What is Artificial Intelligence?

- It is a branch of Computer Science that pursues creating the computers or machines as intelligent as human beings.
- It is the science and engineering of making intelligent machines, especially intelligent computer programs.
- It is related to the similar task of using computers to understand human intelligence, but **AI** does not have to confine itself to methods that are biologically observable

What is Artificial Intelligence?

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

According to the father of Artificial Intelligence, John McCarthy, it is *"The science and engineering of making intelligent machines, especially intelligent computer programs"*.

Artificial Intelligence is a way of **making a computer, a computer-controlled robot, or a software think intelligently**, in the similar manner the intelligent humans think.

What is Artificial Intelligence?

Examples:

- Speech recognition,
- Face detection and recognition,
- Object detection and recognition,
- Learning new skills,
- Decision making,
- Abstract thinking

How …?

- AI is accomplished by studying how human brain thinks and how humans learn, decide, and work while trying to solve a problem, and then using the outcomes of this study as a basis of developing intelligent software and systems.

Why Artificial Intelligence?

It has gained prominence recently due, in part, to big data, or the increase in speed, size and variety of data businesses are now collecting. AI can perform tasks such as identifying patterns in the data more efficiently than humans, enabling businesses to gain more insight out of their data.

From a **business** perspective AI is a set of very powerful tools, and methodologies for using those tools to solve business problems.

From a **programming** perspective, AI includes the study of symbolic programming, problem solving, and search.

AI Vocabulary

- **Intelligence** relates to tasks involving higher mental processes, e.g. creativity, solving problems, pattern recognition, classification, learning, induction, deduction, building analogies, optimization, language processing, knowledge and many more. Intelligence is the computational part of the ability to achieve goals.
- **Intelligent behaviour** is depicted by perceiving one's environment, acting in complex environments, learning and understanding from experience, reasoning to solve problems and discover hidden knowledge, applying knowledge successfully in new situations, thinking abstractly, using analogies, communicating with others and more.

AI Vocabulary

- **Science based goals of AI** pertain to developing concepts, mechanisms and understanding biological intelligent behaviour. The emphasis is on understanding intelligent behaviour.
- **Engineering based goals of AI** relate to developing concepts, theory and practice of building intelligent machines. The emphasis is on system building.
- **Applications of AI** refers to problem solving, search and control strategies, speech recognition, natural language understanding, computer vision, expert systems, etc.

AI Vocabulary

- **AI Techniques** depict how we represent, manipulate and reason with knowledge in order to solve problems. Knowledge is a collection of ‘facts’. To manipulate these facts by a program, a suitable representation is required. A good representation facilitates problem solving.
- **Learning** means that programs learn from what facts or behaviour can represent. Learning denotes changes in the systems that are adaptive in other words, it enables the system to do the same task(s) more efficiently next time.

Task Domains of AI

Mundane Tasks:

- Perception
  - Vision
  - Speech
- Natural Languages
  - Understanding
  - Generation
  - Translation
- Common sense reasoning
- Robot Control

Formal Tasks

- Games : chess, checkers etc
- Mathematics: Geometry, logic, Proving properties of programs

Expert Tasks:

- Engineering (Design, Fault finding, Manufacturing planning)
- Scientific Analysis
- Medical Diagnosis
- Financial Analysis

AI Problems

- A person who knows how to perform tasks from several of the categories shown in above list learn the necessary skills in a standard order.
  - First perceptual, linguistic, and commonsense skills are learned.
  - Later expert skills such as engineering, medicine, or finance are acquired
- Earlier skills are easier, for this reason much of the initial work in AI work was concentrated in those early areas.
- The problem areas where now AI is flourishing most as a practical discipline are primarily the domains that require only specialized expertise without the assistance of commonsense knowledge.
- Expert systems (AI programs) now are up for day-to-day tasks that aim at solving part, or perhaps all, of practical, significant problem that previously required high human expertise.

STEPS TO SOLVE A PROBLEM

To solve the problem of building a system you should take the following steps:

Define the problem accurately including detailed specifications and what constitutes a suitable solution.

Scrutinize the problem carefully, for some features may have a central affect on the chosen method of solution.

Segregate and represent the background knowledge needed in the solution of the problem.

Choose the best solving techniques for the problem to solve a solution.

