MODULE 1

Problems – problem spaces and search, production systems, problem characteristics, Searching strategies – Generate & Test, Heuristic Search Techniques – Hill climbing – issues in hill climbing, General example problems.

Python – Introduction to python – Lists, Dictionaries & Tuples in python – Python implementation of hill climbing.

Introduction

Artificial Intelligence is the study of how to make computers do things which at the moment people do better.

AI Problems

→ Much of the early work in the field focused on formal tasks, such as game playing and theorm proving.

- Checkers playing game pgm and chess pgm comes under game playing category.
- Logic theorist was an early attempt to prove mathematical theorms.
- Game playing & theorm proving share the property that people who do them well are considered to be displaying intelligence. Computers could perform well at these tasks by being fast at exploring a large number of solution paths & then selecting the best one.

→ Another work AI focussed on was commonsense reasoning ie a sort of problem solving that we do everyday when we decide how to get to work in the morning.

- It includes reasoning about physical objects and their relationship to each other, (ie an object can be in only one place at a time) as well as reasoning about actions & their consequences. (ie if you let go of something it will fall down & may break).

- To investigate this sort of reasoning, GPS ie General Problem Solver was built.

⇒ As AI research progressed & techniques for handling larger amount of world knowledge was developed, some tasks were attempted which includes perception (vision and speech), natural language understanding and problem solving in specialized domains such as medical diagnosis & chemical analysis.

- Perceptual tasks are difficult because they involve analog (rather than digital) signals. Also signals are typically very noisy.

- Ability to use language to communicate distinguishes humans from animals. The problem of understanding spoken language is a perceptual problem & is hard to solve. But if the problem is restricted to written language, it becomes easier to solve. This is referred to as natural language understanding.

- Inorder to understand sentences about a topic, it is necessary to know not only about the language itself (ie vocabulary & grammer) but also a good deal about the topic so that unstated assumptions can be recognized.

Some of the Task Domains of AI

Mundane Tasks (or day to day tasks)

- Perception
  - Vision
  - Speech
- Natural language
  - Understanding
  - Generation
  - Translation
- Commonsense reasoning
- Robot control

Formal Tasks

- Games
  - Chess
  - Backgammon
  - Checkers
- Mathematics
  - Geometry
  - Logic
  - Integral calculus
  - Proving properties of programs

Expert Tasks

- Engineering
  - Design
  - Fault finding
- Scientific Analysis
- Financial analysis

- Before studying AI problems & solution techniques, it is important to answer the following 4 Questions.

  1. What are our underlying assumptions about intelligence?
  2. What kinds of techniques will be useful for solving/representing AI problems?
  3. At what level of detail, if at all, are we trying to model human intelligence?
  4. How will we know when we have succeeded in building an intelligent pgm?

1. Underlying Assumption

- At the heart of research in AI lies the physical symbol system hypothesis.
- A physical symbol system consists of a set of entities, called symbols, which are physical patterns that can occur as components of another type of entity called an expression. Thus a symbol structure is composed of a numbers of instances of symbols related in some physical way. Besides these structures, the system also contains a collection of processes that operate on expressions to produce other expressions: processes of creation, modification, reproduction & destruction...

- A physical symbol system states that " a physical symbol s/m has the necessary and sufficient means for general intelligent action.

eg:

system	Symbol	Expressions	Processes
Digital computer	0, 1	00001100	Program
Chess	chess pieces	Position of pieces on board	Legal Chess Moves
Brain	Neurons	Thoughts	Mental operations like thinking

2. AI technique.

-> AI problems span a very broad spectrum. They have very little things in common. However there are techniques that are appropriate for the solution of a variety of these problems.

-> One of the few hard and fast results to come out of the first three decades of AI research is that Intelligence requires knowledge

-> Despite of its overpowering asset, knowledge possesses some less desirable properties which include

- being voluminous
- hard to characterize accurately
- Constantl changing
- It differs from data by being organized in a way that corresponds to the ways it will be used.

-> AI technique is a method that exploits knowledge that should be represented in such a way that:

* Knowledge captures generalization.

It is not necessary to represent seperately each individual situation. Instead situations that share common properties are grouped together. If this property was not present then large amount of memory & updation would be required. Something without this property is called data rather than knowledge.

* It can be understood by people who must provide it. The bulk of data can be acquired automatically for many pgms. (eg: taking readings from a variety of instruments). However in many AI domains, most of the knowledge a program has must be provided by people in terms they understand.

* It can easily be modified to correct errors and to reflect changes in the world & in our world view.

* Can be used in many situations though it is not totally accurate or complete.

* It can be used to help overcome its own sheer bulk by helping narrow down the range of possibilities that can be considered.

eg: Tic-Tac-Toe

It is a two player game with one player marking O and the other marking X, at their turn in the spaces in a 3x3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical or diagonal row wins the game.

Here we are considering one human player and the other player to be a computer pgm. The objective to play the game using computer is to write a pgm which never loses.

let us represent 3x3 board as 9 element vector. Each element in a vector can contain any of the foll 3 digits.

0 - representing blank position
1 - representing x player move
2 - representing 0 player move

Index	Current Board Pos	New Board Pos
0	000000000	000 010 000
1	000 000 001	020000001
2	000 000 002	000 100002
3	000 000 010	002 000 010

eg2: Question Answering. Program.

The program attempts to answer questions using the literal ~~input txt~~ input text. It simply matches text fragments in the questions against the input text.

Data structures

Question patterns : A set of templates that match common Qn forms and produce patterns to be used to match against inputs. Templates and patterns (text patterns) are paired so that if a template matches successfully against an input question then its associated text patterns are used to find appropriate answers in the text.

eg: if template "who did x y" matches input ~~text~~ Qn, then text pattern " x y z" is matched against input text and value z is given as answers to the question.

Text The input text stored simply as a long character string

Question The current Question also stored as character string.

Algorithm

To answer a Qn, do the following

1. Compare each element of Question Patterns against the Question & use all those that match successfully to generate a set of text patterns.

2. Pass each of these patterns through a substitution process that generates alternative forms of verbs. (eg: go & went) This step generates new & expanded set of text patterns.

3. Apply each of these text patterns to Text & collect all resulting answers.

4. Reply with the set of answers and collected

eg: Text:

Mary went shopping for a new coat. She found a red one she really liked. When she got home, she discovered that it went perfectly with her favorite dress

Q1: What did Mary go shopping for?

Q2: What did Mary find that she liked?

Q3: Did Mary buy anything?

→ The first two Que can be answered by following the above mentioned algorithm. But to answer the 3rd Qn, the meaning of the sentences and prior knowledge about objects & situations has to be considered.

⇒ So the 3 important AI techniques are Blind search.

a) Search: Provides a way of solving problems for which no direct approach is available into which any direct techniques that are available can be embedded. [eg: Tic Tac Toe] BFS, DFS.

no extra knowledge

additional info

b) Use of knowledge: Provides a way of solving complex problems by exploiting the structures of the objects that are involved.

c) Abstraction: Provides a way of seperating important features and variations from many unimportant ones that would otherwise overwhelm any process. [Question Answering]

what is ur father family name Question

3) Level of the Model

→ Before we are out to do something, it is good idea to decide exactly what we are trying to do.

- So we must ask ourselves, " what is our goal in trying to produce pgms that do the intellectual things that people do? -

- Are we trying to produce pgms that do the tasks the same way people do?
- Or, are we attempting to produce pgms that simply do the tasks in whatever way appears easiest?

→ Efforts to build pgms that perform tasks the way people do can be divided into 2 classes.

- Pgms in 1st class attempt to solve problem that do not really fit the definition of AI task. They are the problems, that a computer could easily solve.

- Pgms in 2nd class attempt to model human performance by doing things that fall within definition of AI task. They do things that are not trivial for the computer.

* It becomes necessary to model human performance at these sort of tasks:

a) To enable computers to understand human reasoning. eg: ~~for~~ a computer to be able to read a newspaper story and then answer a question " Why did the terrorists kill the hostages? its pgm must be able to simulate the reasoning processes of people.

b) to enable people to understand computer reasoning. People are often reluctant to reply on the o/p of a computer unless they understand how the machine arrived at the result. If the computers reasoning process is similar to that of people, then producing acceptable explaination is much easier.

4) Criteria for Success

How will we know if we have constructed a machine that is intelligent?

- In 1950, Allan Turing proposed the following method for determining whether a machine can think. This method is known as Turing test.

- To conduct this test, we need 2 people and the machine to be evaluated. One person plays the role of the interrogator, who is in a seperate room from the computer and the other person. The interrogator can ask questions of either the person or the computer by typing questions and receiving typed responses. However, the interrogator knows them only as A & B and aims to determine which is the person & which is the machine.

The goal of the machine is to fool the interrogator into believing that it is the person. If the machine succeeds at this, then we can conclude that the machine can think. The m/c is allowed to do whatever it can to fool the interrogator.

- The above test takes time. It is possible to measure ~~achievemt~~ achievement of AI in more restricted domains.

eg. a) In the case of chess pgm, rating is based on the number of players the pgm can beat.

b) by comparing the time it takes for a program to complete a task to the time taken by a person to do the same task.

envt in which search takes place

Problems, Problem Spaces and Search

To build a system to solve a particular problem, we need to do 4 things :

1. Define the problem precisely : This definition must include precise specifications of what the initial situations will be as well as what final situations constitute acceptable solutions to the problem.

2. Analyze the problem : A few imp features can have an immense impact on the appropriateness of various possible techniques for solving the problem.

3. Isolate and seperate the Task knowledge that is necessary to solve the problem.

4. Choose the best problem-solving technique and apply it to the particular problem.

1) Defining the problem as a State Space Search

- State space search is a process used in the field of Computer Science including AI, in which successive configurations or states of an instance are considered, with the goal of finding a goal state with a desired property.

- Problems are often modelled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation that can be performed to transform the first state into the second.

- Suppose we start with the problem "Play chess". Inorder to build a pgm that could play chess, we would first have to specify the starting position of the chess board, the rules that define the legal moves, and the board positions that represent a win for one side or the other.

- In addition, we must make explicit the previously implicit goal of not only playing a legal game of chess but also winning the game, if possible.

- For the problem "Play Chess", it is fairly easy to provide a formal and complete problem description. starting position can be described as an 8x8 array where each position contains a symbol standing for the appropriate piece.

Goal : can be defined as any board position in which the opponent does not have a legal move & his/her king is under attack.

Legal moves provide a way of getting from the initial state to a goal state. They can be described as a set of rules that consists of 2 parts :

a) left side that serves as a pattern to be matched against the current board position

b) right side that describes the change to be made to the board position to reflect the move.

→ There are several ways in which the rules can be written.

Fig1

Starting position

	a	b	c	d	e	f	g	h
1	Rook	Knight	Bishop	Queen	King	Bishop	Knight	Rook.
2	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn	Pawn.

Rule

Fig2

white pawn at
square (board file e, rank 2)
AND
square (file e, rank 3) is empty
AND
square (file e, rank 4) is empty

→ move pawn from square (file e, rank 2) to square (file e, rank 4)

- If we write rules like in fig1, we'll have to write a very large number of them since there has to be a seperate rule for each of the roughly 10^{120} possible board positions. Using so many rules poses two serious practical difficulties.

a) No person could ever supply a complete set of such rules. It will take too long.

b) No pgm could easily handle all those rules.

- Inorder to minimize such problems, rules describing the legal moves has to be written in a general way. as in fig2.

- Thus the problem of playing chess can be considered as a problem of moving around in a state space, where each state corresponds to a legal position of the board. [set of states corresponds to board positions]

* - The structure of the state space corresponds to the structure of problem solving in 2 imp ways.

  a) It allows for a formal definition of a problem as the need to convert some given situation into some desired situation using a set of permissible operators.

  b) It permits us to define the process of solving a particular problem as a combination of known techniques (each represented as a rule defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state.

- Inorder to show the generality of the state space representation, A water jug problem can be considered.

