ARTIFICIAL INTELLIGENCE

UNIT-I PROBLEM SOLVING

Introduction – Agents – Problem formulation - Uninformed search strategies – Heuristics - Informed search strategies - Constraint satisfaction

What is artificial intelligence?

- **Artificial Intelligence** is the branch of computer science concerned with making computers behave like humans.
- Major AI textbooks define artificial intelligence as "the study and design of intelligent agents," where an **intelligent agent** is a system that **perceives** its **environment** and **takes actions** which maximize its chances of success.
- **John McCarthy**, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines, especially intelligent computer programs."
- The definitions of AI according to some text books are categorized into four approaches and are summarized in the table below :

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

Applications of Artificial Intelligence:

- **Autonomous planning and scheduling:**

A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson *et* al., 2000). Remote Agent generated plans from high-level goals specified from the ground, and it monitored the operation of the spacecraft as the plans were executed-detecting, diagnosing, and recovering from problems as they occurred.

- **Game playing:**

  IBM's Deep Blue became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997).

- **Autonomous control:**

  The ALVINN computer vision system was trained to steer a car to keep it following a lane. It was placed in CMU's NAVLAB computer-controlled minivan and used to navigate across the United States-for 2850 miles it was in control of steering the vehicle 98% of the time.

- **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.

- **Logistics Planning:**

  During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques allowed a plan to be generated in hours that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

- **Robotics:**

  Many surgeons now use robot assistants in microsurgery. HipNav (DiGioia *et* al., 1996) is a system that uses computer vision techniques to create a three-dimensional model of a patient's internal anatomy and then uses robotic control to guide the insertion of a hip replacement prosthesis.

- **Language understanding and problem solving:**

  PROVERB (Littman *et al.*, 1999) is a computer program that solves crossword puzzles better than most humans, using constraints on possible word fillers, a large database of past puzzles, and a variety of information sources including dictionaries and online databases such as a list of movies and the actors that appear in them.

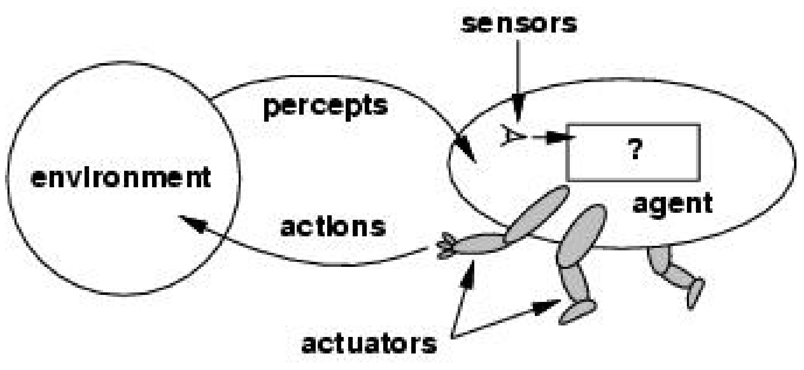
AGENTS:

Rationality concept can be used to develop a smallest of design principle for building successful agents; these systems are reasonably called as Intelligent.

Agents and environments:

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

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



Percept

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

Percept Sequence

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

Agent function

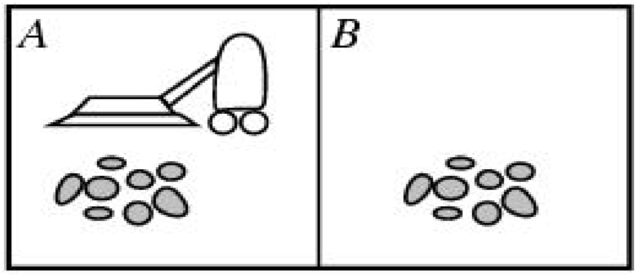
Mathematically speaking, we say that an agent's behavior is described by the **agent function** that maps any given percept sequence to an action.

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

Agent program

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

- To illustrate these ideas, we will use a very simple example-the vacuum-cleaner world shown in Figure.
- This particular world has just two locations: squares A and B.
- The vacuum agent perceives which square it is in and whether there is dirt in the square.
- It can choose to move left, move right, suck up the dirt, or do nothing.
- One very simple agent function is the following:
- if the current square is dirty, then suck, otherwise,
- it move to the other square.
- A partial tabulation of this agent function is shown in Figure.



Agent function

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

Agent program

```
function Reflex-VACUUM-AGENT ([locations, status]) returns an action

    if status = Dirty then return Suck
    else if location = A then return Right
    elseif location = B then return Left
```

Good Behavior: The concept of Rationality

- A **rational agent** is one that does the right thing-conceptually speaking; every entry in the table for the agent function is filled out correctly.
- Obviously, doing the right thing is better than doing the wrong thing.
- The right action is the one that will cause the agent to be most successful.

Performance measures

- **A performance measure** embodies the **criterion for success** of an agent's behavior.
- When an agent is plunked down in an environment, it generates a sequence of actions according to the percepts it receives.
- This sequence of actions causes the environment to go through a sequence of states.
- If the sequence is desirable, then the agent has performed well.

Rationality

What is rational at any given time depends on four things:

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

Omniscience, learning, and autonomy

- An **omniscient agent** knows the actual outcome of its actions and can act accordingly; but omniscience is impossible in reality.
- Doing actions in order to modify future percepts-sometimes called **information gathering**-is an important part of rationality.
- Our definition requires a rational agent not only to gather information, but also to **learn** as much as possible from what it perceives.
- To the extent that an agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks autonomy.
- A rational agent should be **autonomous**-it should learn what it can to compensate for partial or incorrect prior knowledge.

Task environments

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

Specifying the task environment

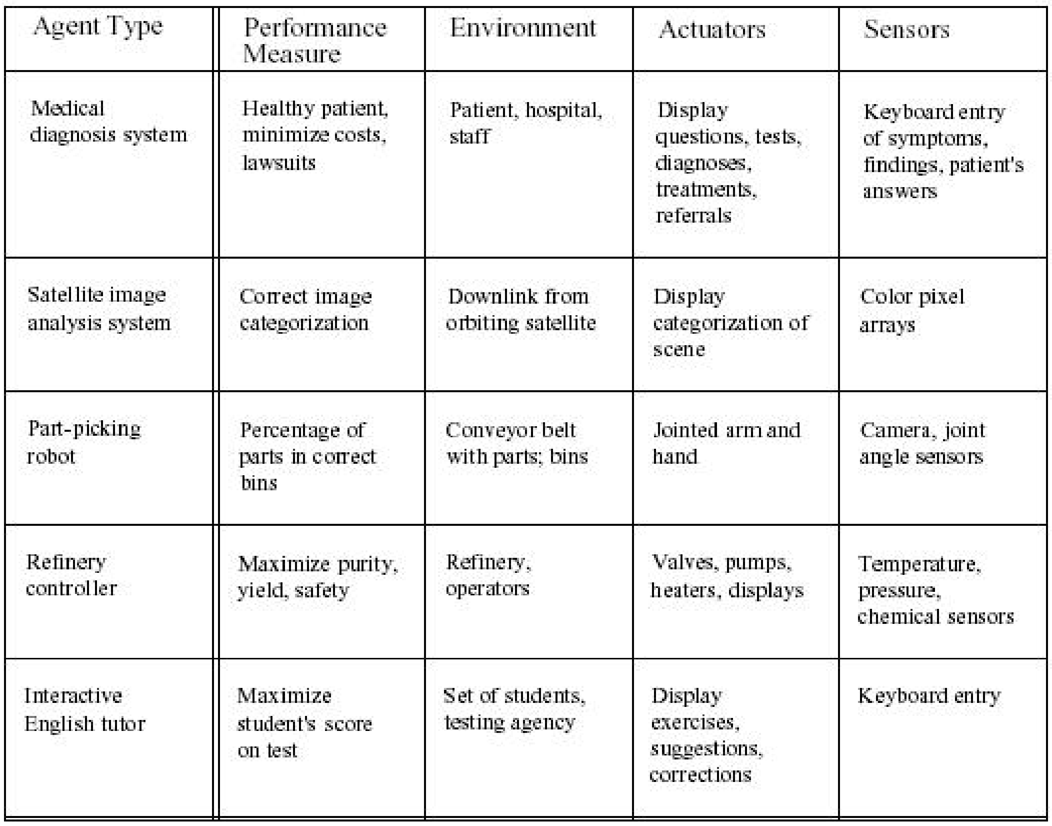
- The rationality of the simple vacuum-cleaner agent, needs specification of
  - ✓ the performance measure
  - ✓ the environment
  - ✓ the agent's actuators
  - ✓ Sensors.

PEAS

- All these are grouped together under the heading of the **task environment.**
- We call this the **PEAS** (Performance, Environment, Actuators, Sensors) description.
- In designing an agent, the first step must always be to specify the task environment as fully as possible.
- The following table shows PEAS description of the task environment for an automated taxi.

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

- The following table shows PEAS description of the task environment for some other agent type.



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

Properties of task environments

- **Fully observable vs. partially observable**
- **Deterministic vs. stochastic**
- **Episodic vs. sequential**
- **Static vs. dynamic**
- **Discrete vs. continuous**
- **Single agent vs. multiagent**

Fully observable vs. **partially observable.**

- If an agent's sensors give it access to the complete state of the environment at each point in time, then we say that the task environment is fully observable.
- A task environment is effectively fully observable if the sensors detect all aspects that are *relevant* to the choice of action;
- An environment might be partially observable because of noisy and inaccurate sensors or because parts of the state are simply missing from the sensor data.