Tic Tac Toe

Three programs are presented :

The programs in the Series increase in

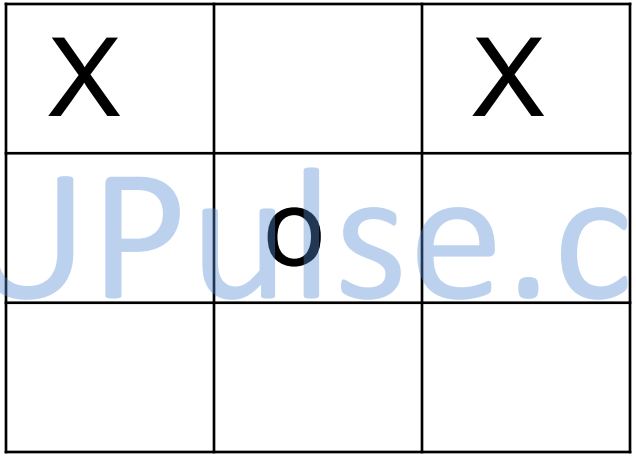
Their complexity

Use of generalization

Clarity of their knowledge

Extensibility of their approach

Introductory Problem: Tic-Tac-Toe



Introductory Problem: Tic-Tac-Toe

Program 1:
Data Structures:
Board: 9 element vector representing the board, with 1-9 for each square. An element contains the value 0 if it is blank, 1 if it is filled by X, or 2 if it is filled with a O
Movetable: A large vector of 19,683 elements (3^9), each element is 9-element vector.

Algorithm:

1. View the vector as a ternary number. Convert it to a decimal number.
2. Use the computed number as an index into Move-Table and access the vector stored there.
3. Set the new board to that vector.

Introductory Problem: Tic-Tac-Toe

Comments:
This program is very efficient in time.

1. A lot of space to store the Move-Table.

2. A lot of work to specify all the entries in the Move-Table.

3. Difficult to extend.

Introductory Problem: Tic-Tac-Toe

1	2	3
4	5	6
7	8	9

Introductory Problem: Tic-Tac-Toe

Program 2:
Data Structure: A nine element vector representing the board. But instead of using 0,1 and 2 in each element, we store 2 for blank, 3 for X and 5 for O
Functions:
Make2: returns 5 if the center sqaure is blank. Else any other balnk sq
Posswin(p): Returns 0 if the player p cannot win on his next move; otherwise it returns the number of the square that constitutes a winning move. If the product is 18 (3x3x2), then X can win. If the product is 50 (5x5x2) then O can win.
Go(n): Makes a move in the square n

Strategy:

Turn = 1	Go(1)
Turn = 2	If Board[5] is blank, Go(5), else Go(1)
Turn = 3	If Board[9] is blank, Go(9), else Go(3)
Turn = 4	If Posswin(X) $\neq$ 0, then Go(Posswin(X))

.....

Introductory Problem: Tic-Tac-Toe

Comments:

1. Not efficient in time, as it has to check several conditions before making each move.

2. Easier to understand the program's strategy.

3. Hard to generalize.

Introductory Problem: Tic-Tac-Toe

8	3	4
1	5	9
6	7	2

15 – (8 + 5)

Introductory Problem: Tic-Tac-Toe

Comments:

1. Checking for a possible win is quicker.
2. Human finds the row-scan approach easier, while computer finds the number-counting approach more efficient.

PROBLEMS, PROBLEM SPACES AND SEARCH

—State space search

—Search strategies

—Problem characteristics

—Design of search programs

Introductory Problem: Tic-Tac-Toe

Program 3:

1. If it is a win, give it the highest rating.
2. Otherwise, consider all the moves the opponent could make next. Assume the opponent will make the move that is worst for us. Assign the rating of that move to the current node.
3. The best node is then the one with the highest rating.

STATE SPACE SEARCH

A state space consists of

- A (possibly infinite) set of states
  - The start state represents the initial problem
  - Each state represents some configuration reachable from the start state
  - Some states may be goal states (solutions)
- A set of rules
  - Applying an operator to a state transforms it to another state in the state space
  - Not all operators are applicable to all states

State spaces are used extensively in Artificial Intelligence (AI)

STATE SPACE SEARCH - Example 1: Maze

A maze can be represented as a state space

- Each state represents “where you are” in the maze
- The start state represents your starting position
- The goal state represents the exit from the maze

Rules (for a rectangular maze) are: move north, move south, move east, and move west

- Each rule takes you to a new state (maze location)
- Rules may not always apply, because of walls in the maze

STATE SPACE SEARCH - Example 2: The 15-puzzle

Start state:

3	10	13	7
9	14	6	1
4		15	2
11	8	5	12

Goal state:

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

The start state is some (almost) random configuration of the tiles

The goal state is as shown

Rules are

Move empty space up

Move empty space down

Move empty space right

Move empty space left

Rules apply if not against edge

STATE SPACE SEARCH - Example 3: Playing Chess

- Each position can be described by an 8-by-8 array.
- Initial position is the game opening position.
- Goal position is any position in which the opponent does not have a legal move and his or her king is under attack.
- Legal moves can be described by a set of rules:
  - Left sides are matched against the current state.
  - Right sides describe the new resulting state.