Water Jug Problem

You are given 2 jugs, a 4 gallon and a 3 gallon one. Neither has any 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 into a 4 gallon jug?

- The state space for this problem is the set of ordered pairs of integers (x,y) such that $x = 0,1,2,3$ or 4 and $y = 0,1,2$ or 3. Here x represents the no. of gallons of water in 4 gallon jug and y represents the quantity of water in the 3 gallon jug.
- Start state is $(0,0)$ and goal state is $(2,n)$ for any value of n.
- The operators or production rules to solve the problem is given below:

fig3

	Condition		
1.	(x,y) if $x<4$	$\rightarrow (4,y)$	Fill the 4-gallon jug.
2.	(x,y) if $y<3$	$\rightarrow (x,3)$	Fill the 3-gallon jug
3.	(x,y) if $x>0$	$\rightarrow (x-d, y)$	Pour some water out of 4 gallon jug
4.	(x,y) if $y>0$	$\rightarrow (x, y-d)$	Pour some water out of 3 gallon jug.
5.	(x,y) if $x>0$	$\rightarrow (0,y)$	Empty the 4 gallon jug on the ground

6. (x,y) if $y>0$ $\rightarrow$ $(x,0)$ Empty the 3-gallon jug on the ground.

7. (x,y) if $x+y \geq 4$ and $y>0$ $\Rightarrow$ $(4, y-(4-x))$ Pour water from 3 gallon jug into 4 gallon jug until 4 gallon jug is full

8. (x,y) if $x+y \geq 3$ and $x>0$ $\rightarrow$ $(x-(3-y), 3)$ Pour water from 4 gallon jug into 3 gallon until 3 gallon is full

9. (x,y) if $x+y \leq 4$ and $y>0$ $\rightarrow$ $(x+y, 0)$ Pour all water from 3 gallon to 4 gallon jug

10. (x,y) if $x+y \leq 3$ and $x>0$ $\rightarrow$ $(0, x+y)$ Pour all water from 4 gallon jug to 3 gallon jug.

11. $(0,2)$ $\rightarrow$ $(2,0)$ Pour the 2 gallons from the 3 gallon jug into 4 gallon.

12. $(2,y)$ $\rightarrow$ $(0,y)$ Empty the 2 gallons in the 4 gallon jug on the ground.

- Inorder to describe operators completely, some assumptions has to be made which is not mentioned problem stmt.

  Assumptions are we can fill a jug from the pump, we can pour water out of a jug onto the ground, we can pour water from one jug to another and there is no other measuring devices available.

- There are several sequences of operators that solve the problem. One such solution sequence is given below in fig4.

  It is often desirable to find the smallest sequence that reaches the goal state.

Fig 4

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	2
0	3	9
3	0	2
3	3	7
4	2	5 ~~or 12~~
0	2	9 / or 11
2	0	

One Solution to the Water Jug Problem.

→ Inorder to provide a formal description of a problem, the following has to be done.

a) Define a state space that contains all the possible configuration of the relevant objects.

b) Specify one or more states within that space that describe possible situations from which the problem solving process may start. These states are called Initial states

c) ~~Define~~ Specify one or more states that would be acceptable as solutions to the problem. These states are called goal states.

d) Specify a set of rules that describes the actions available. This will require thought on issues like → the assumptions used
→ generality of rules.

Production Systems

- search forms the core of many intelligent processes. So it is useful to structure AI programs in a way that facilitates describing & performing the search process. Production systems provide such structures.

- A production system consists of
  - A set of rules, each consisting of a left side that determines the applicability of the rule and a right side that describes the operations to be performed if the rule is applied.
  - One or more knowledge/databases that contain whatever information is appropriate for the particular task. ~~Some parts of the database~~
  - A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving conflicts that arise when several rules match at once.
  - A rule applier.

- Inorder to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space.

- The problem is then solved by searching for a path through the space from an initial state to a goal state. The process of solving the problem can usefully be modelled as a production system.

Control Strategies

- There can be more than one rule whose left side matches the current state. So we cannot ignore the question of how to decide which rule to apply next during the process of searching for a solution to a problem.

- The 1st requirement of a good control strategy is that it causes motion.
    - Suppose we implement water jug problem with a simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one.
    - If we do so we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water.
    - Control strategies that do not cause motion will never lead to solution.

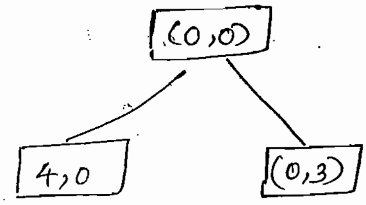
- The 2nd requirement of a good control strategy is that it should be systematic.
    - If we follow a control strategy for water jug problem which is to choose at random from available rules
    - This strategy is better than the 1st (ie above strategy) as it will cause motion & lead to a solution eventually.

- However we are likely to arrive at the same state several times during the process and to use many more steps than are necessary.

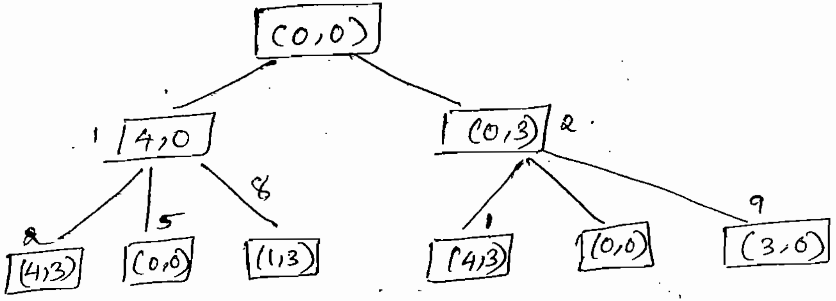
- Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution.

- One systematic control strategy for the water jug problem is as follows: Construct a tree with initial state as root. Generate all the offspring of the root by applying each of the applicable rules to the initial state.

  Now for each leaf node, generate all its successors by applying all the rules that are appropriate. Continue this process until some rule produces a goal state. This process is called breadth first search. (BFS)

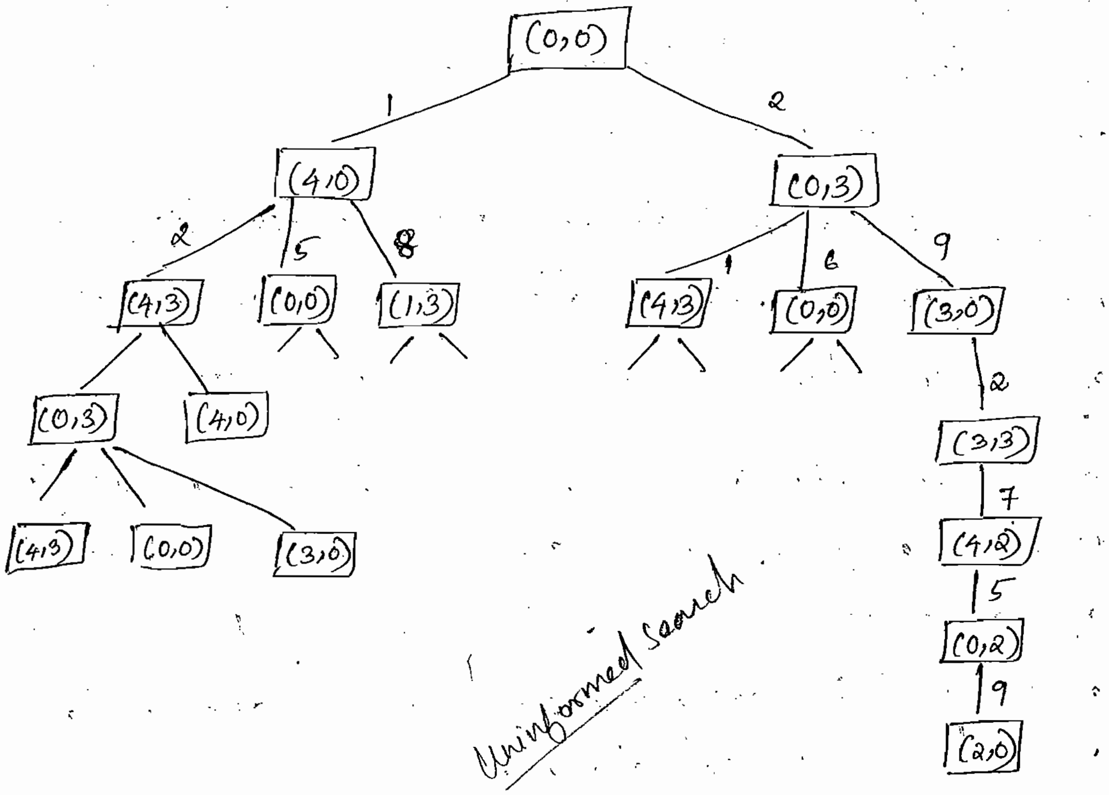


One level of a Breadth-First Search Tree.



Two levels of a Breadth First search Tree.

(ignoring Rules 3, 4, 11 and 12)



Uninformed search

Algorithm: Breadth First Search. (implemented as a FIFO queue i.e. the path selected is the one added earliest)