Deterministic vs. **stochastic.**

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

Episodic vs. **sequential**

- In an **episodic task environment**, the agent's experience is divided into atomic episodes.
- Each episode consists of the agent perceiving and then performing a single action. Crucially, the next episode does not depend on the actions taken in previous episodes.
- For example, an agent that has to spot defective parts on an assembly line bases each decision on the current part, regardless of previous decisions;
- In **sequential environments**, on the other hand, the current decision Could affect all future decisions.
- Chess and taxi driving are sequential:

Discrete vs. **continuous.**

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

Single agent vs. **multiagent.**

- An agent solving a crossword puzzle by itself is clearly in a single-agent environment,
- Where as an agent playing chess is in a two-agent environment.
- Multiagent is further classified in to two ways
    - ✓ Competitive multiagent environment
    - ✓ Cooperative multiagent environment

Agent programs

- The job of Artificial Intelligence is to design the agent program that implements the agent function mapping percepts to actions
- The agent program will run in an architecture
- An architecture is a computing device with physical sensors and actuators
- Where Agent is combination of Program and Architecture

Agent = Program + Architecture

- An agent program takes the current percept as input while the agent function takes the entire percept history
- Current percept is taken as input to the agent program because nothing more is available from the environment
- The following TABLE-DRIVEN_AGENT program is invoked for each new percept and returns an action each time

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

static: percepts, a sequence initially empty
table, a table of actions, indexed by percept sequence

append percept to the end of percepts
action ⟵ LOOKUP(percepts, table)
return action
```

Drawbacks:

- **Table lookup** of percept-action pairs defining all possible condition-action rules necessary to interact in an environment
- **Problems**
    - Too big to generate and to store (Chess has about 10^120 states, for example)
    - No knowledge of non-perceptual parts of the current state
    - Not adaptive to changes in the environment; requires entire table to be updated if changes occur
    - Looping: Can't make actions conditional

- Take a long time to build the table
- No autonomy
- Even with learning, need a long time to learn the table entries

Some Agent Types

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

Kinds of Agent Programs

- The following are the agent programs,
    - Simple reflex agents
    - Mode-based reflex agents
    - Goal-based reflex agents
    - Utility-based agents

Simple Reflex Agent

- The simplest kind of agent is the **simple reflex agent.**
- These agents select actions on the basis of the *current* percept, ignoring the rest of the percept history.
- For example, the vacuum agent whose agent function is tabulated is given below,
- a simple reflex agent, because its decision is based only on the current location and on whether that contains dirt.
- Select action on the basis of *only the current* percept.E.g. the vacuum-agent
- Large reduction in possible percept/action situations(next page).
- Implemented through *condition-action rules*
- If dirty then suck

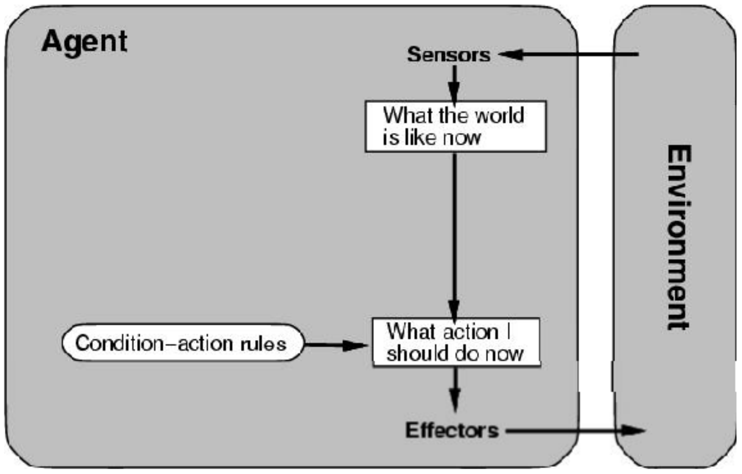
A Simple Reflex Agent: Schema

- Schematic diagram of a simple reflex agent.
- The following simple reflex agents, acts according to a rule whose condition matches the current state, as defined by the percept

function SIMPLE-REFLEX-AGENT(*percept*) **returns** an action
static: *rules*, a set of condition-action rules
state ← INTERPRET-

INPUT(*percept*)

```
rule ← RULE-
MATCH(state, rule)
action ← RULE-
ACTION[rule] return action
```



- The agent program for a simple reflex agent in the two-state vacuum environment.

```
function REFLEX-VACUUM-AGENT ([location, status]) return an action
if status == Dirty then return Suck
else if location == A then return Right
else if location == B then return Left
```

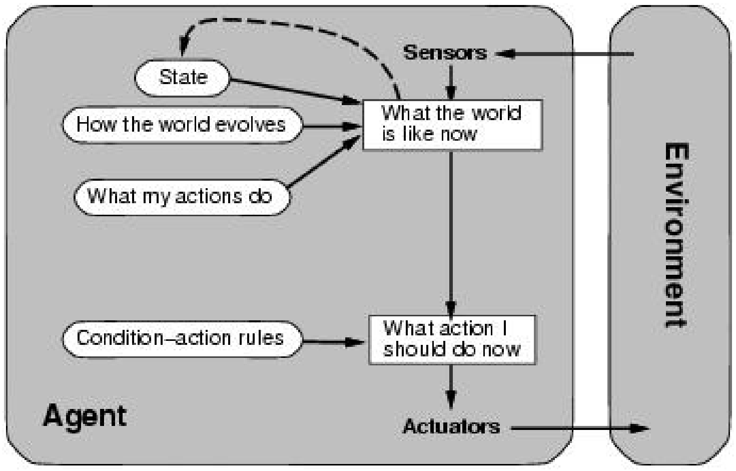
Characteristics

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

Model-based reflex agents

- The most effective way to handle partial observability is for the agent to *keep track of the part of the world it can't see now.*
- That is, 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 as time goes by requires two kinds of knowledge to be encoded in the agent program.
- First, we need some information about how the world evolves independently of the agent
- For example, that an overtaking car generally will be closer behind than it was a moment ago.
- Second, we need some information about how the agent's own actions affect the world

- For example, that when the agent turns the steering wheel clockwise, the car turns to the right or that after driving for five minutes northbound on the freeway one is usually about five miles north of where one was five minutes ago.
- This knowledge about "how the world working - whether implemented in simple Boolean circuits or in complete scientific theories-is called a **model** of the world.
- An agent that uses such a MODEL-BASED model is called a **model-based agent.**
- Schematic diagram of A model based reflex agent

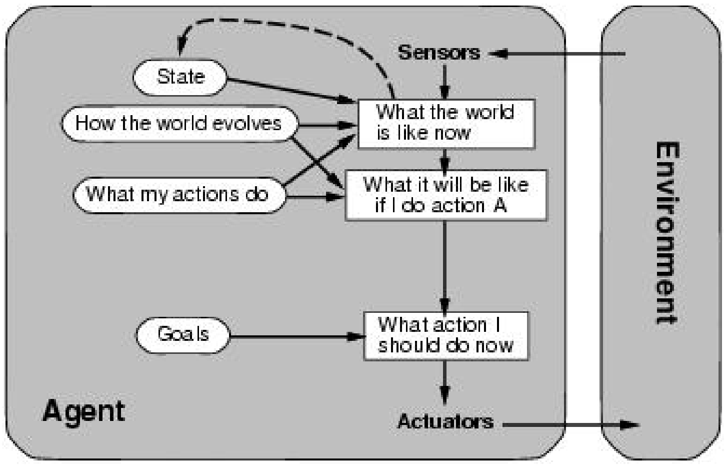


- Model based reflex agent. It keeps track of the current state of the world using an internal model. It then chooses an action in the same way as the reflex agent.