STATE SPACE SEARCH

- Many problems in AI take the form of ***state-space search***.
- The ***states*** might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
- The ***state-space*** is the configuration of the possible states and how they connect to each other e.g. the legal moves between states.
- When we don't have an *algorithm* which tells us definitively how to negotiate the state-space we need to search the state-space to find an optimal path from a start state to a goal state.
- We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible.
- Chess, for example, is a very difficult state-space search problem.

STATE SPACE SEARCH - SEARCHING FOR THE OPTIMUM

- State-space search is all about finding, in a state-space (which may be *extremely* large: e.g. chess), some *optimal state/node*.
- `Optimal' can mean very different things depending on the nature of the domain being searched.
- For a puzzle, `optimal' might mean the goal state e.g. connect4
- For a route-finder, like our problem, which searches for shortest routes between towns, or components of an integrated circuit, `optimal' might mean the shortest path between two towns/components.
- For a game such as chess, in which we typically can't see the goal state, `optimal' might mean the best move we think we can make, *looking ahead* to see what effects the possible moves have.

STATE SPACE SEARCH - SEARCHING FOR THE OPTIMUM

- The state space can be HUGE! (Combinatorial explosion)
- Theorem Proving: Infinite!
- Chess: 10^{120} (in an average length game)
- Checkers: 10^{40}
- Eight puzzle:181,440
- Right representation helps

STATE-SPACE REPRESENTATION: GENERAL OUTLINE

Select some way to represent states in the problem in an unambiguous way.

Formulate all actions that can be preformed in states:

including their preconditions and effects

== PRODUCTION RULES

Represent the initial state (s).

Formulate precisely when a state satisfies the goal of our problem.

Activate the production rules on the initial state and its descendants, until a goal state is reached.

Example: the 8-puzzle

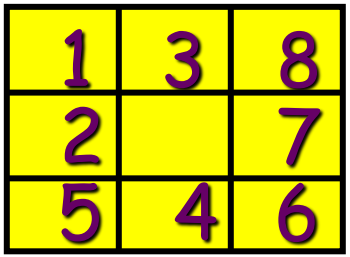
- Given: a board situation for the 8-puzzle:

1	3	8
2		7
5	4	6

- Problem: find a sequence of moves (allowed under the rules of the 8-puzzle game) that transform this board situation in a desired goal situation