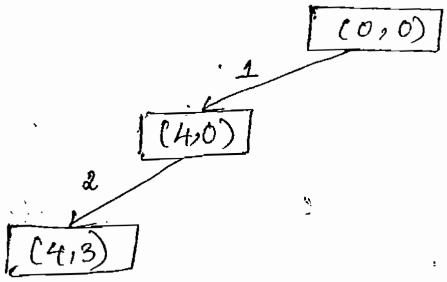
```
1. Create a variable called NODE-LIST and set it to the initial
   state.
2. Until a goal state is found or NODE-LIST is empty:
    a) Remove the first element from NODE-LIST and call
       it E. If NODE-LIST was empty, quit.
    b) For each ~~way that each~~ rule that ~~can~~ match the state
 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

Other systematic control strategies are also available. For eg, we could persue a single branch of the tree until it yields a solution or until a decision to terminate the path is made. A path can be terminated if it reaches the dead end. In such case a backtracking occurs. The most recently created state from which alternative moves are available will be revisited and a newstate will be created. This search procedure is known as Depth First Search. (DFS)

Algorithm : Depth-First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signaled.
   a) Generate a successor, E of the initial state. If there are no more successors, signal failure.
   b) Call Depth-First Search with E as the initial state.
   c) If success is returned, signal success. Otherwise continue in this loop.

eg: Snapshot of a DFS for the water jug problem.



Advantages of Depth-First Search

- DFS requires less memory since only the nodes on the current path are stored.
  This contrasts with BFS, where all of the tree that has so far been generated must be stored.

- DFS may find a solution without examining much of the search space at all.
  This contrasts with BFS in which all parts of the tree must be examined to level n before any node on level n+1 can be examined.

Advantages of BFS

- BFS will not get trapped exploring a blind or dead path.
  This contrasts with DFS, which may follow a single, unfruitful path for a very long time, perhaps forever before the path actually terminates in a state that has no successor.

- If there is a solution, then BFS is guarenteed to find it.
  Furthermore, if there are multiple solutions, then a minimal solution will be found. This is guarenteed by the fact that longer paths are never explored until all shorter ones have already been examined.
  This contrasts with DFS, which may find a long path to a solution in one part of a tree, when a shorter path exists in some other, unexplored part of the tree.

Problem characteristics

- Inorder to choose the most appropriate method for a particular problem, it is necessary to analyze the problem along several key dimension :

1. Is the problem decomposable into a set of independent smaller or easier sub problems ?

2. Can solution steps be ignored or atleast undone if they prove unwise ?

3. Is the problem universe predictable ?

4. Is a good solution to the problem obvious without comparison to all other possible solutions ?

5. Is the desired solution a state of the world or a path to a state ?

6. Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search ?

7. Can a computer that is simply given the problem return the solution, or will the solution of the problem require interaction b/w computer and a person ?

1. Is the Problem Decomposable?

- Suppose we want to solve the problem of computing the expression:
  $\int (x^2+3x+\sin^2 x\cos^2 x)\,dx$

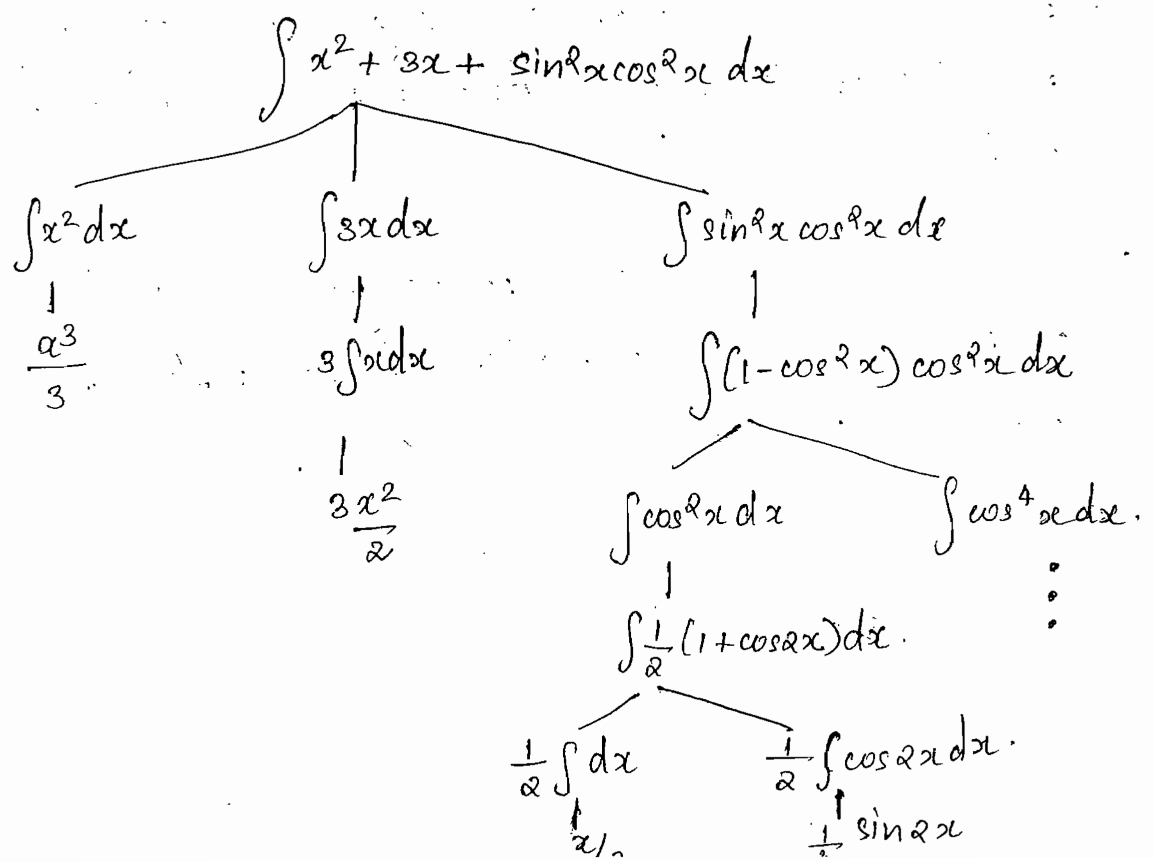
- We can solve this problem by breaking it down into three smaller problems, each of which we can solve by using a small collection of specific rules.

- At each step, it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly.

- If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems.

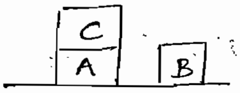
- Using this technique of problem decomposition, we can often solve large problems easily.

- Decomposible problems can be solved by Divide & Conquer technique.
  eg: A decomposible problem.

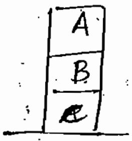

$\int x^2 + 3x + \sin^2 x\cos^2 x\,dx$
$\int x^2 dx$
$\int 3x\,dx$
$\int \sin^2 x\cos^2 x\,dx$
$\frac{x^3}{3}$
$3\int x\,dx$
$\int (1-\cos^2 x)\cos^2 x\,dx$
$\frac{3x^2}{2}$
$\int \cos^2 x\,dx$
$\int \cos^4 x\,dx$
⋮
$\int \frac{1}{2}(1+\cos 2x)dx$
$\frac{1}{2}\int dx$
$\frac{1}{2}\int \cos 2x\,dx$
$x/2$
$\frac{1}{2}\sin 2x$

- Consider the Block World Problem. which is an eg for non decomposible problem.

start :



Goal

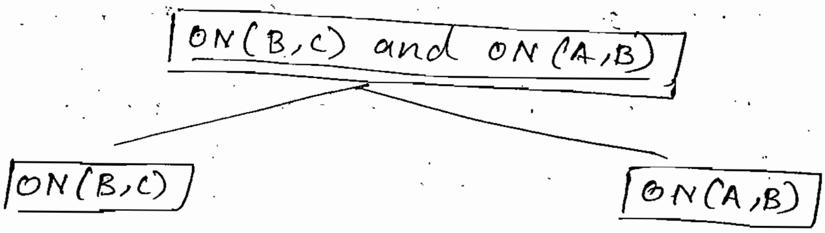


Assume the following operators are available:

i) CLEAR (x) [block x has nothing on it] → ON (x, Table) [Pick up x & put it on the table]

ii) CLEAR (x) and CLEAR (y) → ON (x,y) [put x on y]

- operators allow only to pick a single block at a time
- The problem cannot be decomposed as the 2 subproblems are not independent.



1) CLEAR (C) → ON(C,Table)

2) CLEAR(B) & CLEAR(C) → ON (B,C)

3) CLEAR (A) & CLEAR (B) → ON(A,B)

& Can Solution steps be ignored or undone?

- Suppose we are trying to prove a mathematical theorm. We proceed by first proving a lemma that we think will be useful. Eventually, we realize that the lemma is no help at all. We are still not in trouble as everything we need to prove the theorm is still true. All we have lost is the effort that we spent exploring the alley.

- Consider the 8-Puzzle Problem

8 Puzzle is a square tray in which are placed 8 square tiles. The remaining ninth 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. The game consists of a starting position and a specified goal position. The goal is to transform the starting position into the goal position by sliding the tiles around.

eg: start

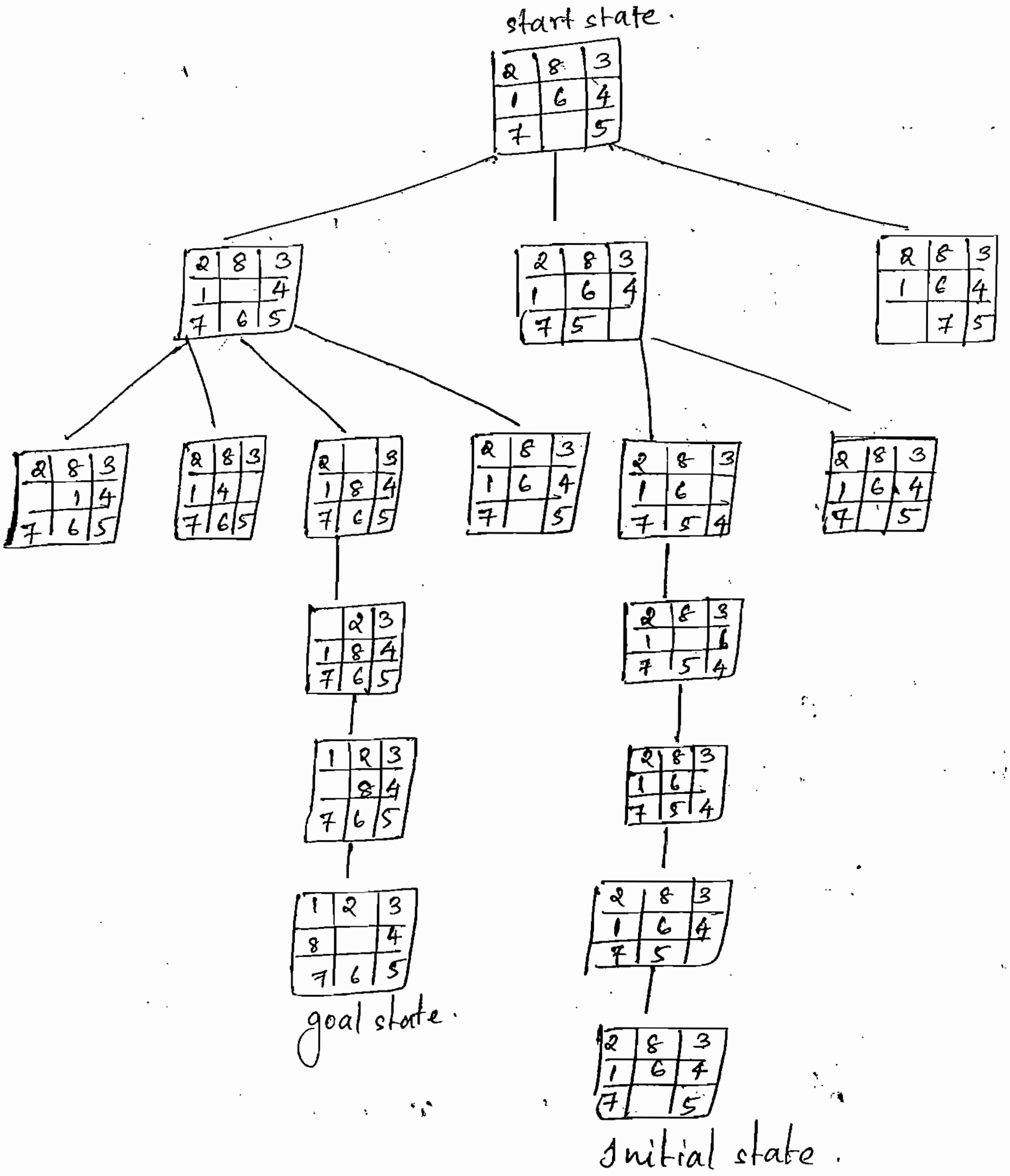
2	8	3
1	6	4
7		5

Goal

1	2	3
8		4
7	6	5

- In attempting to solve the 8-Puzzle, we might make a stupid move. But it is possible to undo the step & reach back to initial state.

- ~~For eg, in the game shown~~



- Mistakes can be recovered but not as easily as theorm proving problem.
- Additional step must be performed to <u>undo each incorrect</u> step whereas no action was required to undo a useless lemma.
- In addition, the control mechanism for an 8-puzzle solver must keep track of the order in which operations are performed so that operations can be undone, if necessary.

- Consider the problem of playing Chess.
  - In the case chess pgm, if one realizes his/her stupid move

after a couple of moves later, it is not possible to back up and start the game over from that point.

- All that can be done is to make the best of the current situation and go on from there.

→ Thus we have 3 classes of problems.

- Ignorable (eg: theorm proving)
  in which solution steps can be ignored.
- Recoverable (eg: 8-Puzzle)
  in which solution steps can be undone.
- Irrecoverable (eg: chess)
  in which solution steps cannot be undone.

- The recoverability of the problem plays an imp role in determining the complexity of the control structure necessary ~~to so~~ for the problem solution.
- Ignorable problem can be solved using a simple control structure that never backtracks. Hence it is easy to implement
- Backtracking is necessary to recover from mistakes made by recoverable problems. So the control structure must be implemented using a push down stack in which decisions are recorded in case they need to be undone later.
- Irrecoverable pblms will need to be solved by a system that expends a great deal of effort making each decision.

since the decision must be final. Much planning is required as the entire sequence has to be analyzed in advance before making a move.

Aim of game each partnership tries to win (or take) as many tricks as

Bridge is played with a deck of 52 cards & 4 people sitting at a square table with the players who are sitting across from each other forming a partnership.

each 13 ace → highest; 4 suits of cards: clubs, diamond, heart, spade.

3. Is the Universe Predictable?

- In 8-puzzle problem, every time we make a move, we know exactly what will happen. This mean that it is possible to plan an entire sequence of move and be confident that we know what the resulting state will be.

- We can use planning to avoid having to undo actual moves, although it becomes necessary to backtrack those moves one at a time during planning process. Thus a control structure that allows backtracking will be necessary.

- However in games other than 8-Puzzle, this planning process may not be possible.

- Suppose we want to play bridge. One of the decisions we will have to make is which card to play on the first trick. We would have to plan the entire hand before making the first play.

But this sort of planning cannot be done with certainity as we cannot know exactly where all the cards are or what the other players will do at their turns. The best we can do is to investigate several plans and use probabilities of the various outcomes to choose a path that has the highest estimated probability of leading to a good score on hand.

- These 2 games illustrate difference b/w certain outcomes

- Planning can be described as problem solving without feedback from the environment

- For certain-outcome problems, the result of an action can be predicted perfectly. Thus planning can be used to generate a sequence of operators that is guarenteed to lead to a solution.

- For uncertain-outcome problems, planning can at best generate a sequence of operators that has good probability of leading to a solution. To plan such problems we need to allow a process called plan revision (ie plan is carried out & necessary feedback is provided)

- One way to solve irrecoverable problem is to plan an entire solution before embarking on an implementation of the plan. But this planning process is effective for certain-outcome problems.

- One of the hardest problems to solve is the irrecoverable, uncertain-outcome.

A few egs of such problems are:

- Playing bridge
- Controlling a robot arm: Outcome is uncertain for a variety of reasons. Someone might move something into the path of the arm. The gears of the arm might stick. A slight error can also be a problem.

4. Is a good solution Absolute or Relative?

- Consider the problem of answering questions based on a database of simple facts such as following:

1. Marcus was a man.
2. Marcus was a Pompeian.
3. Marcus was born in 40 A.D.
4. All men are mortal.
5. All Pompeians died when the volcano erupted in 79 AD.
6. No mortal lives longer than 150 years.
7. It is now 1991 A.D.

- Suppose we ask the question "is Marcus alive?" The below reasoning paths will lead to answer.

		Justification
1.	Marcus was a man	Axiom 1
4.	All men are mortal	Axiom 4
8.	Marcus is mortal	1, 4
3.	Marcus was born in 40 AD	axiom 3
7.	It is now 1991 AD	axiom 7.
9.	Marcus age is 1951 yrs	3, 7
6.	No mortal lives more than 150 yrs	axiom 6
10.	Marcus is dead	8, 6, 9

$\boxed{\text{OR}}$

	Justification.
7. It is now 1991 AD	Axiom 7.
5. All pompeians died in 79 AD	Axiom 5
11. All pompeians are dead now	7,5
2. Marcus was a pompeian	axiom 2.
12. Marcus is dead	11,2

- There are 2 ways of deciding that Marcus is Dead. Since we are interested in finding the answer to the question; it does not matter which path we follow.

- Consider the <u>traveling salesman problem</u>. Our goal is to find the <u>shortest route that visits each city exactly once</u>. Cities to be visited and the distances b/w them are shown below.