```
function REFLEX-AGENT-WITH-STATE(percept) returns an action
static: rules, a set of condition-action rules
state, a description of the current world state
action, the most recent action.
state ⟵ VPDATE-STATE(state, action, percept)
rule ⟵ RVLE-
MATCH(state, rule)
action ⟵ RVLE-
ACTION[rule] return action
```

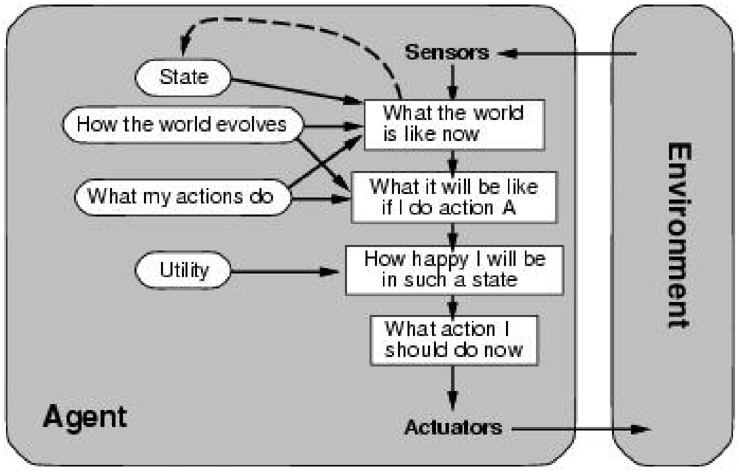
Goal-based agents

- 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.
- Schematic diagram of the goal-based agent's structure.


Sensors
State
How the world evolves
What my actions do
What the world is like now
What it will be like if I do action A
Goals
What action I should do now
Agent
Actuators
Environment

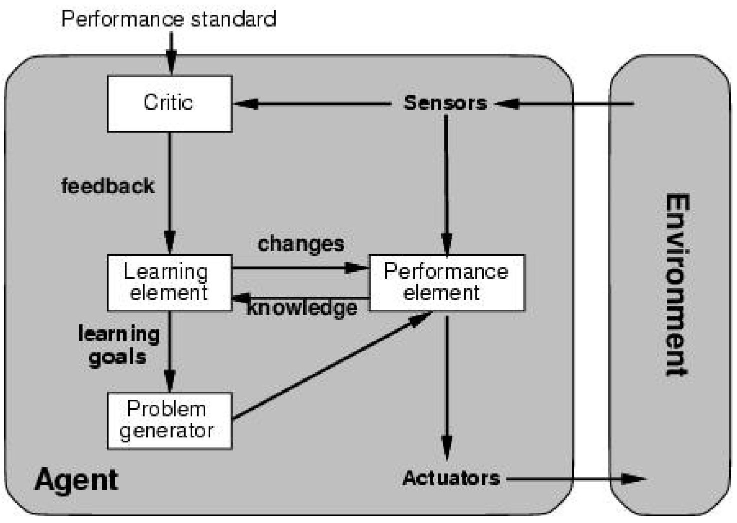
Utility-based agents

- Goals alone are not really enough to generate high-quality behavior in most environments.
- For example, there are many action sequences that will get the taxi to its destination (thereby achieving the goal) but some are quicker, safer, more reliable, or cheaper than others.
- Goals just provide a crude binary distinction between "happy" and "unhappy" states, whereas a more general **performance measure** should allow a comparison of different world states according to exactly how happy they would make the agent if they could be achieved. Because "happy" does not sound very scientific, the customary terminology is to say that if one world state is preferred to another, then it has higher **utility** for the agent.
- Schematic diagram of a utility-based agents
- It uses a model of the world, along with a utility function that measures its preferences among states of the world.
- Then it chooses the action that leads to the best expected utility, where expected utility is computed by averaging over all possible outcome states, weighted by the probability of the outcome.
- Certain goals can be reached in different ways
  - Some are better, have a higher utility
- Vtility function maps a (Sequence of) state(S) onto a real number.
- Improves on goal:
  - Selecting between conflicting goals
  - Select appropriately between several goals based on likelihood of Success


Sensors
State
How the world evolves
What my actions do
Utility
What the world is like now
What it will be like if I do action A
How happy I will be in such a state
What action I should do now
Agent
Actuators
Environment

Learning Agent

Schematic diagram of Learning Agent



- All agents can improve their performance through learning.
- A learning agent can be divided into four conceptual components, as,
  - Learning element
  - Performance element
  - Critic
  - Problem generator
- The most important distinction is between the **learning element**, which is responsible for making improvements,
- The **performance element**, which is responsible for selecting external actions.
- The performance element is what we have previously considered to be the entire agent: it takes in percepts and decides on actions.
- The learning element uses feedback from the **critic** on how the agent is doing and determines how the performance element should be modified to do better in the future.
- The last component of the learning agent is the **problem generator**.

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

Summary: Intelligent Agents

- An **agent** perceives and acts in an environment, has an architecture, and is implemented by an agent program.
- Task environment **– PEAS (P**erformance**, E**nvironment, **A**ctuators, **S**ensors**)**
- The most challenging environments are inaccessible, nondeterministic, dynamic, and continuous.
- An **ideal agent** always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An **agent program** maps from percept to action and updates internal state.
  - **Reflex agents** respond immediately to percepts.
    - simple reflex agents
    - model-based reflex agents
  - **Goal-based agents** act in order to achieve their goal(s).
  - **Utility-based agents** maximize their own utility function.
- All agents can improve their performance through **learning.**

Problem Formulation

- An important aspect of intelligence is *goal-based* problem solving.
- The solution of many problems can be described by finding a sequence of actions that lead to a desirable goal.
- Each action changes the *state* and the aim is to find the sequence of actions and states that lead from the initial (start) state to a final (goal) state.
- **A well-defined problem can be described by:**
  - **Initial state**
  - **Operator or successor function -** for any state x returns s(x), the set of states reachable from x with one action
  - **State space -** all states reachable from initial by any sequence of actions
  - **Path -** sequence through state space
  - **Path cost -** function that assigns a cost to a path. Cost of a path is the sum of costs of individual actions along the path
  - **Goal test -** test to determine if at goal state
- What is **Search**?
- Search is the systematic examination of states to find path from the start/root state to the goal state.
- The set of possible states, together with *operators* defining their connectivity constitute the *search space*.
- The output of a search algorithm is a solution, that is, a path from the initial state to a state that satisfies the goal test.

Problem-solving agents

- A Problem solving agent is a goal-based agent.
- It decides 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
- For 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
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

- Goal formulation and 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.

Search

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

- The **search algorithm** takes a **problem** as **input** and returns a **solution** in the form of **action sequence.**
- Once a solution is found,the **execution phase** consists of carrying out the recommended action.
- The following shows a simple "formulate,search,execute" design for the agent.
- Once solution has been executed, the agent will formulate a new goal.
- It first formulates a **goal** and a **problem**,searches for a sequence of actions that would solve a problem,and executes the actions one at a time.

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

- The agent design assumes the Environment is

- **Static**: The entire process carried out without paying attention to changes that might be occurring in the environment.
- **Observable :** The initial state is known and the agent's sensor detects all aspects that are relevant to the choice of action
- **Discrete :** With respect to the state of the environment and percepts and actions so that alternate courses of action can be taken
- **Deterministic:** The next state of the environment is completely determined by the current state and the actions executed by the agent. Solutions to the problem are single sequence of actions

An agent carries out its plan with eye closed. This is called an open loop system because ignoring the percepts breaks the loop between the agent and the environment.

Well-defined problems and solutions

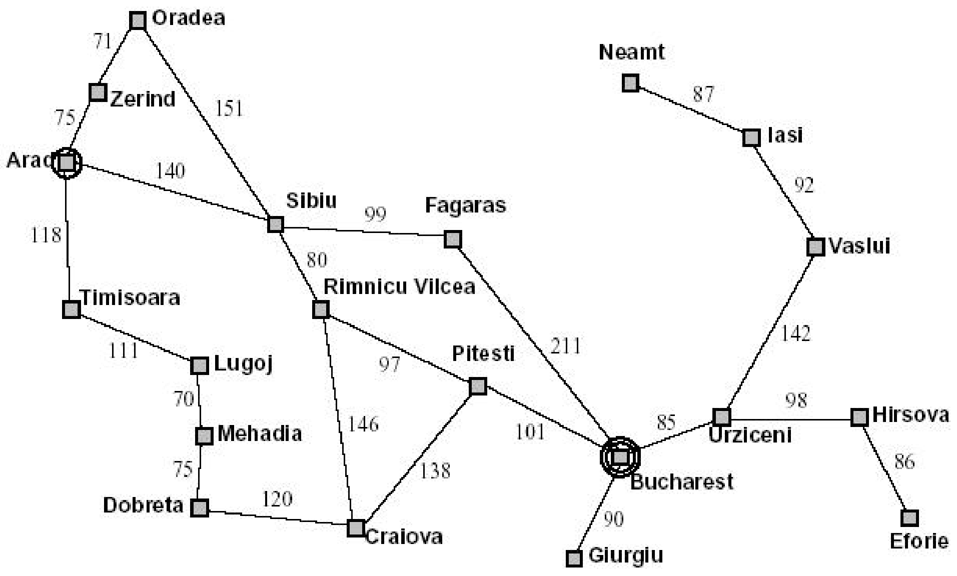
A **problem** can be formally defined by **four components**:

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

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

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

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



A simplified Road Map of part of Romania

Advantages:

- They are easy enough because they can be carried out without further search or planning
- The choice of a good abstraction thus involves removing as much details as possible while retaining validity and ensuring that the abstract actions are easy to carry out.

EXAMPLE PROBLEMS

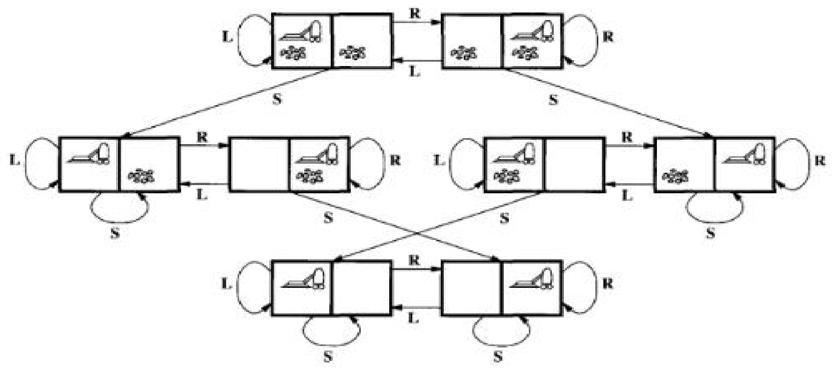
- The problem solving approach has been applied to a vast array of task environments. Some best known problems are summarized below.
- They are distinguished as toy or real-world problems
  - A T**oy problem** is intended to illustrate various problem solving methods. It can be easily used by different researchers to compare the performance of algorithms.
  - A R**eal world problem** is one whose solutions people actually care about.

TOY PROBLEMS

Vacuum World Example

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

Vacuum World State Space

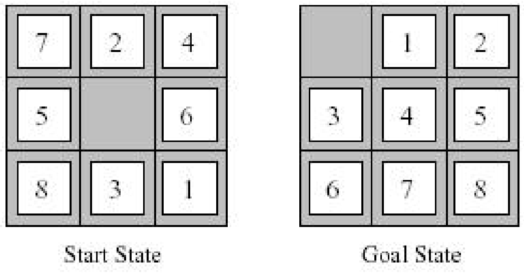


The state space for the vacuum world.
Arcs denote actions: L = Left,R = Right,S = Suck

8-puzzle:

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

Example: The 8-puzzle



A typical instance of 8-puzzle.

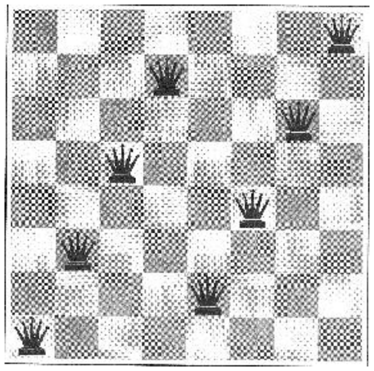
The problem formulation is as follows :

- **States** : A state description specifies the location of each of the eight tiles and the blank in one of the nine squares.
- **Initial state** : Any state can be designated as the initial state. It can be noted that any given goal can be reached from exactly half of the possible initial states.
- **Successor function** : This generates the legal states that result from trying the four actions (blank moves Left,Right,Vp or down).
- **Goal Test** : This checks whether the state matches the goal configuration shown in figure 2.4.(Other goal configurations are possible)
- **Path cost**: Each step costs 1,so the path cost is the number of steps in the path.
- **The 8-puzzle** belongs to the family **of sliding-block puzzles**, which are often used as test problems for new search algorithms in AI.
- This general class is known as NP-complete.
- The **8-puzzle** has 9!/2 = 181,440 reachable states and is easily solved.