1	2	3
8		4
7	6	5

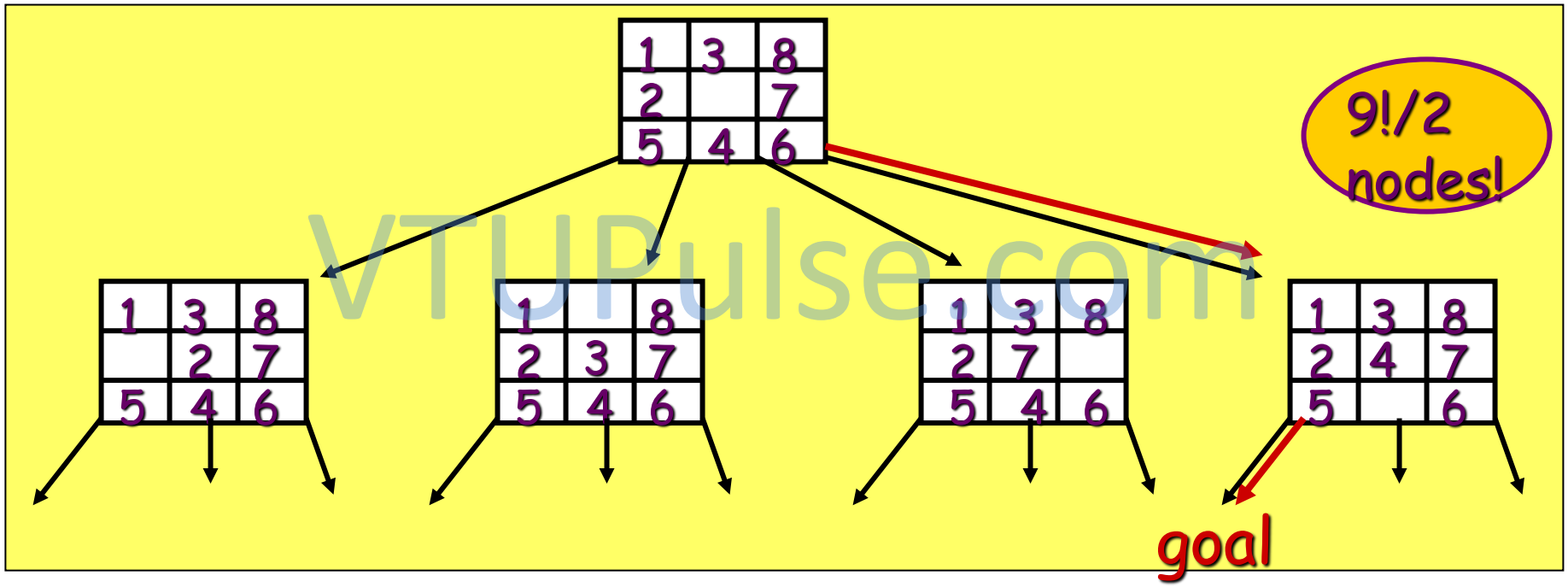
INITIAL ISSUES TO SOLVE:



- How to formulate production rules? (repr. 2)
  - ➔ Ex.:
    - express how/when squares may be moved?
    - Or: express how/when the blank space is moved?
- How to represent states? (repr. 1)
  - Ex: using a 3 X 3 matrix
- When is a rule applicable to a state? (matching)
- How to formulate when the goal criterion is satified and how to verify that it is?
- How/which rules to activate? (control)

The (implicit) search tree

- Each state-space representation defines a search tree:



ISSUES AND TRADE-OFFS

1. How to choose the rules?

2. Should we search through the implicit tree or through an implicit graph?

3. Do we need an optimal solution, or just any solution?

   'optimal path problems'

4. Can we decompose states into components on which simple rules can in an independent way?

   Problem reduction or decomposability

5. Should we search forwards from the initial state, or backwards from a goal state?

State Space Search: Water Jug Problem

- “You are given two jugs, a 4-litre one and a 3-litre 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 litres of water into 4-litre jug.”

State Space Search: Water Jug Problem

- State: (x, y)

  x = 0, 1, 2, 3, or 4 y = 0, 1, 2, 3

- x represents quantity of water in 4 gallon jug and y represents quantity of water in 3 gallon jug
- Start state: (0, 0).
- Goal state: (2, n) for any n.
- Attempting to end up in a goal state.

State Space Search: Water Jug Problem – Production Rules

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

State Space Search: Water Jug Problem – One Possible Solution

current state = (0, 0)

2. Loop until reaching the goal state (2, 0)
   - Apply a rule whose left side matches the current state
   - Set the new current state to be the resulting state

(0, 0)

(0, 3) – Rule 2

(3, 0) - Rule 9

(3, 3) - Rule 2

(4, 2) – Rule 9

(0, 2) - Rule 5

(2, 0) – Rule 9

State Space Search: Water Jug Problem – One Possible Solution

1. current state = (0, 0)
2. Loop until reaching the goal state (2, 0)
   - Apply a rule whose left side matches the current state
   - Set the new current state to be the resulting state

(0, 0)

(4, 0) – Rule 1

(1, 3) - Rule 3

(1, 0) - Rule 6

(0, 1) – Rule 3

(4, 1) - Rule 1

(2, 3) – Rule 6

Production Systems

Since search forms the core of many intelligent processes, it is useful to structure Al programs in a way that facilitates **describing and performing the search process**.

Production systems provide such structures.

A production system consists of:

A set of rules, each consisting of a left side (a pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.;

One or more knowledge/databases that contain whatever information is appropriate for the particular task. Some pans of the database may be permanent, while other plans of it may pertain only to the solution of the current problem. A control strategy that specifies the order in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.

A rule applier.

Control / Search Strategies

- So far, we have completely ignored the question of how to decide which rule to apply next during the process of searching for a solution to a problem.
- This question arises once often more than one rule (and sometimes fewer than one rule) will have its left side match the current state.
- Even without a great deal of thought, it is clear that how such decisions are made will have a crucial impact on how quickly. and even whether, a problem is finally solved.