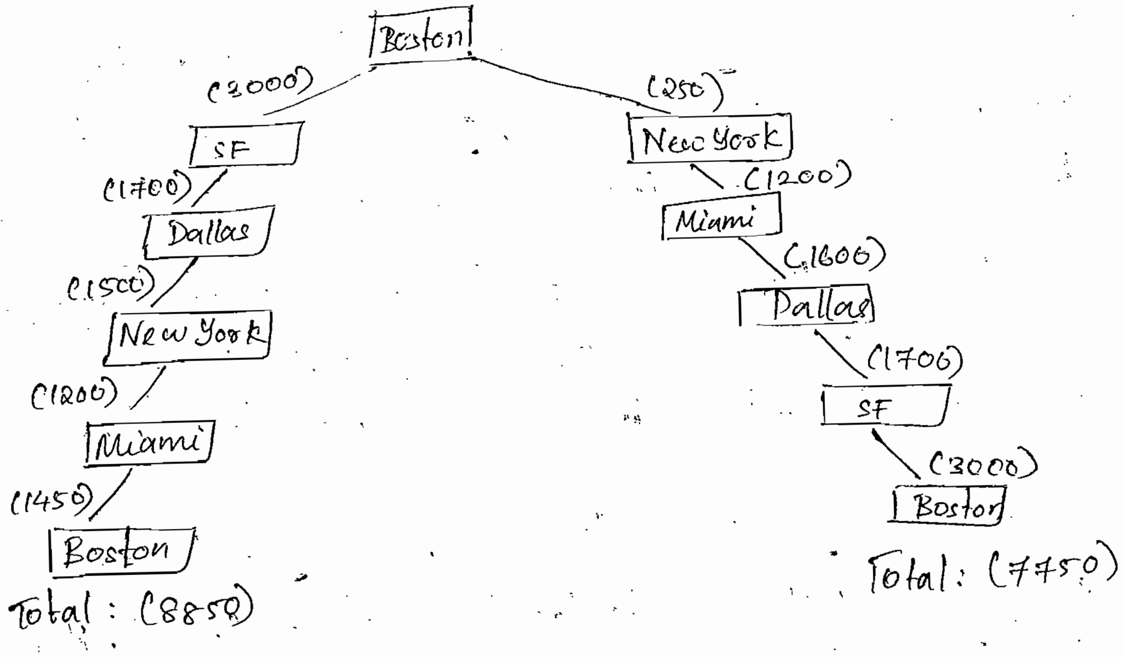
	Boston	NewYork	Miami	Dallas	SF
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
SF	3000	2900	3300	1700.	

One place salesman could start is Boston.

Boston $\xrightarrow{3000}$ San Francisco $\xrightarrow{1700}$ Dallas $\xrightarrow{1500}$ NewYork $\xrightarrow{1200}$

Miami $\xrightarrow{1450}$ Boston. Total: (8850)

We cannot fix the above path as the solution to the pblm. Other paths has to be explored inorder to so fix the solution path.



5. Is the solution a state or a Path?

→ Consider the problem of finding a consistent interpretor for the sentence " The Bank president ate a dish of pasta salad with the fork".

- There are several components to this sentence, each of which, in isolation, may have more than one interpretation.

- Some of the sources of ambiguity in this sentence are the foll:

  - The word "bank" may refer either to a financial institution or to a state of a river. But only one of them may have a president.
  - The word "dish" is the object of the verb "eat". It is possible that a dish was not eaten. But it is more likely that the pasta salad in the dish was eaten.
  - Pasta salad is a salad containing pasta. But there are other ways meaning can be formed from pairs of nouns. eg: dog food does not normally contain dogs.

- Because of the interaction among the interpretations of the constituents of this sentence, some search may be required to find a complete interpretation for the sentence.

- Here to solve the problem of finding interpretation, we need to produce only the interpretation itself. No record of the processing by which interpretation was found is necessary.

→ But for the water Jug Problem, we need to obtain the path that will lead to the final state (2,0). Here obtaining the final state alone is not important.

- These two egs, natural language understanding & the water jug problem, illustrate the difference b/w problems whose solution is a state of the world and problems whose solution is a path to a state.

6. What is the Role of knowledge?

→ Consider the problem of playing chess. Suppose we had unlimited computing power available. How much knowledge would be required by a perfect pgm?

- The answer to this question is very little - just the rules for determining legal moves and some simple control mechanisms that implements an appropriate search procedure.

Additional knowledge about good strategy & tactics can help to constrain the search & speed up execution of the program.

→ Now consider the problem of scanning daily newspaper to decide which are supporting the democrats & which are supporting the republicans in the upcoming election. Again assuming unlimited computing power, how much knowledge is required by a computer trying to solve this problem?

A great deal of knowledge is required. It would have to know such things as.

- The name of candidates in each party.
- The fact that if major thing you want to see done is having taxes lowered, you are probably supporting the Republicans.
- The fact that if major thing you want to see done is improved education for minority students, you are probably supporting democrats.
- The fact that if you are opposed to big govt, you are probably supporting republicans.

- These 2 problems, chess & newspaper understanding, illustrate the difference b/w problems for which a lot of knowledge is imp to constrain the search for a sol^n & those for which a lot of knowledge is required even to be able to recognize a sol^n.

7. Does the task require interaction with a person?

- Sometimes it is useful to program computers to solve problems in ways that the majority of people would not be able to understand.
  - This is fine if the level of interaction b/w the computer & its human users is problem-in, solution out.

- But increasingly we are building pgms that require intermediate interaction with people, both to provide additional input to the pgm and to provide additional reassurance to the user

- eg: Consider the problem of proving mathematical theorms. If

  a) All we want is to know there is a proof.

  b) The pgm is capable of finding a proof by itself.

  then it does not matter what strategy the pgm takes to find the proof.

  But however if we are trying to prove some new difficult theorm. We might demand a proof that follows traditional patterns so that a mathematician can read the proof & check to make sure it is correct.

- As computers move into areas of greater significance to human lives, such as medical diagnosis, people will be unwilling to accept a pgm whose reasoning they cannot follow.

- Thus there are 2 types of problems:

  a) Solitary, in which a computer is given a problem description and produces an answer with no intermediate comm^n and with no demand for an explaination of the reasoning process.

  b) Conversational, in which there is intermediate comm^n b/w person and computer, either to provide additional assistance to the computer or to provide additional information to the user, or both.

Searching Strategies

There are 2 types of search strategies

↳ Uninformed / Blind Search

↳ Informed Search / Heuristic Search.

- In Uninformed Search, no additional information about the states beyond that are provided in the problem definition.
- All they can do is generate successors and distinguish a goal state from a non goal state.
- strategies that know whether one non goal state is more promising than another are called informed search.

Uninformed / Blind Search Strategies

- While searching there is no clue whether one non goal state is better than the other. The search is blind.
- Here while generating new states, the states are compared with the goal state. If it is goal state, then the search is stopped. Otherwise the search process is continued.
- Various blind strategies are :

1. Breadth-First Search ✓
2. Uniform-Cost Search
3. Depth First Search ✓
4. Depth limited Search
5. Iterative Deepening Search
6. Bidirectional Search.

(Ticked ones included in syllabus)

1. Breadth First Search

- BFS is a simple strategy in which the root node is expanded 1st, 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 level in the search tree before any node at the next level are expanded.

→ Algorithm : (Same as one discussed before)

- BFS can be evaluated using 4 criteria.

- It is complete as it guarentees to find a solution if one exists.
- It is optimal if the path cost for all action is the same.
- To consider time & space complexity

Time Complexity : How long does it take to find a solution. (no. of nodes generated / expanded)

Space Complexity : No. of nodes stored in memory during search.

They are measured in terms of :

b - Maximum branching factor of search tree.

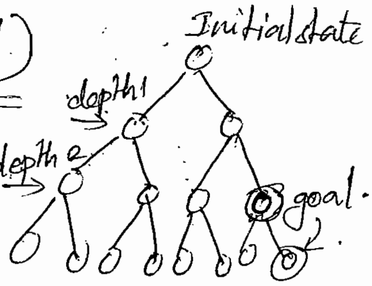
d - Depth of least cost solution.

m - Maximum depth of state space.

Assume a state space where each node has or state has b successors. Root node has b successors, each node at the next level has again b successors (total b^2)

Assume solution is at depth d. We would expand all nodes except the last one at depth d. Thus total number of nodes generated is

$b + b^2 + b^3 + \dots + b^d + (b^{d+1} + \cancel{b}) = \underline{O(b^{d+1})}$



Here each node is retained in memory. So space complexity is same as time complexity ie $O(b^{d+1})$

→ Adv & Disadv : (Discussed before)

2. Depth First Search

- DFS is a control strategy which always expands the deepest node.
- A single branch of the tree is pursued until it yields a solution or until a decision to terminate the path is made. In DFS, the search is conducted from the start state as far as it can be carried until either a dead-end or the goal is reach. If a dead end is reached, backtracking occurs.
- Algorithm (same as before)

Completeness : Does it always find a solution if one exists?
No, unless search space is finite & no loops are possible.

Time complexity : $O(b^m)$

Space complexity : $O(bm)$

Informed / Heuristic Search

- Hill climbing
- Best First Search
- A* Search
- Constraint satisfaction

- Uses problem specific knowledge beyond the definition of the problem itself. It can find solutions more efficiently than an uninformed strategy.
- Heuristic search is best at finding a solution that is good enough rather than the perfect solution.
- To do a heuristic search, an evaluation function is required that helps to rank the option.

Generate - And - Test

- Generate - And - Test strategy is the simplest of all approaches. It guarantees to find a solution if done systematically & if there exists a solution.

Algorithm

1. Generate a possible Solution (or generate a path from a start state.
2. Test to see if this is the expected solution by comparing it with the set of acceptable goal states.
3. If the solution has been found quit else goto step 1.

- Potential solutions that need to be generated vary depending on the kind of problem. For some problems, the possible solutions may be particular points in the problem space & for some problems, paths from start state.

- The most straightforward way to implement systematic generate-and-test is a depth first search tree with backtracking.

- A heuristic is needed to sharpen up the search.

Consider the problem of four 6-sided cubes, and each side of the cube is painted in one of four colors. The four cubes are placed next to one another & the problem lies in arranging them so that the 4 available colors are displayed in whichever way the cubes are viewed.

The problem can only be solved if there are atleast 4 sides coloured in each color & the number of options tested can be reduced using heuristics if the most popular color is hid~~dent~~ by the adjacent cube.

Hill climbing

- It is a variant of generate and test in which feedback from the test procedure is used to help the generator decide which direction to move in the search space.

- In a pure generate & test procedure, the test function responds with only a yes or no.

- But in hill climbing, the test function is a heuristic function that provides an estimate of how close a given state is to a goal state.

- This is good as computation of the heuristic function can be done at almost no cost at the same time the test for a solution is performed.

- Hill Climbing is often used when a good heuristic function is available for evaluating states but when no other useful knowledge is available.

- Hill Climbing is an optimization technique which belongs to the family of Local Search (optimization)

- Hill Climbing attempts to maximize (or minimize) a function, $f(x)$, where x are discrete states. These states are typically represented by vertices in a graph, where edges in the graph encode nearness or similarity of a graph.

- Hill climbing will follow the graph from vertex to vertex, always locally increasing (or decreasing) the value of f, until a local maximum x_m is reached.

① Simple Hill Climbing

- In hill climbing, the basic idea is to always head towards a state which is better than the current one.