- The **15 puzzle** (4 x 4 board) has around 1.3 trillion states, an the random instances can be solved optimally in few milli seconds by the best search algorithms.
- The **24-puzzle** (on a 5 x 5 board) has around 10^{25} states ,and random instances are still quite difficult to solve optimally with current machines and algorithms.

8-queens problem

- The goal of 8-queens problem is to place 8 queens on the chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal).
- The following figure shows an attempted solution that fails: the queen in the right most column is attacked by the queen at the top left.
- An **Incremental formulation** involves operators that augments the state description,starting with an empty state.for 8-queens problem,this means each action adds a queen to the state.
- A **complete-state formulation** starts with all 8 queens on the board and move them around.

In either case the path cost is of no interest because only the final state counts.



8-queens problem

- The first incremental formulation one might try is the following :
    - **States** : Any arrangement of 0 to 8 queens on board is a state.
    - **Initial state** : No queen on the board.
    - **Successor function** : Add a queen to any empty square.
    - **Goal Test** : 8 queens are on the board,none attacked.
- In this formulation, we have 64.63…57 = 3 x 10^{14} possible sequences to investigate.
- A better formulation would prohibit placing a queen in any square that is already attacked. :
    - **States** : Arrangements of n queens (0 <= n < = 8) ,one per column in the left most columns ,with no queen attacking another are states.
    - **Successor function** : Add a queen to any square in the left most empty column such that it is not attacked by any other queen.
- This formulation reduces the 8-queen state space from 3 x 10^{14} to just 2057,and solutions are easy to find.
- For the 100 queens the initial formulation has roughly 10^{400} states whereas the improved formulation has about 10^{52} states.
- This is a huge reduction, but the improved state space is still too big for the algorithms to handle.

REAL WORLD PROBLEMS

- A real world problem is one whose solutions people actually care about.
- They tend not to have a single agreed upon description, but attempt is made to give general flavor of their formulation,
- The following are the some real world problems,
    - Route Finding Problem
    - Touring Problems
    - Travelling Salesman Problem
    - Robot Navigation

ROUTE-FINDING PROBLEM

- Route-finding problem is defined in terms of specified locations and transitions along links between them.
- Route-finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, and air line travel planning systems.

AIRLINE TRAVEL PROBLEM

The **airline travel problem** is specifies as follows **:**

- **States :** Each is represented by a location(e.g.,an airport) and the current time.
- **Initial state :** This is specified by the problem.
- **Successor function :** This returns the states resulting from taking any scheduled flight(further specified by seat class and location),leaving later than the current time plus the within-airport transit time,from the current airport to another.
- **Goal Test :** Are we at the destination by some prespecified time?
- **Path cost :** This depends upon the monetary cost,waiting time,flight time,customs and immigration procedures,seat quality,time of dat,type of air plane,frequent-flyer mileage awards, and so on.

TOURING PROBLEMS

- **Touring problems** are closely related to route-finding problems,but with an important difference.
- Consider for example, the problem, "Visit every city at least once" as shown in Romania map.
- As with route-finding the actions correspond to trips between adjacent cities. The state space, however,is quite different.
    - **Initial state** would be "In Bucharest; visited{Bucharest}".
    - **Intermediate state** would be "In Vaslui; visited {Bucharest,Vrziceni,Vaslui}".
    - **Goal test** would check whether the agent is in Bucharest and all 20 cities have been visited.

THE TRAVELLING SALESPERSON PROBLEM (TSP)

- ✓ TSP is a touring problem in which each city must be visited exactly once.
- ✓ The aim is to find the shortest tour. The problem is known to be **NP-hard**.
- ✓ Enormous efforts have been expended to improve the capabilities of TSP algorithms.
- ✓ These algorithms are also used in tasks such as planning movements of **automatic circuit-board drills** and of **stocking machines** on shop floors.
- ✓

VLSI layout

A **VLSI layout** problem requires positioning millions of components and connections on a chip to minimize area ,minimize circuit delays,minimize stray capacitances,and maximize manufacturing yield. The layout problem is split into two parts : **cell layout** and **channel routing**.

ROBOT navigation

ROBOT navigation is a generalization of the route-finding problem. Rather than a discrete set of routes,a robot can move in a continuous space with an infinite set of possible actions and states. For a circular Robot moving on a flat surface,the space is essentially two-dimensional.

When the robot has arms and legs or wheels that also must be controlled,the search space becomes multi-dimensional. Advanced techniques are required to make the search space finite.

AUTOMATIC ASSEMBLY SEQUENCING

The example includes assembly of intricate objects such as electric motors. The aim in assembly problems is to find the order in which to assemble the parts of some objects. If the wrong order is choosen, there will be no way to add some part later without undoing somework already done.

Another important assembly problem is protein design, in which the goal is to find a sequence of

Amino acids that will be fold into a three-dimensional protein with the right properties to cure some disease.

INTERNET SEARCHING

In recent years there has been increased demand for software robots that perform Internet searching, looking for answers to questions, for related information, or for shopping deals.

The searching techniques consider internet as a graph of nodes (pages) connected by links.

MEASURING PROBLEM-SOLVING PERFORMANCE

- ✓ The output of problem-solving algorithm is either failure or a solution. (Some algorithms might struck in an infinite loop and never return an output.)
- ✓ The algorithm's performance can be measured in four ways :
    - o **Completeness**: Is the algorithm guaranteed to find a solution when there is one?
    - o **Optinality** : Does the strategy find the optimal solution
    - o **Time complexity:** How long does it take to find a solution?
    - o **Space complexity:** How much memory is needed to perform the search?

UNINFORMED SEARCH STRATGES

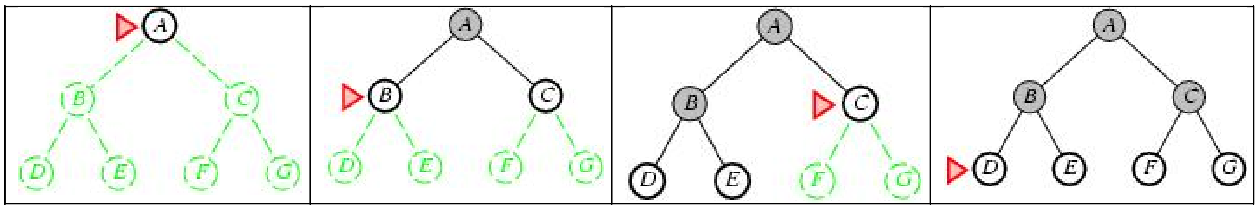
- ✓ **Uninformed Search Strategies** have no additional information about states beyond that provided in the **problem definition**.
- ✓ **Strategies** that know whether one non goal state is "more promising" than another are called I**nformed search or heuristic search** strategies.

There are five uninformed search strategies as given below.

- o Breadth-first search
- o Vniform-cost search
- o Depth-first search
- o Depth-limited search
- o Iterative deepening search
- o Bidirectional Search

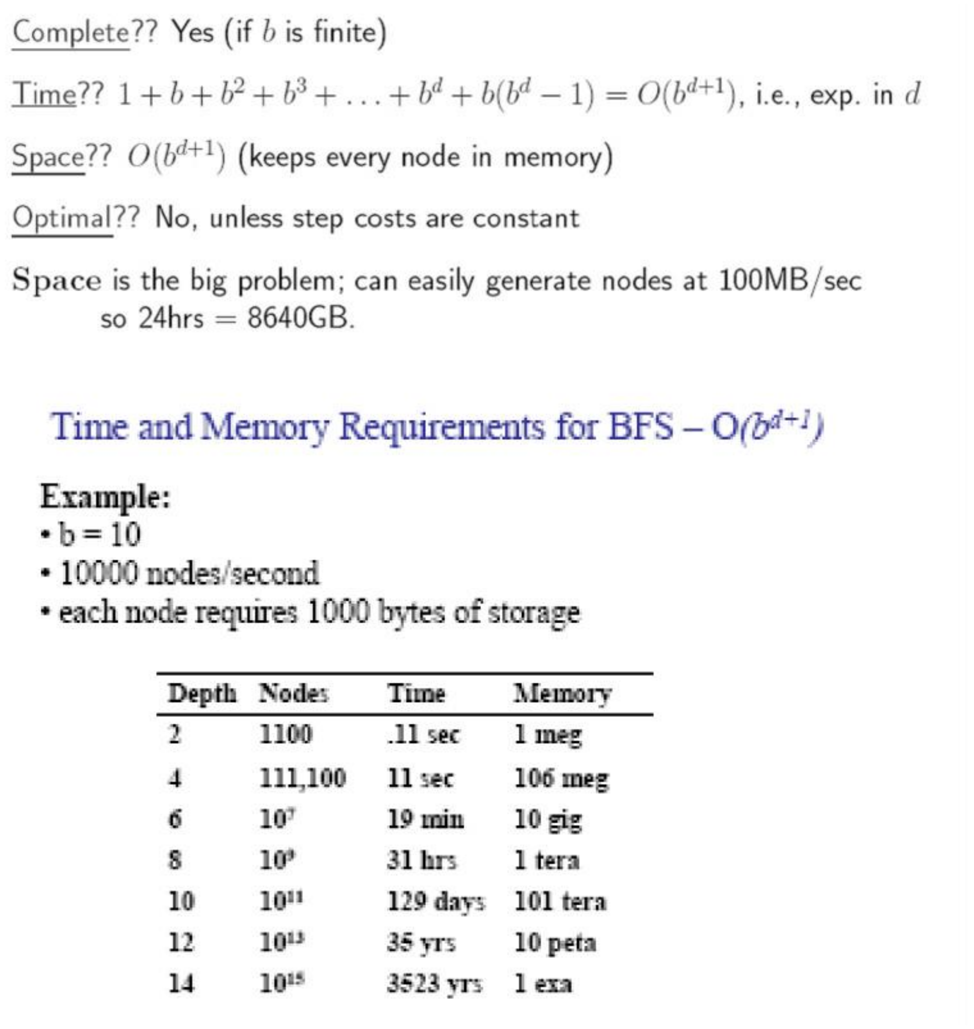
Breadth-first search

- ✓ 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-QVEVE()) 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.



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.

Time and Memory Requirements for BFS – $O(b^{d+1})$

Example:
- b = 10
- 10000 nodes/second
- each node requires 1000 bytes of storage

Depth	Nodes	Time	Memory
2	1100	.11 sec	1 meg
4	111,100	11 sec	106 meg
6	10^7	19 min	10 gig
8	10^9	31 hrs	1 tera
10	10^{11}	129 days	101 tera
12	10^{13}	35 yrs	10 peta
14	10^{15}	3523 yrs	1 exa

Time and memory requirements for breadth-first-search.

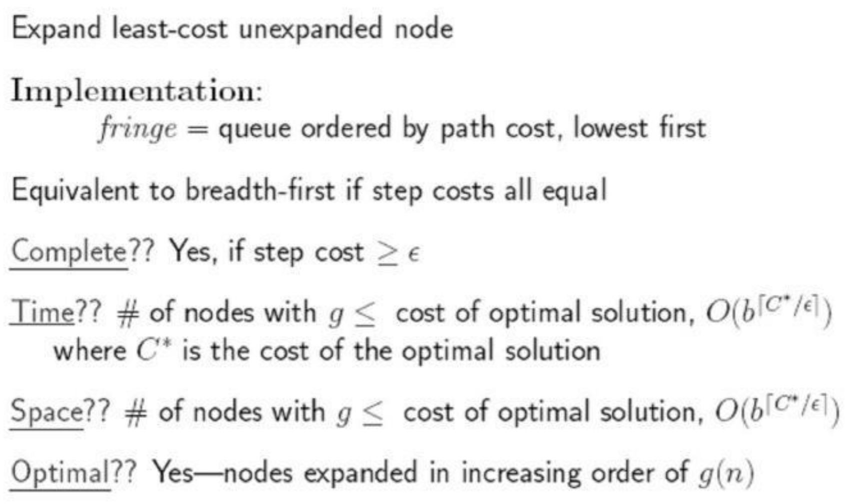
Time complexity for BFS

- ✓ 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
  $b + b^2 + b^3 + \ldots + b^d + (bd^{+1} + b) = O(b^{d+1).}$
- ✓ 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

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.

Properties of Uniform-cost-search:



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 $\geq \epsilon$

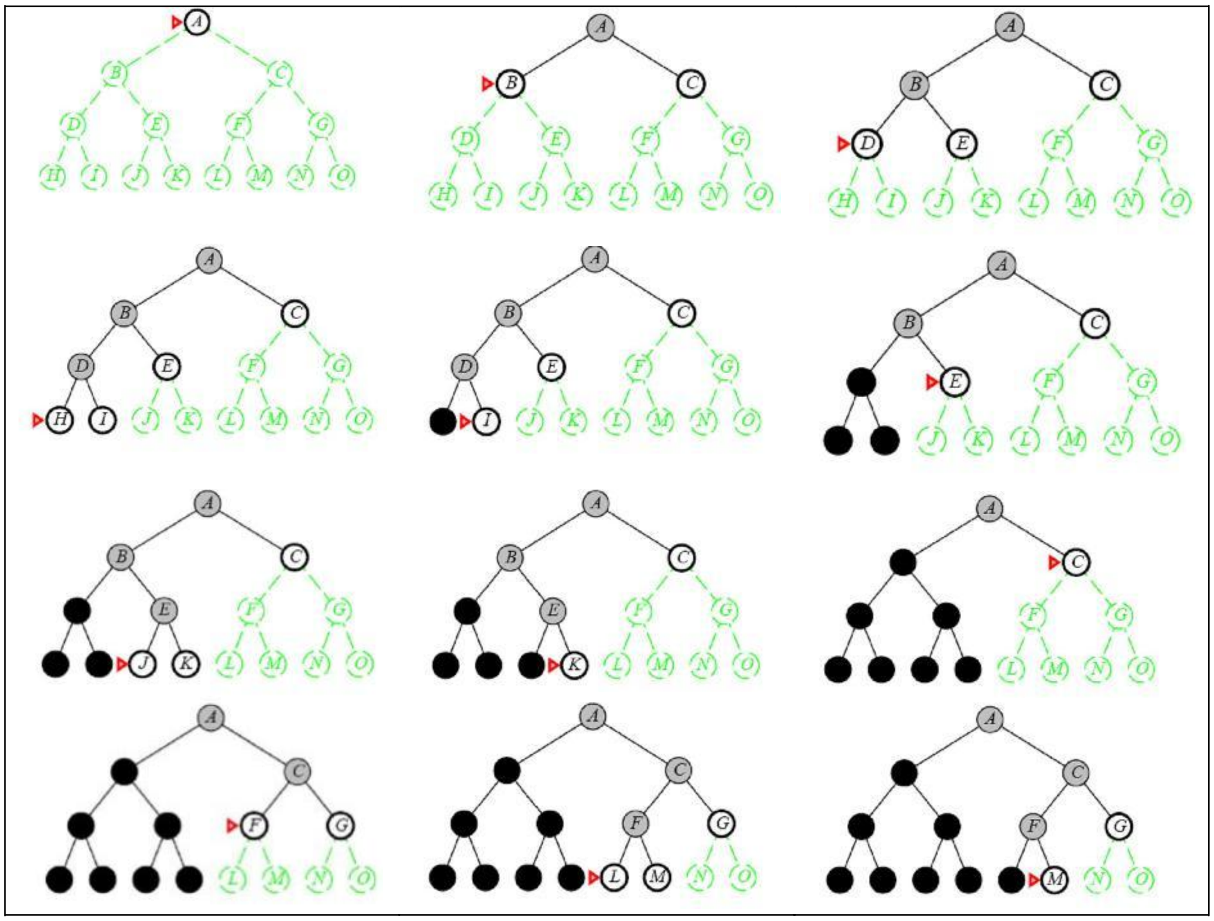
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

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

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

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



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

Drawback of Depth-first-search

The drawback of depth-first-search is that it can make a wrong choice and get stuck going down very long(or even infinite) path when a different choice would lead to solution near the root of the search tree.

For example, depth-first-search will explore the entire left subtree even if node C is a goal node.

BACKTRACKING SEARCH

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

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 solves the infinite path problem.
- Depth limited search will be nonoptimal if we choose l > d. Its time complexity is $O(b^l)$ and its space compleiy is O(bl).
- Depth-first-search can be viewed as a special case of depth-limited search with l = oo
- 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 if there is a solution.,it must be of length 19 at the longest,So l = 10 is a possible choice.
- However,it oocan 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.
- 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
- Recursive implementation of Depth-limited-search:

```
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
```

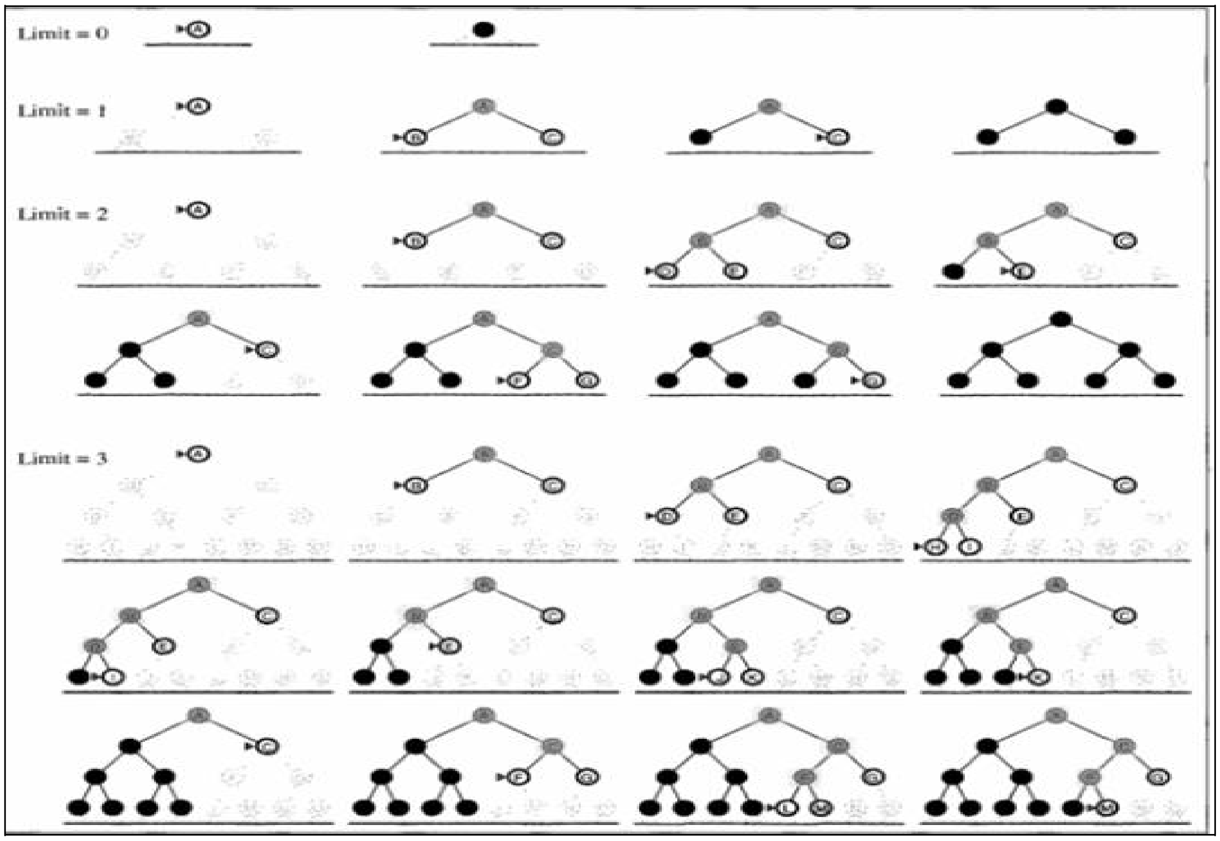
ITERATIVE DEEPENING DEPTH-FIRST SEARCH

- 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.

- Iterative deepening combines the benefits of depth-first and breadth-first-search
- Like depth-first-search,its memory requirements are modest;O(bd) to be precise.
- Like Breadth-first-search,it is complete when the branching factor is finite and optimal when the path cost is a non decreasing function of the depth of the node.
- The following figure shows the four iterations of ITERATIVE-DEEPENING_SEARCH on a binary search tree,where the solution is found on the fourth iteration.

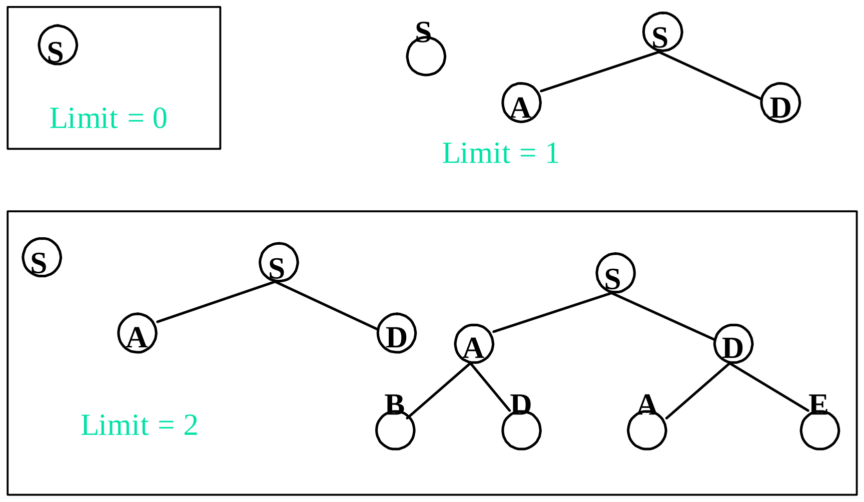
```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ cutoff then return result
    end
```

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.



Four iterations of iterative deepening search on a binary tree

Iterative deepening search



- 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:

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

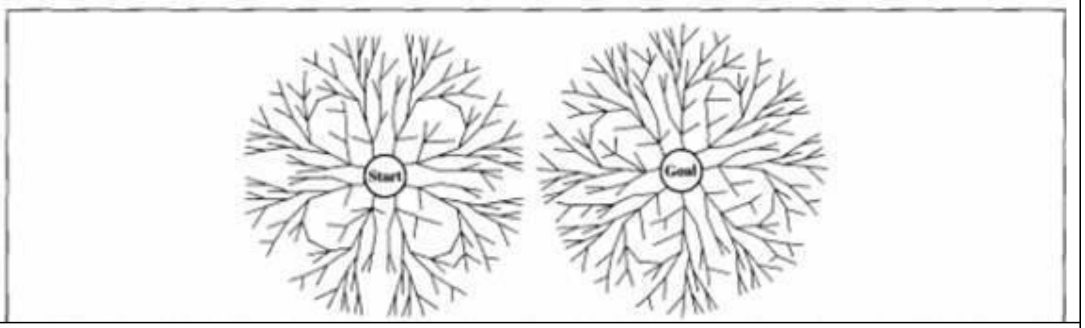
IDS does better because other nodes at depth d are not expanded

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

- In general,iterative deepening is the prefered uninformed search method when there is a large search space and the depth of solution is not known.

Bidirectional Search

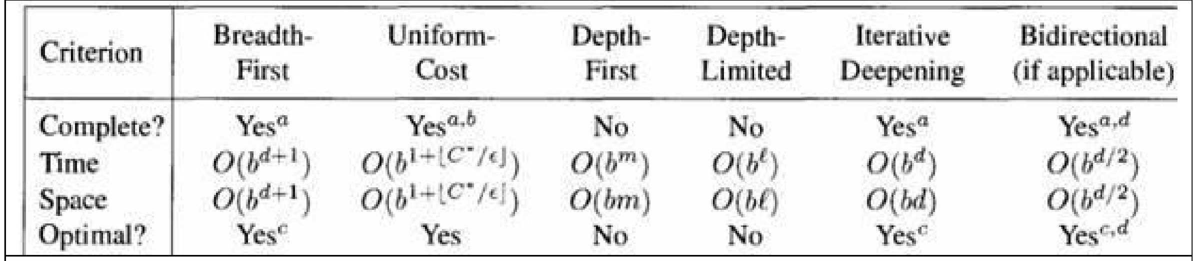
- The idea behind bidirectional search is to run two simultaneous searches
  - ✓ one forward from the initial state and
  - ✓ other backward from the goal,
- It stops when the two searches meet in the middle.
- The motivation is that $b^{d/2} + b^{d/2}$ much less than $b^{d/}$



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

Comparing Uninformed Search Strategies

The following table compares search strategies in terms of the four evaluation criteria.



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

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.

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.

Heuristic functions

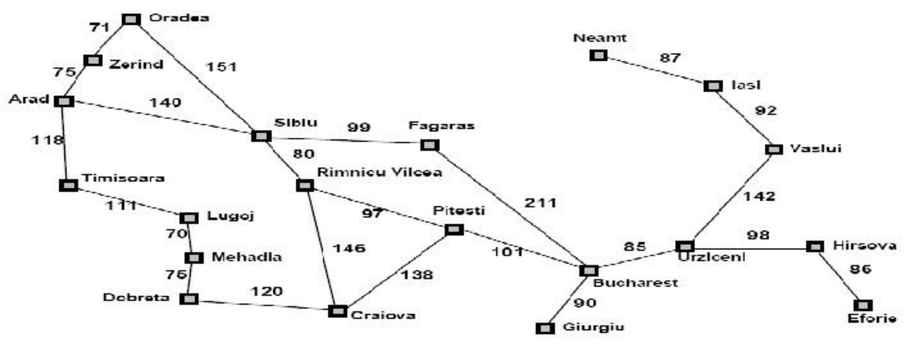
- A **heuristic function** or simply a **heuristic** is a function that ranks alternatives in various search algorithms at each branching step basing on an available information in order to make a decision which branch is to be followed during a search.
- The key component of Best-first search algorithm is a **heuristic function**,denoted by h(n):

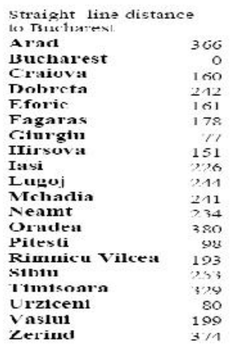
h(n) = estimated cost of the **cheapest path** from node n to a **goal node**.

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

Greedy Best-first search

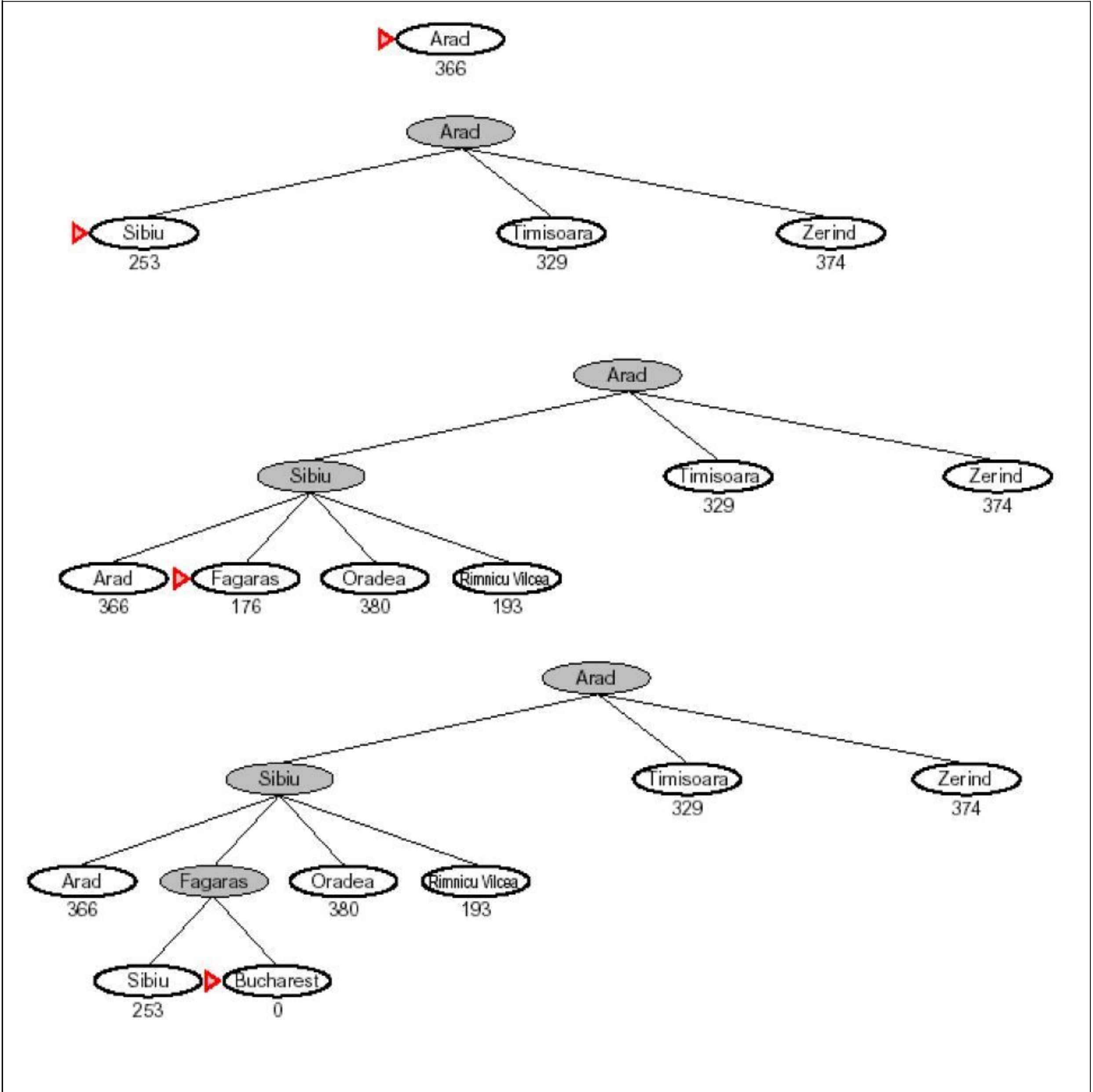
- **Greedy best-first search** tries to expand the node that is closest to the goal,on the grounds that this is likely to a solution quickly.
- It evaluates the nodes by using the heuristic function f(n) = h(n).
- Taking the example of **Route-finding problems** in Romania, the goal is to reach Bucharest starting from the city Arad.
- We need to know the straight-line distances to Bucharest from various cities.
- For example, the initial state is In(Arad) ,and the straight line distance heuristic h_{SLD}(In(Arad)) is found to be 366.
- Vsing the **straight-line distance** heuristic $\mathbf{h_{SLD}}$ **,**the goal state can be reached faster.





Straight line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

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



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

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

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

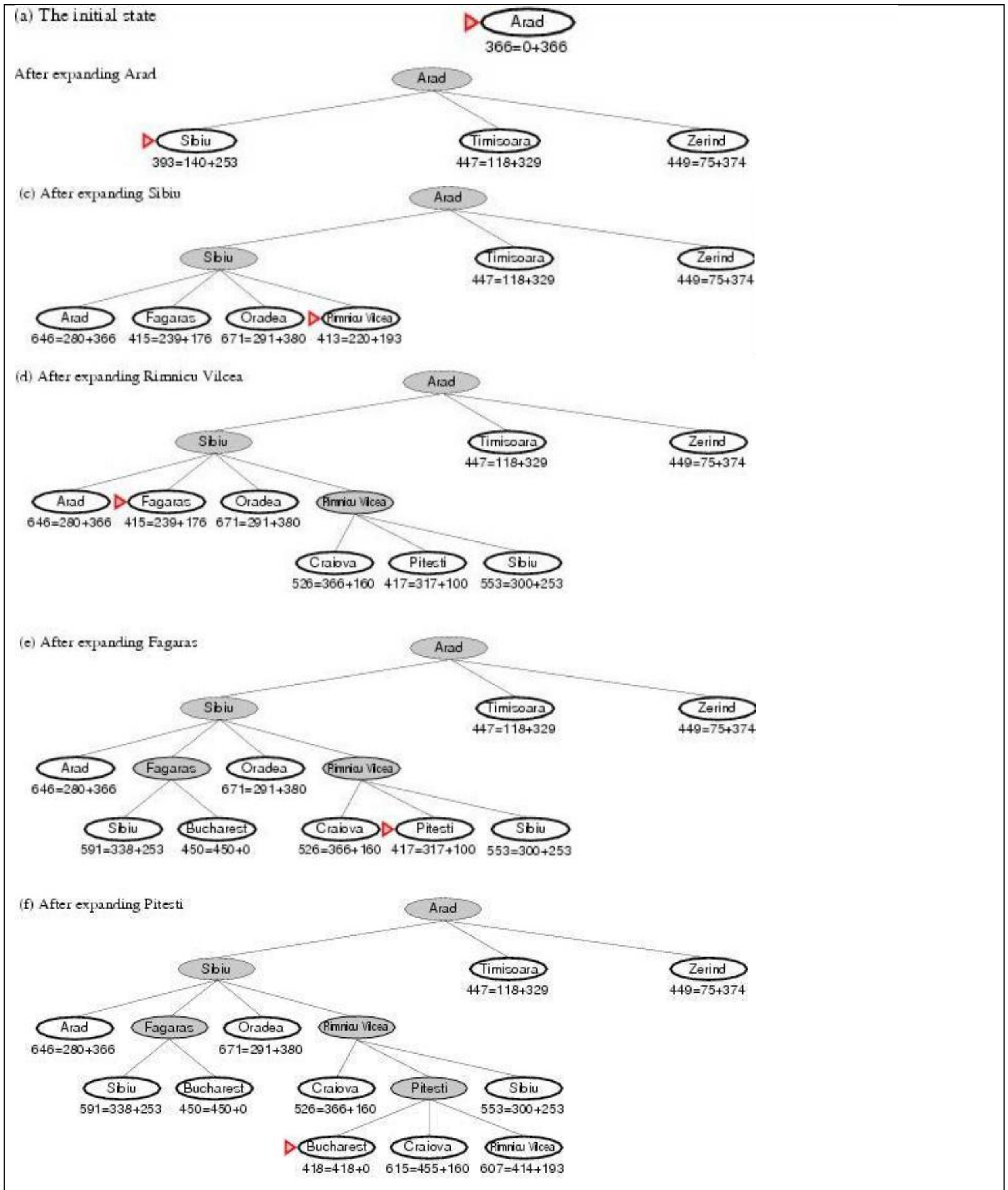
- Greedy best-first search is not optimal,and it is incomplete.
- The worst-case time and space complexity is $O(b^m)$,where m is the maximum depth of the search space.

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 admissible heuristic. The obvious example of admissible heuristic is the straight-line distance h_{SLD}.
- It cannot be an overestimate.
- A^*Search is optimal if h(n) is an admissible heuristic – that is,provided that h(n) never overestimates the cost to reach the goal.