Control / Search Strategies

- The first requirement of a good control strategy is that it causes **motion**. Consider again the water jug problem. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, 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 a solution.

Control / Search Strategies

- The second requirement of a good control strategy is that it be **systematic**. Consider again the water jug problem. On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. But 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.

Search Strategies - Breadth-First Search

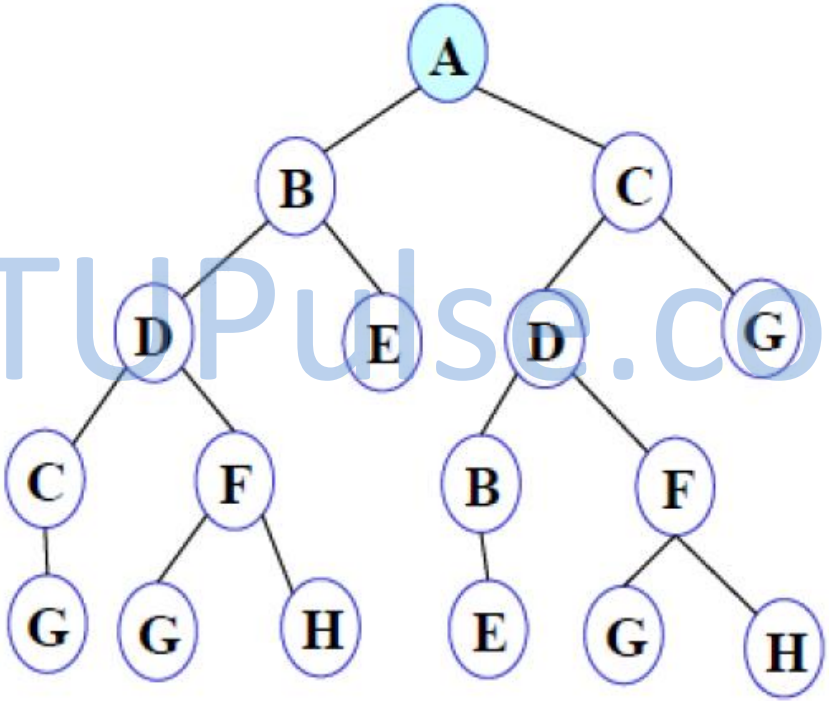
Algorithm: Breadth-First Search

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 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

Breadth-First Search - Example

- **Step 1:** Initially NODE-LIST contains only one node corresponding to the source state A.

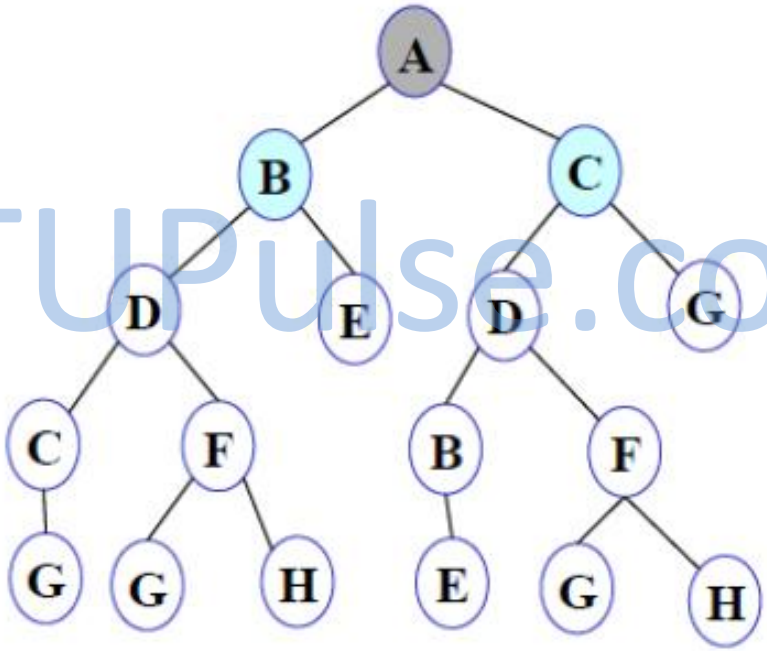
- **NODE-LIST: A**



Breadth-First Search - Example

- **Step 2:** A is removed from NODE-LIST. The node is expanded, and its children B and C are generated. They are placed at the back of NODE-LIST.