eg: If a person is at town A and can get to town B & town C (target is town D), which move the person should make. He can make a move to town B or C if it appears nearer to town D than town A does.

Algorithm :

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state

as the current state.

2. Loop until a solution is found or until there are no new operators left to be applied to the current state.

   a) select an operator that has not yet been applied to the current state and apply it to produce a new state.

   b) Evaluate the new state:

      i) If it is a goal state, then return it & quit.

      ii) If it is not a goal state, but it is better than the current state, then make it the current state.

      iii) If it is not ~~the~~ better than the current state, then continue in the loop.

- Here the evaluation function is used to inject task specific knowledge into the control process. For algorithm to work, a precise definition of better must be provided. In some cases it means a higher value of heuristic function. In others it means a lower value.

- From any given node, we go to the next node that is better than current node. If there is no node better than ~~than~~ the current node, then hill climbing halts. (but no going backward)

eg: 4 Queen Problem

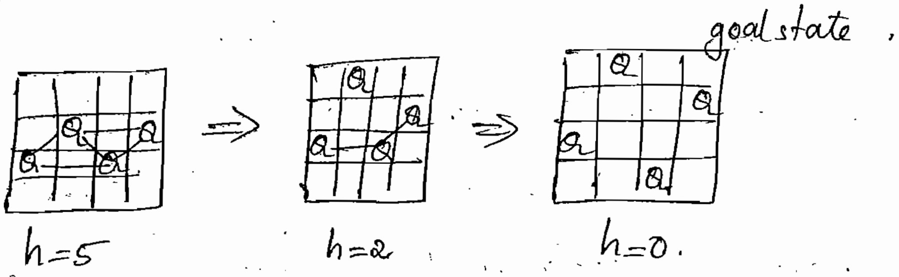
States : 4 Queens in 4 Columns.

Initial state : A blank configuration (4×4)

Neighbourhood operators : move Queen in column such that there is no attack.

Evaluation/↓function : h(n) = no of attacks/conflicts.
optimization

goal ~~sta~~ test : no attacks ie $h(G) = 0$ ·



→ so here in the above problem, we trig to minimize h.

② Steepest-Ascent Hill Climbing or Gradient Search

- A variation on simple hill climbing ~~that~~ ie it considers all the moves from the current state and selects the best one as the next state.

- ~~It~~ Following the steepest path can lead to goal faster but ~~then~~ there is no guarentee that it will find the goal.

Algorithm :

1. Evaluate the initial state. If it is also a goal state, then return it and quit. Otherwise continue with initial state as current state.

2. Loop until a solution is found or until a complete iteration produces no change to current state.

   a) let succ be a state such that any possible successor of the current state will be better than succ.

   b) For each operator that applies to the current state do:

      i) Apply the operator & generate a new state.

      ii) Evaluate the new state. If it is a goal state, then return it and quit. If it is not a goal state, compare it to succ. If it is better, then set succ to this state.

If it is not better, leave succ alone.

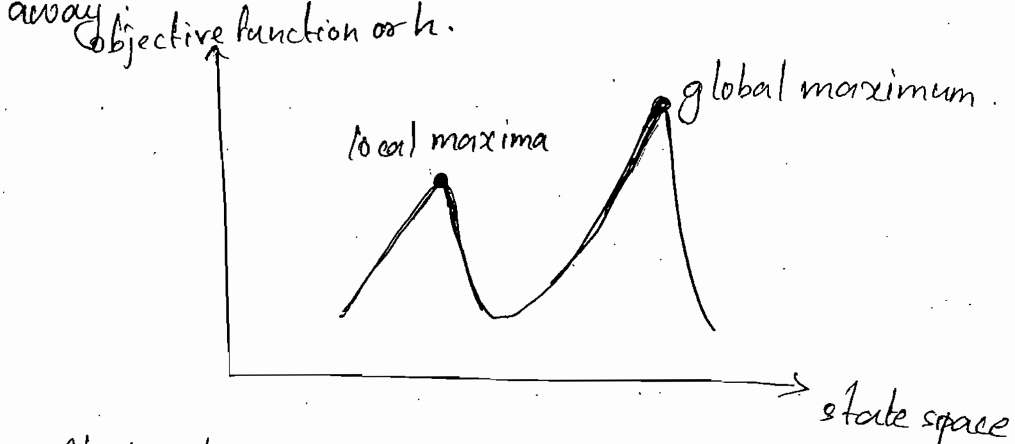
c) If succ is better than current state, then set current state to succ.

- Both simple and steepest-ascent hill climbing may fail to find a solution.

- The Problems or Issues in Hill Climbing are

ⓐ Local maximum

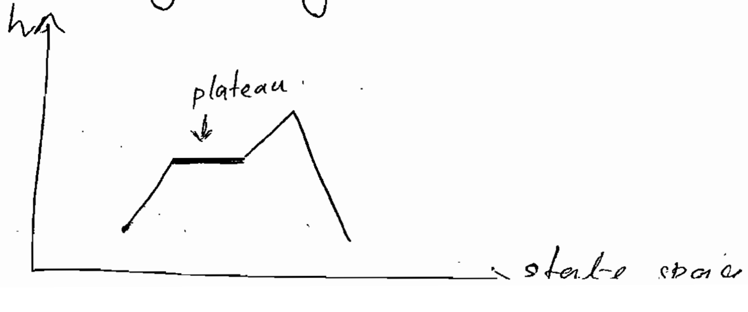
- A local maximum is a state that is better than all its neighbors but is not better than some other states farther away.



- At local maximum, all moves appear to make things worse.
- Can be overcome by random walks and simulated annealing or by iterating hill climbing algorithm.

ⓑ Plateau

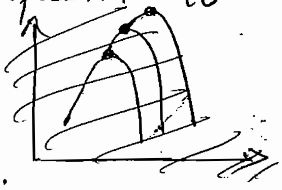
- A plateau is a flat area of the search space in which a whole set of neighboring states have the same value.



- on a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

© Ridges

- A ridge is a curve in the search space that leads to a maximum ie it is an area of the search space that is higher than surrounding areas and that itself has a slope. But the orientation of the ridge, compared to the available moves and directions in which they move, make it impossible to traverse a ridge by single moves.



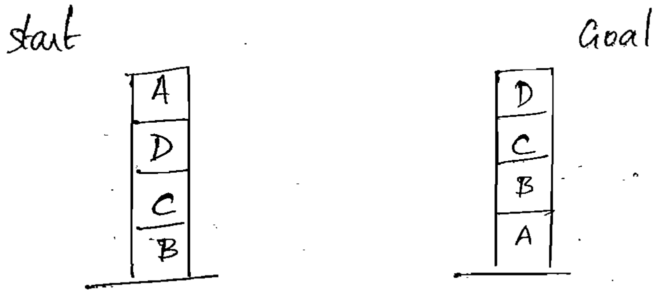
Some ways to deal with these problems are:

- Backtrack to some earlier node and try going in a different direction. To implement this strategy, maintain a list of paths almost taken and go back to one of them if the path ~~take~~ that was taken leads to a dead end.
  - A good way to deal with local maxima situation.

- Make a big jump in some direction to try to get to a new section of the search space.
  - A good way to deal with plateaus.
  - If the only rules available describe single small steps, apply them several times in the same direction.

- Apply 2 or more rules before doing the test: This corresponds to moving in several directions at once.
  - A good strategy to deal with ridges.
  - Simulated Annealing is also an

eg: consider the Block World Problem.



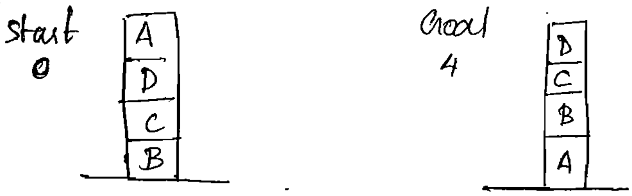
Assume the operators:

1. Pick up one block & put it on the table.
2. Pick up one block and put it on another one.

- Suppose we use the foll heuristic fn.

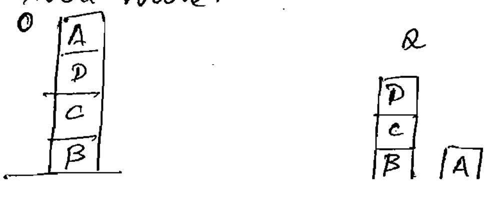
Local Heuristic:

1. Add 1 point for every block that is resting on the thing it is supposed to be resting on.
2. Subtract 1 point for every block that is resting on a wrong thing.

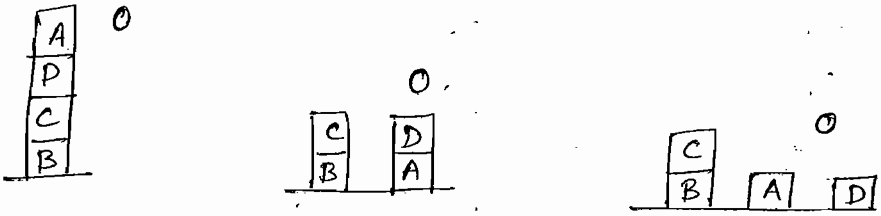


Using this fn, goal has a score of 4. The initial state has a score of 0 (since it gets 1 point added for blocks C & D and 1 point subtracted for blocks A & B).

There is only 1 move from initial state, namely to move block A to the table. That produces a state with score 2. Hill climbing will accept that move.



From the new state, there are 3 possible moves, leading to the 3 states shown below.



Hill climbing will halt because all these states have lower scores than the current state. This process reached a local maximum that is not the global maximum.

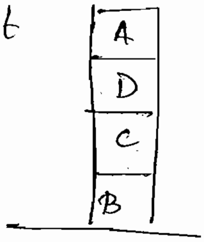
- Suppose we try the following heuristic function in place of first one:

Global Heuristic:

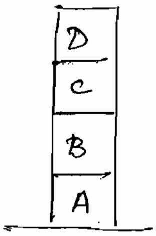
1. For each block that has the correct support structure (ie complete structure underneath, it is exactly as it should be), add 1 point to every block in the support structure

2. For each block that has a wrong support structure, subtract 1 point for every block in the existing support structure.

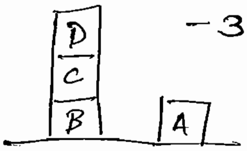
Start
-6



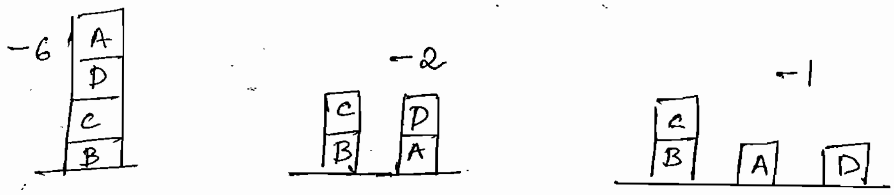
Goal
6



Using this heuristic fn, goal state has the score 6 (1 for B, 2 for C, 3 for D). Initial state has score -6. Moving A to table yields a state with score -3 since A has no longer 3 wrong blocks under it.



3 new states can be generated next having the scores -6, -2 and -1



This new heuristic fn captures the 2 key aspects of this problem:
incorrect structures are bad and should be taken apart and correct structures are good and should be built up.

③ Simulated Annealing

- The problem of local maxima is overcome using simulated annealing search.
- The normal hill climbing algorithm never makes movement towards downhill. such alg get stuck on a local maximum.
- On the other hand, a purely random walk - ie moving to a successor chosen uniformly at random from the set of successors - is complete but extremely inefficient.
- Simulated Annealing is a variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.
- The idea is to do enough exploration of the whole space early so that the final solution is relatively insensitive to the starting state. This will lower the chances of getting stuck at local maxima, plateau or ridge.
- In metallurgy, annealing is the process used to harden metals and glass by heating them to a high temp & then gradually cooling them. thus allowing the material to

coalesce into a low energy crystalline state.

- Here a state is accepted probabilistically.
  Suppose the probability $p' = e^{-\Delta E/T}$ is of new state.
  A random number is generated from a uniform distribution $[0,1]$ (ie values b/w 0 & 1)
  If the number generated is less than p', then the new state with probability p' is accepted. Otherwise search is continued from the old state itself.

Algorithm

1. Evaluate the initial state. If it is a goal state, then return it and quit. Otherwise continue with initial state as current state.

2. Initialize BEST-SO-FAR to the current state.

3. Initialize T according to the annealing schedule.
   [The rate at which the system is cooled is called the annealing schedule]

4. Loop until a solution is found or ~~thee~~ until there are no new operators left to be applied in the current state.

   a) Select an operator that has not yet been applied to the current state & apply it to produce a new state

   b) Evaluate the new state. Compute
      $$\Delta E = (\text{Value of Current}) - (\text{Value of new state})$$

$\boxed{e^{-2}}$ e^{-3} greater $e^2 < e^3$

      i) If the new state is a goal state, then return it and quit.

      ii) If it is not a goal state but it is better than the current state, then make it the current state. Also set BEST-SO-FAR to this new state.

iii) If it is not better than the current state, then make it to the current state with probability p'. ~~The step is~~ ie $p' = e^{-\Delta E/T}$.

The step is usually implemented by invoking a random number generator to produce a no. in the range [0,1]. If that no. is less than p', then the move is accepted. Otherwise, do nothing.

c) Revise T as necessary according to annealing schedule.

- To implement the above algorithm, it is necessary to select an annealing schedule, which has 3 components

i) The initial value to be used for temperature.
[eg: T could be initialized to a value such that, for an average ΔE, p' would be 0.5]

ii) The criteria that will be used to decide when the temperature of the system should be reduced.
(As T approaches 0, the prob. of accepting a move to a worse state goes to 0 & simulated annealing becomes identical to hill climbing)

iii) Amount by which temp will be reduced each time it is changed.

A* Algorithm

AI Module - 1

I/p: an implicit search graph problem with cost on the arcs.

O/p: the minimal cost path from start to goal node.

1. Put the start node, s on OPEN.
2. If open is empty, exit with failure.
3. Remove from OPEN & place on CLOSED a node n having minimum f.
4. If n is a goal node exit successfully with a solution path obtained by tracing back pointers from n to s.
5. Otherwise, expand n generating its children & directing pointers from each child node to n.
   - For every child node n' do
     - evaluate h(n') and compute ~~f(n')~~ $f(n') = g(n') + h(n') = g(n) + c(n,n') + h(n')$
     - If n' is already on OPEN or CLOSED compare its new f with the old f and attach the lowest f to n'
     - put n' with its f value in the right order in OPEN

Goto step 2.

incl. sentence
and connectives
assignments to
the whether pro
is true
gives ordered
connectives are
onal logic, bu
variables for i
quantifiers, symb

Python Programming Language

- Python is an example for high level language.
- Python is considered as interpreted language as python program are executed by an intepreter (Line by Line execution).
- There are two ways to use the interpreter:
  - Interactive Mode
  - Script Mode.
- In Interactive mode, you type python programs at the terminal and interpreter prints the result.

eg:

```
>>> 1 + 2
3
```

The chevron, >>> is the prompt the interpreter uses to indicate that it is ready. If 1+2 is typed, the interpreter replies 2.

- In Script Mode, code is stored in a file and interpreter is used to execute the contents of the file.
  - Python scripts have names that end with .py

To execute the script, the name of the file has to be provided. In a UNIX command window, type python eg.py, where eg.py is the name of the file

- Python can only execute a program if the syntax is correct; Otherwise the interpreter displays an error message.

eg: (1+2) is legal, but 8) is a syntax error. Syntax refers to the structure of a program and the rules about that struc

Program to display messages

- 'print' is the keyword used to display messages on the scree

```
eg: >>>print 'Hello world'
```

- The quotation marks in the program mark the beginning & enc of text to be displayed. They don't appear in the result

Values & types

- A value is one of the basic things a program work with, like a letter or a number.
- The values belong to different types.
    - int (representing integer values)
    - float (representing floating point values)
    - str (representing string)
    - bool (representing boolean values)
- Interpreter returns the type of the value when 'type' ~~keyw~~ is used along with the value.

```
>>>type('Hello World')
<type 'str'>

>>>type(17)
 <type 'int'>

>>>type(3:2)
 <type 'float'>
```

```
>>>type('3.2')
 <type 'str'>

>>>type(True)
<type 'bool'>
```

Variables

- A variable is a name that refers to a value.
- An assignment statement creates new variables & gives them values.

eg:
```
>>>message = 'Hello World'
>>> n = 17
>>> pi = 3.14
```

- To display the value of a variable, use print statement.

eg:
```
>>>print n
17
```

- The type of a variable is the type of the value it refers to

eg:
```
>>>type(message)
 <type 'str'>
```

Variable Names ~~& Keywords~~

- Choose names for the variable in such a way that they are meaningful.
- Variable names can be arbitrarily long. They can contain both letters and numbers, but they have to begin with a letter.
- Can use uppercase letters, but better to begin variable names with a lower case letter.
- Underscore character (_) can appear in a name.
- A syntax error occurs if a variable has an illegal name.

Keywords

- Interpreter uses keyword to recognize the structure of the program.

- Python has 31 keywords.

and
as
assert
break
class
continue
def

del
elif
else
except
exec
finally
for

from
global
if
import
in
is
lambda

not
or
pass
print
raise
return
try

while
with
yield

Statements

- A unit of code python interpreter can execute.

A script usually contains a sequence of statements. If there are more than one statement, the results appear one at a time as the stmt execute.

eg: the script

```
print 1
x=2
print x
```

produces the o/p.

```
1
2
```

Operators and Operands

Operators +, -, *, / and ** perform addition, subtraction, multiplication &, division & exponentiation.

eg: 20+32 , hour-1 , hour*60+minute minute/6 5**2 (5+9)*(15-7) these all are valid cases. (provided variables are assigned values before using them)

- Modulus operator is used to get [illegible] when 1st operand is

Variables

divided by the second. (10)

eg: >>> rem = 7%3 -
print rem.
1

eg >>>minute = 59
>>>minute/60
0 → Reason for o/p. is python performs floor division (as both operands are integers)

Expressions

An expression is a combination of values, variables & operators
eg: following are valid expressions.

```
17
x
x+17
```

Order of Operations (PEMDAS)

- Parenthesis → Exponentiation → Multiplication or Division → Addition or Subtraction.
- Operators of same precedence are evaluated from left to right.

→ Mathematical operations cannot be performed on strings.

eg: '1' - 'o' is invalid.

→ + operator works with strings, it performs concatenation

eg:
```
>>> first = 'hello'
>>> second = 'world'
>>>print first+second
helloworld
```

→ * operator also works on strings, it performs repetition.

eg: >>>'spam'*3 gives o/p SpamSpamSpam.

⇒ Comments can be added to the program. They are includ
starting with the symbol #.

eg: #compute perc. of hr that has elapsed

```
>> perc = (minute * 100)/60.
```

or

```
perc = (minute * 100)/60   # perc. of an hr.
```

Functions

- Function is a named sequence of statements that performs a computation.
- A function is defined by specifying a name & including a set of statements. Function can be later called using its name.

```
>>> type(32)
 <type 'int'>
```

- The name of the function is type. Result returned is called Return value.

Type Conversion Function

- Python provides built in function that convert values from one ty to another.

eg1: int function takes any value (other than string) & converts it to an integer.

```
>>>int('32')     or    >>>int(-2.3)
 32                      -2.
```

eg2:

```
>>>float(32)
 32.0
```

str function converts its argument to a string.

```
>>>str(20) ⟶ will give o/p '20'
```

Math Functions

- Python has a math module that provides most of the familiar mathematical functions.
- Before we use math function, we have to import math module.

```
>>> import math   . # stmt creates module obj called math.
```

- Information abt math module is obtained if we print it

```
>>> print math.
```

- To access one of the function, specify the name of the modul and the name of the function seperated by a dot.

eg:
```
>>> r = 0.7
>>> height = math.sin(r)        or math.sqrt(2)
```

Adding new Functions

- A Function definition specifies name of a new function & a seque of statements that execute when the function is called.

eg:
```
def new_fn():         #no arguments
    print "Content1"
    print " Content2"
```

- def is a keyword that indicates that this is a function definitio Name of the fn is new_fn. Rules for naming fns is the same as those of variables.
- 1st line of fn definition is called header; the rest is called body. Header ends with a colon and the body has to be indented. Indention is 4 spaces.

- To end the function, enter an empty line.

```
>>>type(new_fn)
 <type 'function'>
```

- syntax for Function call

```
>>>new_fn()
```

- Can use one function within another. ie Nesting of fns is allowed.

```
def repeat_lyrics():
    new_fn()
    new_fn()
```

- Function definitions do not alter the flow of execution of the program. But however stmts within the function are not executed until the function is called.

Parameters & Arguments

- Arguments are assigned to variables called parameters.
- eg for a user defined fn that takes an argument.

```
def print_twice(bruce):
    print bruce
    print bruce
```

- when the fn print_twice is called

```
>>>print_twice('spam')
```

~~the stm~~ the argument within fn call will be assigned to the parameter bruce. This will give the foll o/p

```
Spam
Spam
```

```
>>> print_twice (17).
    17
    17.   } o/p.
```

- when a variable is created inside a function, it is <u>local</u> which means that it only exists inside the function:

```
>>>def new_twice(part1, part2)
       newvar = part1+part2   #newvar is a local variable
       print_twice (newvar)
```

Suppose

```
>>> line1 = "String1"
>>> line2 = "Text2"

>>> new_twice (line1, line2)
```

o/p will be

```
String1Text2
String1Text2
```

Fns that return boolean values => Boolean fns

```
eg >>>def is_div(x,y)
          if x%y==0
             return True
          else:
             return False.