- An obvious example of an admissible heuristic is the straight-line distance h_{SLD} that we used in getting to Bucharest.
- The progress of an A^*tree search for Bucharest is shown in above figure.
- The values of 'g ' are computed from the step costs shown in the Romania,Also the values of h_{SLD} are given in Figure Route Map of Romania.

Recursive Best-first Search (RBFS)

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

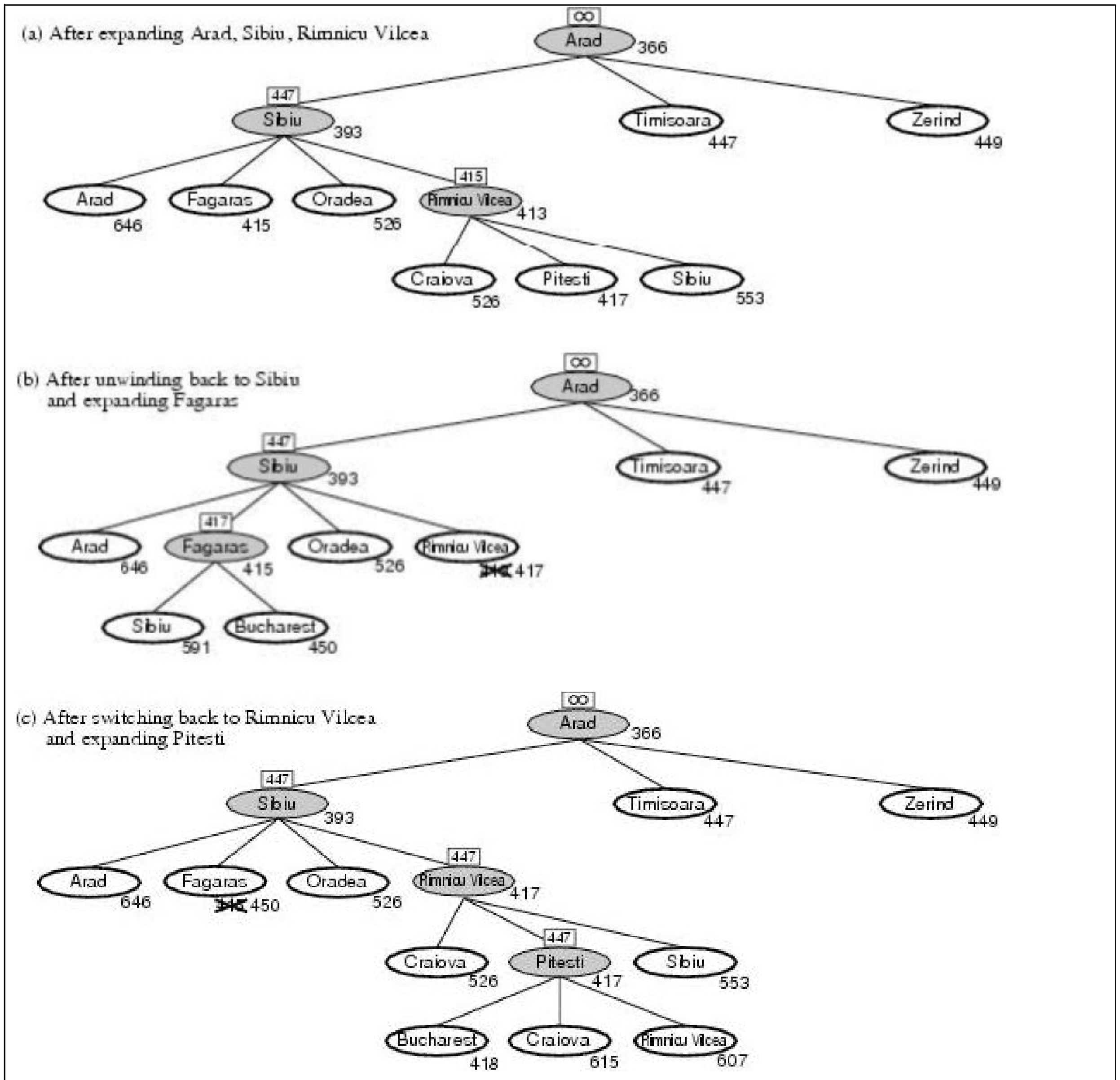


Stages in A* Search for Bucharest. Nodes are labeled with f = g + h . The h-values are the straight-line distances to Bucharest taken from figure Route map of Romania

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

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

The algorithm for recursive best-first search



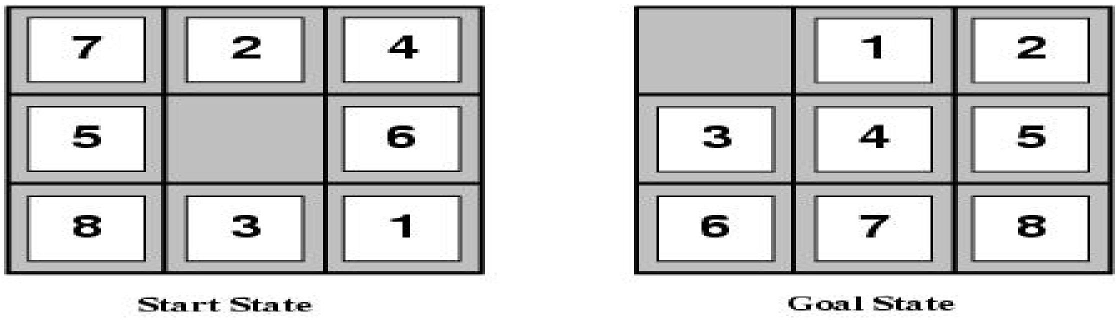
- Stages in an RBFS search for the shortest route to Bucharest. The f-limit value for each recursive call is shown on top of each current node.
- (a) The path via Rimnicu Vilcea is followed until the current best leaf (Pitesti) has a value that is worse than the best alternative path (Fagaras).
- (b) The recursion unwinds and the best leaf value of the forgotten subtree (417) is backed up to Rimnicu Vilcea;then Fagaras is expanded,revealing a best leaf value of 450.
- (c) The recursion unwinds and the best leaf value of the forgotten subtree (450) is backed upto Fagaras; then Rimni Vicea is expanded.
- This time because the best alternative path(through Timisoara) costs atleast 447,the expansion continues to Bucharest

RBFS Evaluation:

- RBFS is a bit more efficient than IDA*
  - Still excessive node generation (mind changes)
- Like A*, optimal if *h(n)* is admissible
- Space complexity is *O(bd)*.
  - IDA* retains only one single number (the current f-cost limit)
- Time complexity difficult to characterize
  - Depends on accuracy if h(n) and how often best path changes.
- IDA* en RBFS suffer from ***too little*** memory.

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



A typical instance of the 8-puzzle.

- The solution is 26 steps long.

The 8-puzzle

- The 8-puzzle is an example of Heuristic search problem.
- The object of the puzzle is to slide the tiles horizontally or vertically into the empty space until the configuration matches the goal configuration
- The average cost for a randomly generated 8-puzzle instance is about 22 steps.
- The branching factor is about 3.(When the empty tile is in the middle,there are four possible moves;when it is in the corner there are two;and when it is along an edge there are three).
- This means that an exhaustive search to depth 22 would look at about 3^{22} approximately = 3.1 X 10^{10} states.
- By keeping track of repeated states, we could cut this down by a factor of about 170, 000, because there are only 9!/2 = 181,440 distinct states that are reachable.
- This is a manageable number, but the corresponding number for the 15-puzzle is roughly 10^{13}.

- If we want to find the shortest solutions by using A^*,we need a heuristic function that never overestimates the number of steps to the goal.
- The two commonly used heuristic functions for the 15-puzzle are :

  (1) h_1 = the number of misplaced tiles.

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

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

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

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

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

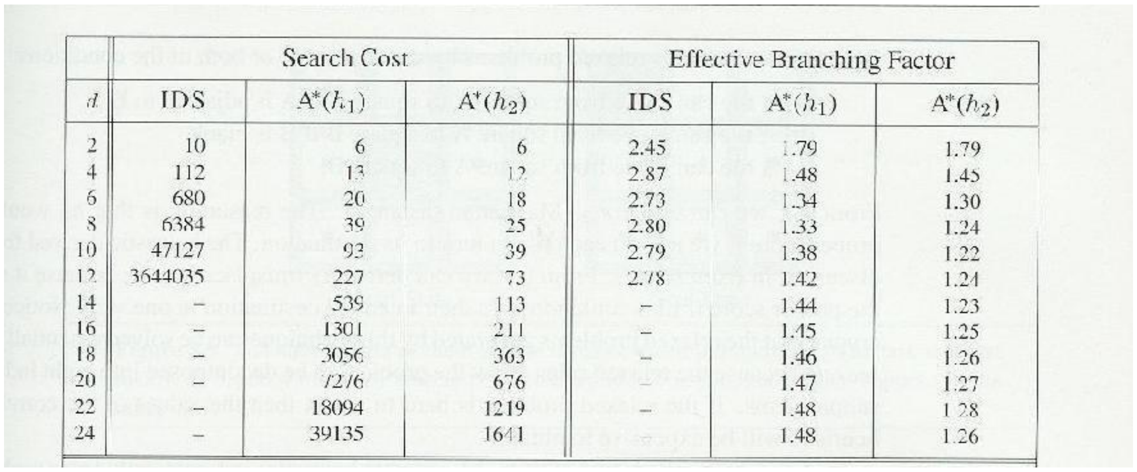
- Neither of these overestimates the true solution cost, which is 26.

The Effective Branching factor

- One way to characterize the **quality of a heuristic** is the **effective branching factor b*.** If the total number of nodes generated by A* for a particular problem is **N**,and the **solution depth** is **d,**then b^* is the branching factor that a uniform tree of depth d would have to have in order to contain N+1 nodes. Thus,

$$N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$$

- For example,if A^* finds a solution at depth 5 using 52 nodes,then effective branching factor is 1.92.
- A well designed heuristic would have a value of b^* close to 1,allowing failru large problems to be solved.