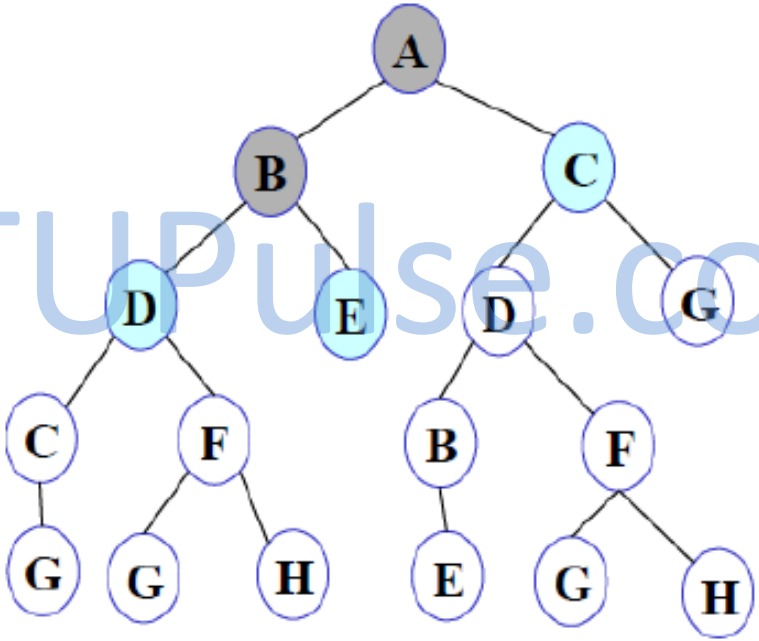
- **NODE-LIST: B C**



Breadth-First Search - Example

- **Step 3:** Node B is removed from NODE-LIST and is expanded. Its children D, E are generated and put at the back of NODE-LIST.

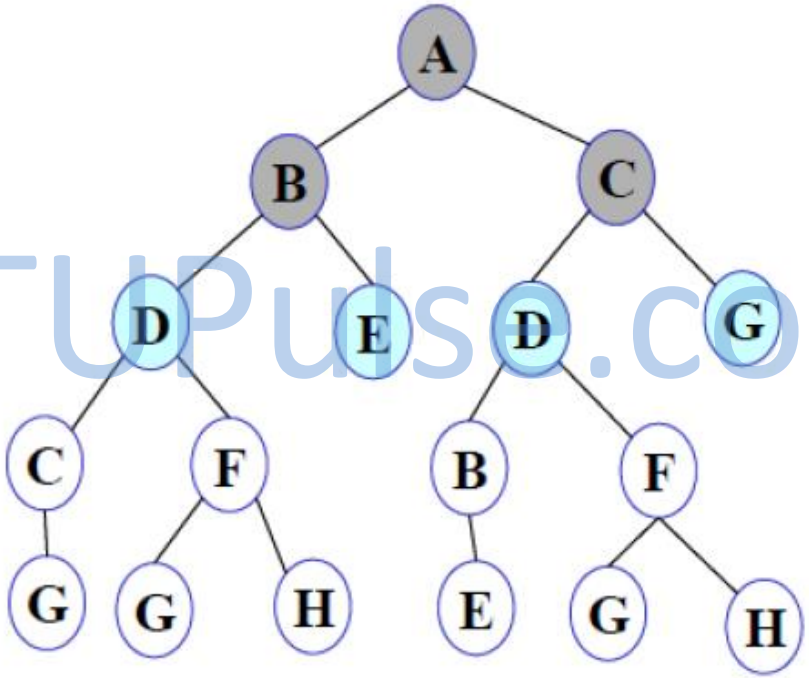
- **NODE-LIST: C D E**



Breadth-First Search - Example

- **Step 4:** Node C is removed from NODE-LIST and is expanded. Its children D and G are added to the back of NODE-LIST.

