-3 | pour water from  jug 3 to jug 4 | 3 | 0 (empty jug) |
| Step-4 | pump water to 3  gallon jug | 3 (remaining 1  gallon ) | 3 (full) |
| Step-5 | Pour water from  3 gallon to 4 gallon jug. | 4 (full) | 2 |
| Step-6 | empty 4 gallon  water on ground | 0 | 2 |
| Step-7 | Pour water from  3 gallon jug to 4 gallon jug. | 2 | 0 |

# Production rules for water jug problem.

1. (x,y)  (4,y) fill the 4 gallon jug. Where x<4
2. (x,y)  (x,3) fill the 3 gallon jug. Where y<3
3. (x,y)  (x-d,y) pour some water out of 4 gallon jug. If x>0
4. (x,y)  (x,y-d) pour some water out from 3 gallon jug. (y>0)
5. (x,y)  (0,y) empty the 4 gallon jug on the ground If (x>0)
6. (x,y)  (x,0) empty the 3 gallon jug on the ground If (y>0
7. (x,y)  (4,y-(4-x)) pour water from the 3 gallon jug into 4 if x+y ≥4 and **y>0** gallon jug until 4 gallon jug is full.
8. (x,y)  (x-(3-y),3)) pour water from the 4 gallon jug into 3 if x+y ≥3 and x>0 gallon jug until 3 gallon jug is full.
9. (x,y)  (x+y,0)pour all the water from 3 gallon into 4 gallon jug if x+y ≤ 4 and y>0
10. (x,y)  (0,x+y)pour all the water from 4 gallon into 3 gallon jug if x+y ≤ 3 and x>0
11. (0,2)  (2,0) pour the 2 gallons from the 3 gallons jug into 4

gallon jug.

1. (2,y)  (0,y) empty the 2 gallons in the 4 gallons jug on the

ground.

# TRAVELLING SALESMAN PROBLEM

A salesman has a list of cities, each of which w must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

# Solution:

* Let G= (V,E) be a directed graph with each edge cost cij.
* Cij is defined such that cij.>0 for all I and j and cij = ∞ if (i,j) not belongs to E.
* A tour of G is a directed simple cycle that includes every vertex in V.
* The cost of a tour is the sum of the cost of edges on the tour. The TVs problem is to find a tour of minimum cost.

**For example:** consider a production environment in wchich several commodities are manufactured by same set of machines. The manufacture proceeds in cycles. In each production cycle, n different commodities are produced. When the machines are changed from production of commodity I to commodity j, a change over cost cij is incurred.

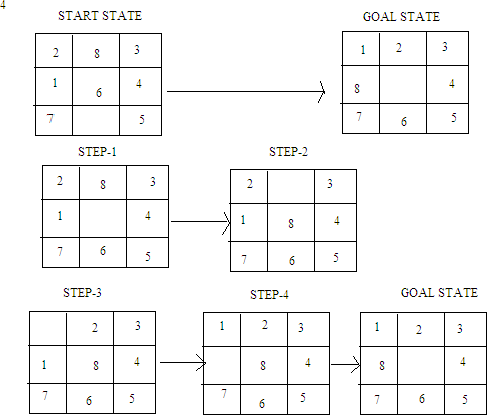
# PUZZLE PROBLEM:

The 8 puzzle is a square tray in which are placed, eight square tiles. The remaining 9th square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can be slid into that space. A game consists of a starting position and a specified goal position.

Solution:

Initial state

Step 1: Move 6th Tile to the empty space. Step 2: Move 8th Tile to the empty space. Step 3: Move 2th Tile to the empty space. Step 4: Move 1th Tile to the empty space. Step 5: Move 8th Tile to the empty space. Step 6: Goal state is reached.

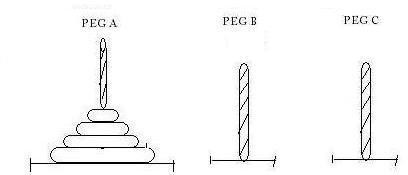


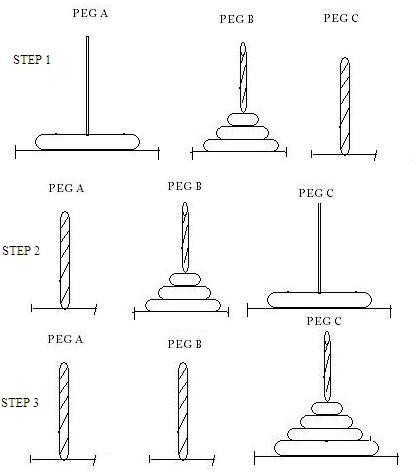
# THE TOWER OF HANOI PROBLEM

The tower of Hanoi is one of example for recursion technique.

1. Three pegs A,B,C are exist. Five disks of differing diameter are placed on peg A so that a larger disk always below a smaller disk.
2. The object is to move the five disk to peg C, using peg B as auxiliary. Only the top disk on any peg moved to any other peg, and a larger disk may never rest on a smaller one.

# PROBLEM:



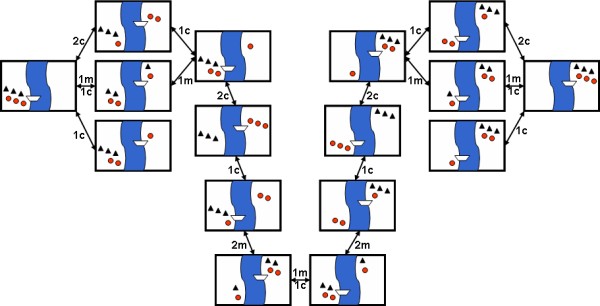


STEP 1: Moving A to B letting larger disk on A itself. STEP 2: Move disk A to C.

STEP 3: Move disk from B to C we can reach GOAL STATE.

# THE MISSIONARIES AND CANNIBALS PROBLEM:

**Problem:** Three missionaries and three cannibals find themselves on one side of a river. They have agreed that they would all like to get to the other side. But the missionaries are not sure what else the cannibals have agreed to. So the missionaries want to manage the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals, who are on the same side. The only boat available holds only two people at a time. How can everyone get across the river without the missionaries risking being eaten?



# 8 QUEEN’S PROBLEM:

**A** classic combinatorial problem is to place eight queens on a 8x8 chess board so that no two attack, that is no two of them are on the same row, column, or diagonal.

Solution: To model this problem

Assume that each queen ia in different column;

Assign a variable Ri (i=1 to N) to the queen in the ith column indicating the position of queen in the row.

Apply “no-threatening” constraints between each couple Ri and Rj of the queens and evaluate the algorithm.

# ssExample:

The 8 queen puzzle has 92 distinct solutions. If the solutions that differ only by symmetry operations (rotations and reflections) of the board are counted as one. The puzzle has 12 unique solutions. Only two solutions are presented above.