- To test the heuristic functions h_1 and h_2,1200 random problems were generated with solution lengths from 2 to 24 and solved them with iterative deepening search and with A^* search using both h_1 and h_2.
- The following table gives the average number of nodes expanded by each strategy and the effective branching factor.
- The results suggest that h_2 is better than h_1,and is far better than using iterative deepening search.
- For a solution length of 14,A^* with h_2 is 30,000 times more efficient than uninformed iterative deepening search.



	Search Cost			Effective Branching Factor		
d	IDS	$A^*(h_1)$	$A^*(h_2)$	IDS	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

Comparison of search costs and effective branching factors for the ITERATIVE-DEEPENING-SEARCH and A^* Algorithms with h_1,and h_2. Data are average over 100 instances of the 8-puzzle,for various solution lengths.

Inventing admissible heuristic functions

Relaxed problems

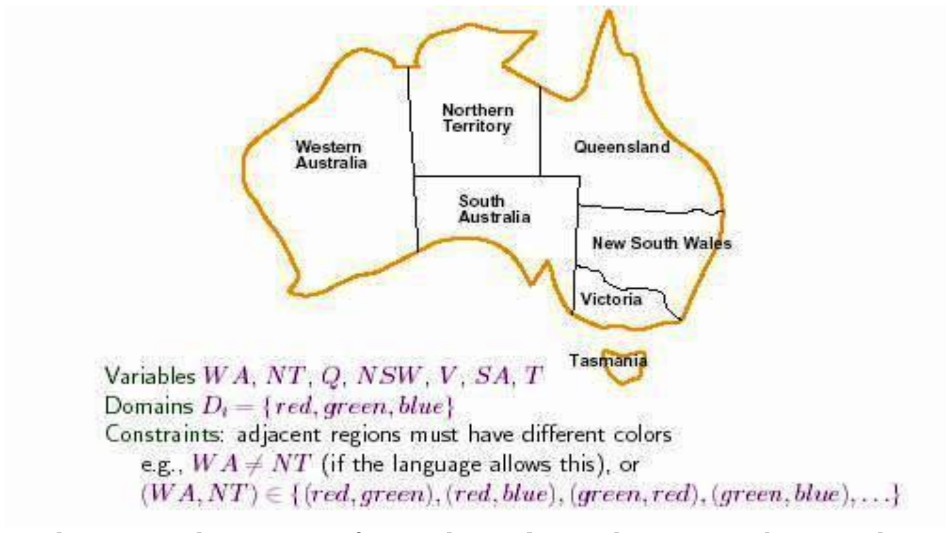
- A problem with fewer restrictions on the actions is called a ***relaxed problem***
- The cost of an optimal solution to a relaxed problem is an admissible heuristic for the original problem
- If the rules of the 8-puzzle are relaxed so that a tile can move *anywhere,* then *hi(n)* gives the shortest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

CONSTRAINT SATISFACTION PROBLEMS (CSP)

- A **Constraint Satisfaction Problem** (or CSP) is defined by a
  - ✓ set of **variables** X_1, X2….X_n, and a
  - ✓ set of constraints C_1,C_2,…,C_m.
  - ✓ Each variable X_i has a nonempty **domain** D,of possible **values**.
  - ✓ Each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset.

- A **State** of the problem is defined by an **assignment** of values to some or all of the variables,{$X_i = v_i$,$X_j = v_j$,…}.
- 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.

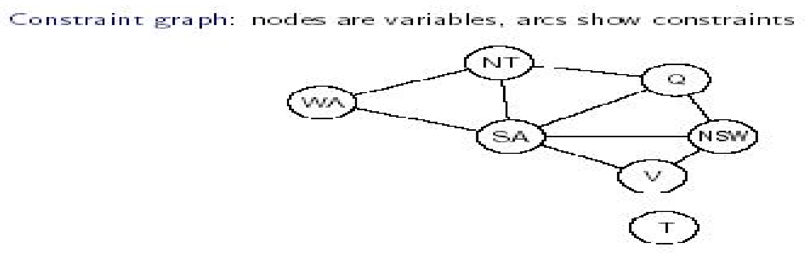
- Some CSPs also require a solution that maximizes an **objective function**.
- For Example for Constraint Satisfaction Problem :
- The following figure shows the map of Australia showing each of its states and territories.
- We are given the task of coloring each region either red,green,or blue in such 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}.



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

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

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


Constraint graph: nodes are variables, arcs show constraints
NT
Q
WA
SA
NSW
V
T

The map coloring problem represented as a constraint graph.

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

- Every solution must be a complete assignment and therefore appears at depth n if there are n variables.
- Depth first search algorithms are popular for CSPs

Varieties of CSPs

(i) Discrete variables

Finite domains

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

Infinite domains

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

(ii) CSPs with continuous domains

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

Varieties of constraints :

(i) Unary constraints involve a single variable.

Example: SA # green

(ii) **Binary constraints** involve pairs of variables.

Example: SA # WA

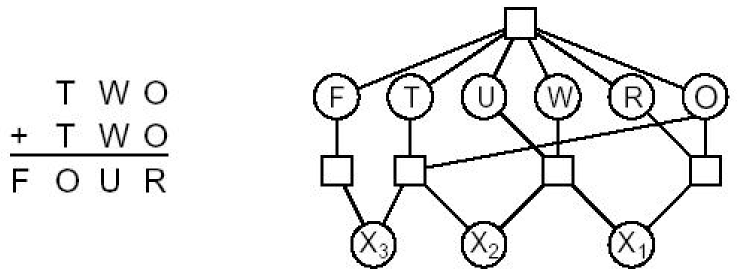
(iii) **Higher order constraints** involve 3 or more variables.

Example: cryptarithmetic puzzles.

(iv) **Absolute constraints** are the constraints, which rules out a potential solution when they are violated

(v) **Preference constraints** are the constraints indicating which solutions are preferred

Example: Vniversity Time Tabling Problem



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)
$O+O=R+10\cdot X_1$, etc.

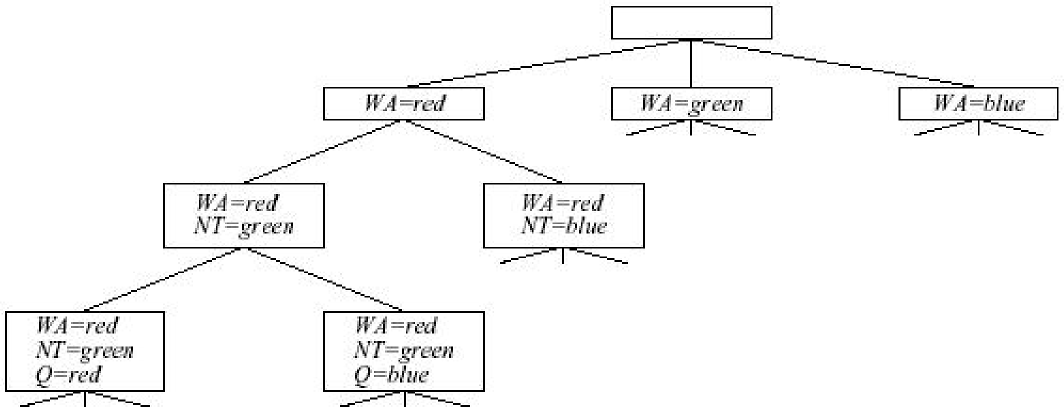
- Cryptarithmetic problem.
- Each letter stands for a distinct digit;the aim is to find a substitution of digits for letters such that the resulting sum is arithmetically correct,with the added restriction that no leading zeros are allowed.
- The constraint hypergraph for the cryptarithmetic problem,showint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

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 following algorithm shows the Backtracking Search for CSP