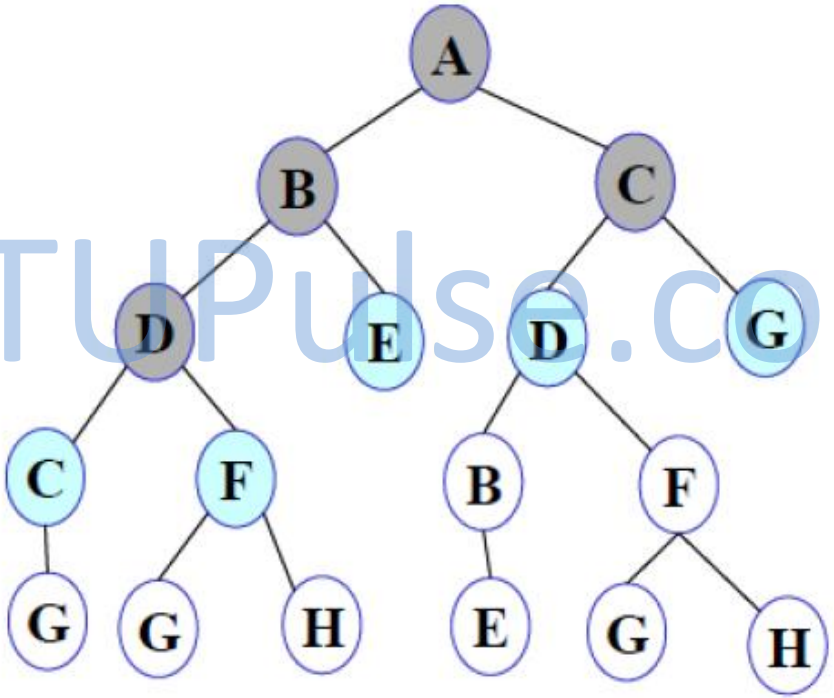
- **NODE-LIST: D E D G**



Breadth-First Search - Example

- **Step 5**: Node D is removed from NODE-LIST. Its children C and F are generated and added to the back of NODE-LIST.

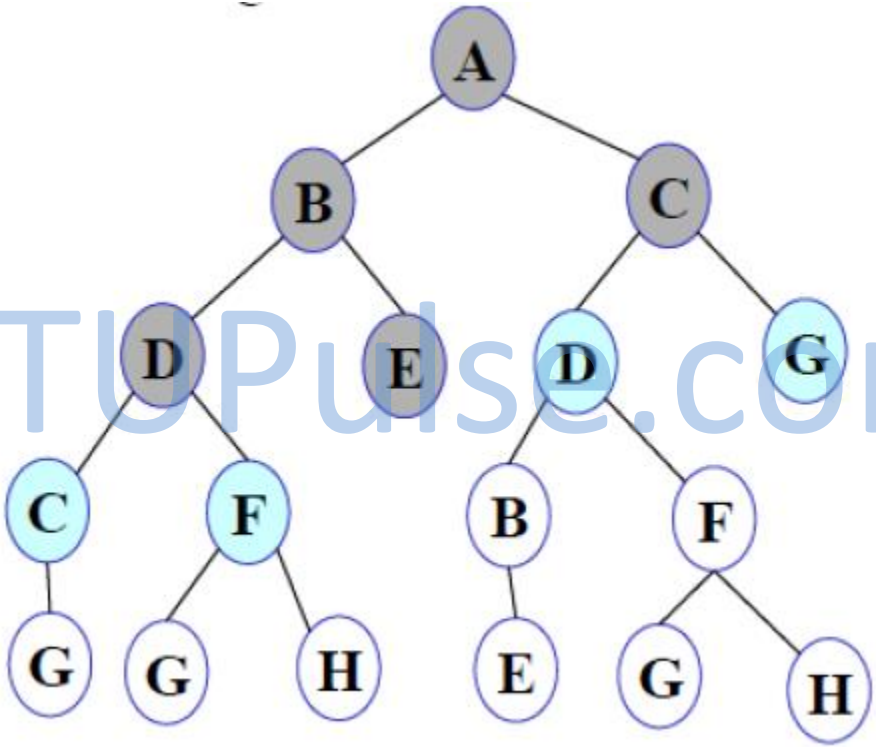
- **NODE-LIST: E D G C**



Breadth-First Search - Example

- **Step 6**: Node E is removed from NODE-LIST. It has no children.