>>> is_div(6,2)
True.        # o/p.
```

- Functions that return some value are called <u>Fruitful Functions</u> & the fns that don't return a value are called <u>void functions</u>

```
eg:  x = math.sin(2)
     print x.
```

- In interactive mode, when a function is called, Python displays the result.

```
>>>math.sqrt(5)
2.2360679
```

- But in a script, if a fruitful function is called by itself, the return value is lost forever.

- Python has a built in function called len which returns the length of the string passed.

  eg: len('allen') is 5

- To print more than one value on a line, you can print a comma-seperated sequence.

  print '+', '-'

- If the sequence ends with a comma, Python leaves the line unfinished, so the value printed next appears on th same line.

```
print '+',
print '-'
```

  The o/p of these statements is '+ -'

- Docstring is a string at the beginning of a function tha explains the interface.

  "doc" is short for documentation.

- Docstring is a triple-quoted string, also known as multili string because the triple quotes allow the string to span mo than one line.

eg:
```
""" Draws two segments with the given length &
    angle (in degrees) b/w them.
"""
```

- Logical Operators

  3 logical operators : and, or, and not.

eg: $x>0$ and $x<10$ is true only if x is greater than 0 a less than 10.

```
>>> 17 and True
    True.
```

Conditional Execution

→ Program executes based on the condition given.

```
if x>0:
   print 'x is positive'
```

- Boolean expression after the if statement is called the condition If it is true, then the indented statement gets executed. If not nothing happens.
- if statements have the same structure as function definitions: a header followed by an indented block. Statements like this are called compound stmts.

→ A second alternate form of if statement is alternative execution, in which there are 2 possibilities and the condition determines which one to execute.

eg:

```
if x%2==0:
     print 'x is even'
else:
     print 'x is odd'
```

→ Chained Conditionals

If we need more than 2 branches.

```
if x<y:
   print 'x is less than y'
elif x>y:
   print 'x is greater than y'
else:
```

→ Nested Conditionals

One condition can also be nested within another.

```
if x==y:
    print 'x and y are equal'
else:
    if x<y:
        print 'x is less than y'
    else:
        print 'x is greater than y'
```

Recursion

It is legal for a function to call itself. This is known a Recursion.

eg:
```
def countdown(n):
    if n<=0:
        print 'Blastoff'
    else:
        print n
        countdown(n-1)
```

```
>>> countdown(3)
3
2
1
Blastoff.
```

Looping Stmt

```
for i in range(n):
    print 'hello'
```

Pgm to find factorial of a no. using recurs

```
>>>def factorial(n):
        if n==0:
            return 1
        else:
            return n*factor

>>>factorial(5)
120
```

→ Nested conditionals

One condition can also be nested within another.

```
if x==y:
   print 'x and y are equal'
else:
    if x<y:
        print 'x is less than y'
    else:
        print 'x is greater than y'
```

Recursion

It is legal for a function to call itself. This is known as Recursion.

eg:
```
def countdown(n):
    if n<=0:
        print 'Blastoff'
    else:
        print n
        countdown(n-1)
```

```
>>> countdown(3)
3
2
1
Blastoff.
```

Looping Stmt

```
for i in range(n):
    print 'hello'
```

Pgm to find factorial of a no. using recursion

```
>>>def factorial(n):
       if n==0:
          return 1
       else:
          return n*factoria
```

```
>>>factorial(5)
120
```

Keyboard Input

- Python provides built in function called raw_input that gets input from the user (ie from keyboard)
- When this function is called, the program stops & waits for the user to type something. Program is resumed when user presses Return or Enter.

eg:
```
>>>input=raw_input()
Sample text
>>> print input
Sample text
```

- Before getting input from the user, it is a good idea to print a prompt telling the user to input. raw_input can take a prompt as an argument.

```
>>> name = raw_input('Enter ur name?\n')   # \n represents newline
Enter ur name?
Cini
>>>print name
Cini
```

Iteration Stmts → stmt used to repeat a set of code

→ for statement

```
>>> for i in range(3)
        print "hello"
```

→ while stmt

```
>>> while n>0:
        print n
```

o/p for n= 3

```
3
2
1
```

Write program to generate 1st 10 even nos.

```
>>>def fn_even(l):
      i=1
      while i<=l:
          if i%2==0:
              print i
          i=i+1
>>>fn_even(10)
```

Strings

- A string is a sequence of characters. You can access the characte one at a time with the bracket operator:

```
>>>fruit = 'banana'
>>>letter = fruit[1]
                 ↳ index.     # index starts from 0.
```

- string indexes must be integer.

len

built in function that number of characters in a string.

```
>>> len(fruit)
    6
```

Traversal with loop

```
index=0
while index<len(fruit):
    letter=fruit[index]
    print letter
```

} program to print each character of a string.

OR

```
for i in fruit:
    print i
```

o/p:

```
'b'
'a'
'n'
'a'
'n'
'a'
```

```
prefix = 'JKLM'
suffix = 'ack'
for letter in prefix:
    print letter+suffix:
```

⟹ o/p

```
Jack
Kack
Lack
Mack .
```

String slice

- A segment of a string is called slice. Selecting a slice is similar to selecting a character

```
>>>s= 'Monty Python'
>>>print s[0:5]
Monty
```

- Operator [n:m] returns the part of the string from the nth character to the mth character, including n but excluding m
- If 1st index is omitted, the slice starts at the beginning of the string. If 2nd index is omitted, slice goes to the end of the string.

```
>>>fruit='banana'
>>> fruit[:3]
   'ban'
>>>fruit[3:]
   'ana
```

~~while i in string~~

```
string1 = 'Jack'
string2= 'abc'
    Jabc
    aabc
    cabc
    kabc
```

```
for i in
 range(len
   print
```

- Strings are immutable

eg : >>>greeting = 'Hello Pgm'
>>>greeting [0] = 'J'

} will return error saying object does not support item assignment.

```
>>> new_greeting= 'J' + greeting[1:]
>>> print new_greeting
 Jello Pgm
```

Q) Pgm to search for a letter in string.. Return index if success otherwise -1.

```
def find(word, letter):
    index=0
    while index< len(word):
        if word[index] = = letter:
            return index
        index = index+1
    return -1
```

Q) Pgm to count the no. of times the letter appears in a string.

```
word = 'banana'
count=0
for letter in word:
    if letter=='a':
        count = count+1
print count.
```

in Operator

- in is a boolean operator that takes 2 strings & returns True if the first appears as a substring in the second.

eg: >>>'a' in 'banana'
True.
>>>'seed' in 'banana'
False.

Q) Pgm to print all letters that appear in both strings.

```
def in_both(word1,word2):
    for letter in word1:
        if letter in word2
            print letter.
```

— String Methods

upper(), find()

string1):
string1[i]
+string2

```
eg1: >>> word = 'banana'
     >>> new_word = word.upper()
     >>> print new_word
         BANANA
```

```
eg2 >>>word = 'banana'
    >>>index = word.find('a')
    >>> print index
    1    3
```

→ finds first occurence of the letter 'a' beginning from 0.

```
>>> word.find ('na', 3)
```

→ indicates start a sequence from index 3.

```
>>> word.find('a', 1, 2)
```

start pos. for search

end index. for search

String Comparison :

- Comparison operator works on strings.
- To check equality of 2 strings

```
if word == 'banana'
    print 'Strings equal'
```

- Other comparison operators are useful in putting words in alphabetical order.

```
if word < 'banana':
    print 'Word' + word + ' comes before banana'
elif word > 'banana':
    print 'Word' + word + ' comes after banana'
else:
    print 'Both are equal'
```

- Python does not handle uppercase & lowercase letters in the same way. All uppercase letters comes before all the lowercase letters.

Q) Pgm to compare 2 strings & check whether 1 string is the reverse of the other. Return true in this case. Otherwise false.

```
def is_reverse (word1, word2):
    if len(word1)!=len(word2)
        return False
    i=0
    j= len(word2)
    j=j-1
    while j>=0:
        if word1[i]! = word2[j]:
            return False
        i= i+1
        j=j-1
    return True
```

Lists

- List is a sequence of values. In a string, values are characters. In a list they can be any type.
- Values in a list are called elements.
- There are several ways to create a list.

Enclose the elements within square bracket '[' and ']'

eg: [10,20,30,40] # It is a list of 4 integers.

['hello', 'world', 'abc'] # List of 3 strings.

- A list can be nested within another list.

['spam', 2.0, 5, [10,20]]

- List that contains no elements is called an empty list
- Can assign list values to variables.

```
>>> listname = ['abc', 'def']
>>> nos = [17,123]
>>> empty =[]
>>> print listname, nos, empty
['abc', 'def'] [17,123] []
```

- Lists are mutable

The elements of a list can be accessed in the same way as that of accessing characters in the string ie using 'bracket operator. Expression inside bracket specifies the index.

```
>>> print nos [0]
17.
```

It is possible change the values stored in the list. (Done using assignment operator)

eg:
```
>>> nos [1] = 5
print nos
[17, 5]
```

- List indices work in the same way as string indices. If an index has a negative value, it counts backward from the end of the list.

- in operator also work on lists

```
>>> msg= ['abc', 'def', 'ghi']
>>> 'def' in msg
True
```

Traversing a List

-> Most common way to traverse the elements of a list is for loop.

Syntax :

```
>>> for var in msg:
        print msg
```

~~op:~~

```
abc
def
ghi
```

-> If you want to write or update the elements of the list, you require indices.

```
>>> for i in range(len(nos)):
        nos[i] = nos[i]*2

>>> print nos
[34, 10]
```

List Operations

- '+' operator concatenates lists.

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> c = a+b
>>> print c
[1,2,3,4,5,6]
```

- '*' operator repeats a list ~~of n~~ a given number of time.

```
eg >>> [0] * 4
     [0,0,0,0]

>>> nos * 2             # if nos = [17, 5]
     [17,5,17,5]
```

List Slices

- Slice operator works on lists.

```
~~text~~>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
      ↓   ↳ last index (exclude this Index)
  1st index.
```

ie characters ~~stri~~ starting from 1st to 3rd index excluding 3rd index.

```
['b', 'c']
>>> t[:4]    # start from beginning.
['a', 'b', 'c', 'd']
>>> t[3:]    # start from 3rd index till last
['d', 'e', 'f']
>>> t[:]
['a', 'b', 'c', 'd', 'e', 'f']
```

Since lists are mutable

```
>>>t[1:3] = ['x', 'y']
print t
~~['a', 'b', 'c'~~
['a', 'x', 'y', 'd', 'e', 'f']
```

List Methods

⇒ Python provides methods that operate on lists.

→ append adds a new element to the end of a list.

```
>>>t= ['a', 'b', 'c']
 >>> t.append ('d')
 >>> print t
['a', 'b', 'c', 'd']
```

→ extend method takes a list as an argument and appends a of the elements.

```
 >>>t1 = ['a', 'b', 'c']
 >>> t2 = ['d', 'e']
 >>> t1.extend(t2)
 >>>print t1
 ['a', 'b', 'c', 'd', 'e']
```

→ sort method arranges the element of the list from low t high.

```
>>> t= ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>>print t
 ['a', 'b', 'c', 'd', 'e']
```

→ List methods are all void. They modify the list and return None

⇒ There are several ~~methods~~ ways to delete elements from a list

pop ( ) can be used if we know the index of the elemen to be deleted.

```
>>>t= ['a', 'b', 'c']
>>> x = t.pop(1)
 >>> print t
['a', 'c']
```

```
>>> print x
b.
```

If index is not provided, it deletes & returns the last element.

```
>>> t.pop()
'c'
```

- If there is no need for the removed value, then del operato can be used.

```
t = ['a','b','c']
>>> del t[1]
>>> print t
['a', 'c']
```

- If you know the element to be removed (not index), can use remove

```
>>> t = ['a','b','c']
>>> t.remove('b')
>>> print t
['a', 'c']
```

- To remove more than one element, can use del with a slice index

```
>>> t = ['a','b', 'c','d','e','f']
>>> del t[1:5]
>>> print t
['a', 'f']
```

⇒ Python provides a built in function, sum that adds up all elements in the list.

```
>>> t = [1,2,3]
>>> sum(t)
```

An operation like this combines a sequence of elements into a single value. Hence it is known as reduce

⇒ Another common operation is to select some of the elements from a list and return a sublist.