```
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
```

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



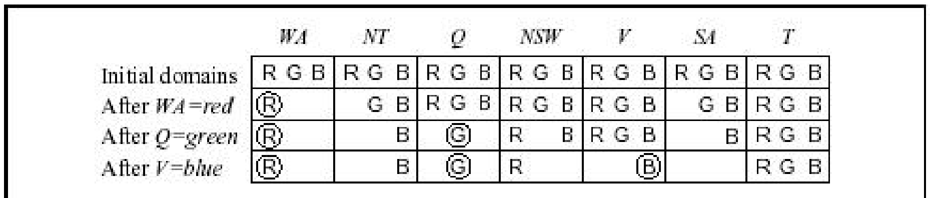
Part of search tree generated by simple backtracking for the map coloring problem.

Propagating information through constraints

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

Forward checking

- One way to make better use of constraints during search is called **forward checking**.
- Whenever a variable X is assigned, the forward checking process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y 's domain any value that is inconsistent with the value chosen for X.
- The following figure 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*	Ⓡ	G B	R G B	R G B	R G B	G B	R G B
After *Q=green*	Ⓡ	B	Ⓖ	R B	R G B	B	R G B
After *V=blue*	Ⓡ	B	Ⓖ	R	Ⓑ		R G B

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

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

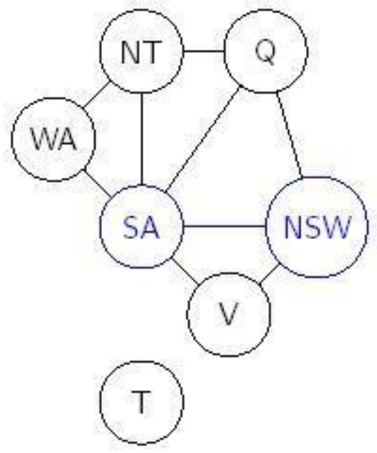
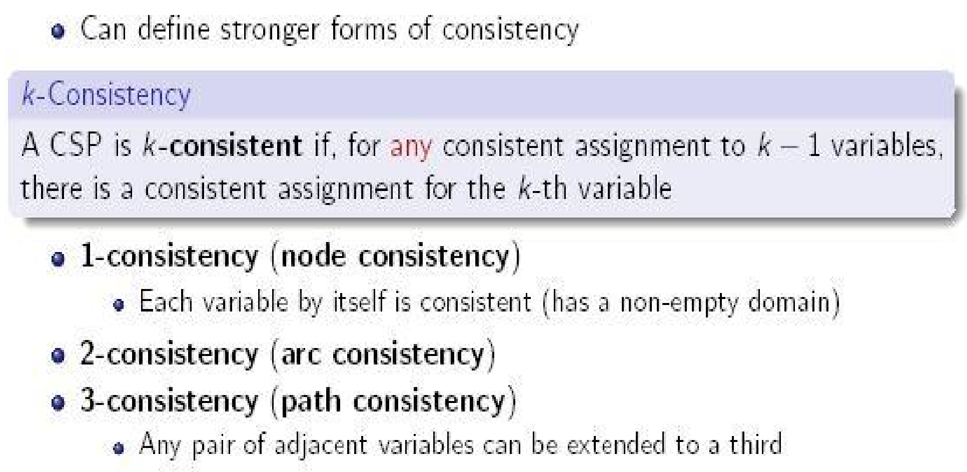


Figure: Australian Territories

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

K-Consistency



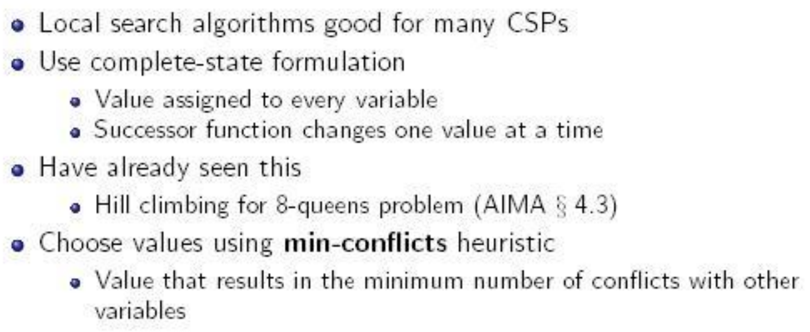
- Can define stronger forms of consistency

k-Consistency

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

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

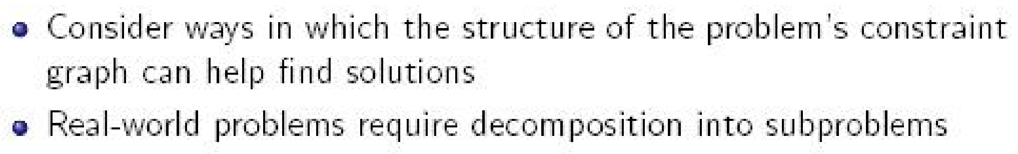
Local Search for CSPs



- 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 Sub problems

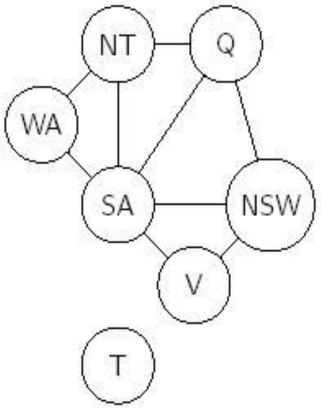


Figure: Australian Territories

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

Tree-Structured CSPs

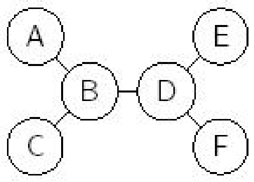


Figure: Tree-Structured CSP

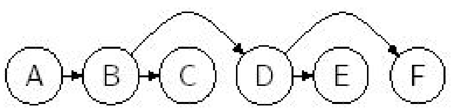


Figure: Linear ordering

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

****

FIRST UNIT-I PROBLEM SOLVING FINISHED

****