Eg: the foll fn. takes a list of strings & returns a list that contains only the uppercase strings.

~~t = ['a', 'B', 'C', 'd']~~

```
def only-upper (t):
    res = [ ]
    for s in t
        if s. isupper( ):
            res. append (s)
    return res.

~~o~~only-upper (raw-input('Enter list\n'))
Enter list
['a', 'B', 'C', 'd']
['B', 'C']
```

- An operation like only-upper is called a filter as it selects some of the elements & filters out the other.

Lists & Strings

- ~~A stri~~ A list of characters is not the same as a string.
- To convert from a string to a list of characters, we can use li

eg:
```
>>> s = 'spam'
>>> t = list(s)          # list is a name of a built in function
>>> print t
['s', 'p', 'a', 'm']
```

- list function breaks a string into individual characters.

<u>split method</u>

- Can break a string into words.

eg:
```
>>> s = 'Sample Pgm for class'
>>> t = s.split()
>>> print t
 ['sample', 'Pgm', 'for', 'class']
```

- An optional argument called a <u>delimiter</u> specifies which charac
  to use as word boundaries.

eg :
```
>>>s = 'spam-spam-spam'
 >>> de = '-'
 >>>s.split(de)
['spam', 'spam', 'spam]
```

- <u>join</u> is the inverse of split : It takes a list of strings & concatenates the elements.

<u>join is a string method</u>, so invoke it on the delimiter & pass the list as a parameter.

```
>>>t= ['sample', 'Pgm', 'for', 'Class']
>>> delimiter = ' '
>>> delimiter.join(t)
'sample Pgm for class'  # join puts a space b/w words as
                          delimiter is a space.
```

<u>Objects & Values</u>

a= 'banana'

b= 'banana'

a & b refer to the same string

- To check whether 2 variables refer to the same object, is operator can be used

```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
True.
```

- Python only created one string object & both a & b refer to it
- But when we create two ~~identical~~ equivalent lists, 2 objects are created.

```
>>> a = [1,2,3]
>>> b = [1,2,3]
>>> a is b.
False
```

Aliasing

- If a refers to an object & if b is assigned to a, then both variables refer to the same object.

```
>>> a = [1,2,3]
>>> b = a
>>> b is a.
True.
```

- Association of a variable with an object is called a referen

Here there are 2 references to the same object.

- An object with more than one reference has more than one name, so we say that object is aliased
- If the aliased object is mutable, changes made with one alias affect the other.

```
>>> b[10] = 17
>>> print a
```

- Passing list to a function is an eg. for pass by reference. If the function modifies a list parameter, the caller sees the change.

eg: delete_head removes the first element from a list

```
def delete_head(t):
    del t[0]
...

>>> l = ['a','b','c']
>>> delete_head(l)
>>> print l
['b','c']
```

Dictionaries

- A dictionary is like a list, but more general. In a list, the indices have to be integers, in a dictionary they can be any type.

- Dictionary can be considered as a mapping b/w a set of indices (kee and a set of values. Each key maps to a value. The associati of a key and a value is called a key-value pair

- Function dict() creates a new dictionary with no items.

```
>>> egdict = dict()
>>> print egdict
{}
```

→ represents empty dictionary. To add items to the dictionary, we can use square brackets.

```
>>> egdict['one'] = 1
```

- The above line creates an item that maps from the key 'one' to the value 1.

```
>>>print egdict      # A key-value pair is obtained with a colo
{'one': 1}           # b/w them
```

- Can assign values to dictionary at the time of creation.

```
eg: >>>egdict = {'one': 1, 'two'; 2, 'three': 3}

 >>>print egdict
   {'one': 1, 'three': 3, 'two': 2}    # the order of the
                                         key-value is not the
                                         same.
```

- thus Order of items in a dictionary is unpredictable.
- Keys are used to look f up for values in the dictionar

>>> ~~eg~~ print egdict['two']

```
    2
```

- If key isn't in the dictionary, an exception is returned.

```
>>>print egdict['four']
keyError: 'fours'
```

- len function works on dictionaries, it returns the num. of key value pairs

```
>>>len(egdict)
   3
```

- in operator works on dictionaries, it tells whether something appears as a key in the dictionary.

```
>>>'one' in egdict
   True.
>>> 1 in egdict
   False.
```

- To see whether something appears as a value in the dictionary, we can use the method values which returns the values as a list. & then use the in operator.

```
>>> vals = egdict.values()
>>> 2 in vals
  True.
```

- Python uses a hashtable structure to store dictionaries. so search time is almost the same for any item.

- Dictionary as a Set of Counters

Suppose a string is given. Count the number of times each letter appears.

- We can create a dictionary with characters of string as the key and counters as corresponding values. The counter value will be incremented ~~if a key~~ for the respective key if it occurs again.

```
>>>def histogram(s):
        d=dict()
        for c in s:
            if c not in d:
                d[c]=1
            else:
                d[c]+=1
        return d
>>> histogram('brontobrbr')
{'b':3,'r':3,'o':2,'n':1,'t':1}
```

& → can be in any order.

Looping and Dictionaries

- for statement can be used to traverse the keys of the dictionary.

```
def print_hist(h):
    for c in h:
        print c, h[c]
```

O/p ~~took~~

```
>>> h = histogram('parrot')
>>> print_hist(h)
 a  1
 p  1
 r  2          # keys can be in any order.
 t  1
o   1
```

Reverse lookup

Given a dictionary d and a key k, it is easy to find the corresponding value v = d[k]. This operation is called lookup

Finding key given the value is known as Reverse lookup For this search has to be done.

Here is a function that takes a value & returns the first key that maps to that value.

```
def reverse_lookup(d, v):
    for k in d:
        if d[k] == v:
            return k
    raise ValueError
```

→ raise statement causes an exception. In this case it is

eg:
```
>>> h= histogram("parrot")
>>> k = reverse_lookup(h,2)
>>> print k
r.
```

Tuples

- A tuple is a sequence of values. The values can be any type, and they are indexed by integers. Similar to lists.

- The only difference with lists is that tuples are immutable.

-> Tuple is a comma-seperated list of values:

```
>>>t='a','b','c','d','e'
```

-> Tuples can also be enclosed in parenthesis

```
>>> t= ('a','b','c','d','e')
```

- To create a tuple with a single element, include the final comma

```
>>> t1 = ('a',)
>>> type(t1)
<type 'tuple'>
```

- without comma, python treats ('a') as a string in parentheses.

```
>>> t2 = ('a')
>>> type(t2)
<type 'str'>
```

-> Another way to create a tuple is the built.in function tuple. with no argument, it creates an empty tuple.

```
>>> t=tuple()
>>> print t
()
```

- If the argument is a sequence (string, list or tuple), the result is a tuple with the elements of the sequence.

```
>>> t = tuple('lupins')
>>> print t
('l', 'u', 'p', 'i', 'n', 's')
>>> t=tuple(l)      where l = [1,2,3,'a']
>>> print t
  ('1,2,3,'a')
```

- Most List operators also work on tuples. Bracket operator indexe an element.

```
>>> t = ('a', 'b', 'c')
>>> print t[0]
'a'
```

- slice operator selects a range of elements.

```
>>> print t[1:3]
    ('b', 'c').
```

- Since tuples are immutable, if we try to modify eleme of the tuple, we'll get an error

```
>>>t[0] = 'A'
TypeError.
```

- Can't modify the elements of a tuple, but can replace one tuple with another

```
>>> t = ('A',) + t[1:]          Suppose t = ('a', 'b', 'c',
 >>> print t
 ('A', 'b', 'c', 'd', 'e')
```

Tuple Assignment

- It is often useful to swap the values of 2 variables. (usually a temporary variable is used)

```
>>> temp=a
>>> a=b
>>> b= temp.
```

- Using tuple assignment, it becomes easy.

```
>>>a , b = b, a.          # Initially a=5 & b=2
print a, b
2  5
```

left side is a tuple of variables, the right side is a tuple of expressions. Each value is assigned to its respective va

- All expressions on the right side are evaluated before any of the assignments.

- Number of variables on the left and number of values on right have to be same

- Right side can be any kind of sequence (string, list or tupl

eg:
```
>>> addr = 'abc@python.org'
>>> uname, domain = addr.split('@')
>>> print uname, domain
abc python.org
```

Tuples as Return values

- Function can only return one value, but if the value is a tuple, the effect is same as returning multiple values.

```
>>> t = divmod(7,3)   # divmod is a built in fn that
>>> print t             returns quotient & remainder.
 (2,1)
```

or
```
>>> quot, rem = divmod(7,3)
>>> print quot, rem.
```

```
def min_max(t):
      return min(t), max(t)   } eg. of a fn. that returns a tupl
```

- max and min are built in fns that return the smallest & larg element of a sequence. The fn return a tuple of 2 values.

Variable-Length Argument Tuples

- Fns can take a variable number of arguments. A parameter name that begins with *, gathers arguments into a tuple.

eg: the foll fn takes any no. of arguments & prints them.

```
def printall(*args):
      print args.
```

```
>>> printall(1, 2.0, '3')
 (1, 2.0, '3')
```

scatter - If you have a sequence of values and you want to pass it to a function as multiple arguments, then * operator can be used.

eg: divmod is a fn that takes 2 ~~of~~ arguments.

```
>>> t= (7,3)
divmod (t) ⇒ will return error.
```

However we can scatter the tuple using * .

ie
```
>>> divmod(*t)
(2,1)
 ↓  ↓
quot rem.
```

List and Tuples

- zip is a built-in-function that takes two or more sequences & zips them into a list of tuples where each tuple contain one element from each sequence.

- eg:- Zipping of a string and a list.

```
>>> s = 'abc'
>>> t = [0,1,2]
>>> zip(s,t)
[('a', 0), ('b,1), ('c',2)]
```

- The result is a list of tuples where each tuple contains a character from the string and the corresponding element from the list.

- If the sequences are not of the same length, the result has the length of the shorter one.

```
>>> zip('Anne', 'Elk')
[('A','E'), ('n','l'), ('n','k')]
```

- Can use tuple assignment in a for loop to traverse a list of tuples:

```
t = [('a',0), ('b',1), ('c',2)]
for letter, number in t:
        print number, letter
```

```
o/p
0  a
1  b
2  c
```

Dictionaries & tuples

- Dictionaries have a method called ~~tuples~~ items that returns a list of tuples, where each tuple is a key-value pair.

```
>>> d = {'a' : 0 , 'b' : 1 , 'c' : 2 }
  >>> t = d.items()
   >>> print t
    [('a', 0) , ('c', 2) , ('b', 1)]    # As in the case of diction
                                          items are in no particular
                                          order.
```

- Also tuples can be used to initialize a new dictionary.

```
>>> t = [('a', 0) , ('c', 2) , ('b', 1)]
   >>> d = dict(t)
   >>> print d
   {'a' : 0 , 'c' : 2 , 'b' : 1}
```

- Combining dict with zip yields a ~~tw~~ concise way to create a dictionary.

```
>>> d = dict(zip('abc', range(3)))
>>> print d
{'a' : 0, 'c' : 2 , 'b' : 1}
```

-
```
>>> for key, val in d.items():     } used to traverse keys & valu
        print val, key                of a dictionary
```

Comparing tuples

- Comparison operators works with tuples & other sequences.
- Python starts by comparing 1st element from each sequence. If they are equal, it goes on to next element & so on until it find elements that differ. Remaining elements are not checked.

```
>>> (0, 1, 2) < (0, 3, 4) → Returns True
```