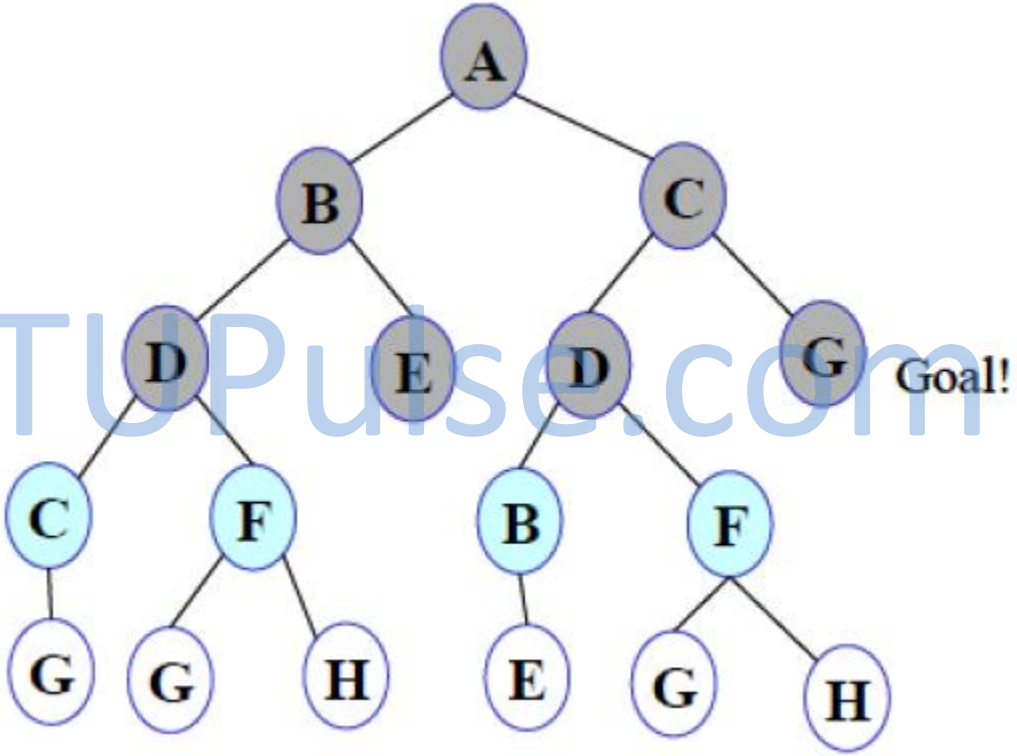
- **NODE-LIST: D G C F**



Breadth-First Search - Example

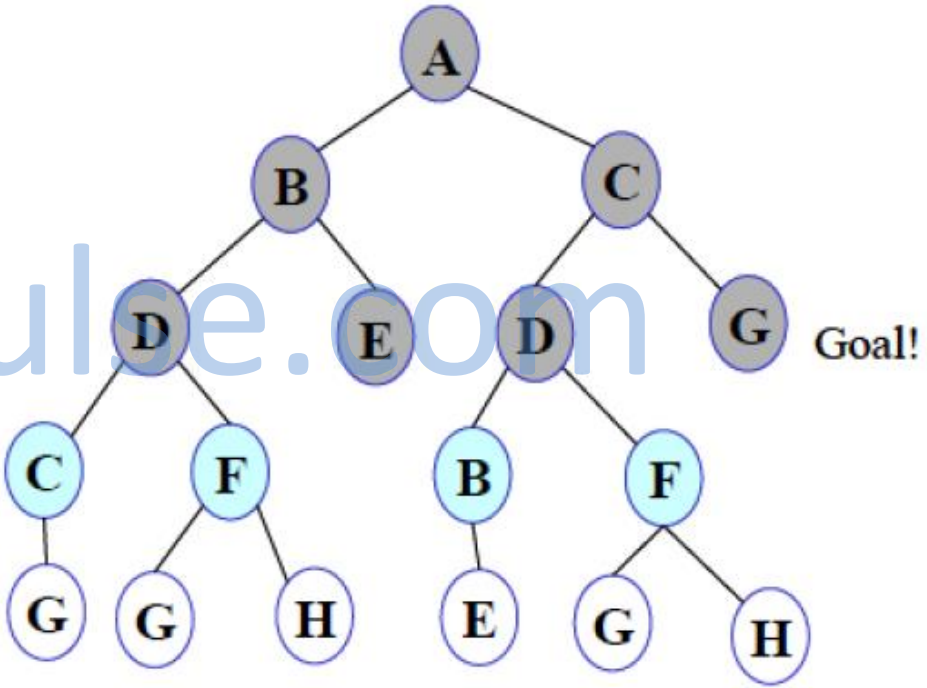
- **Step 7**: D is expanded; B and F are put in OPEN.

- **NODE-LIST: G C F E**



Breadth-First Search - Example

- **Step 8**: G is selected for expansion. It is found to be a goal node. So the algorithm returns the path A C G by following the parent pointers of the node corresponding to G. The algorithm terminates.

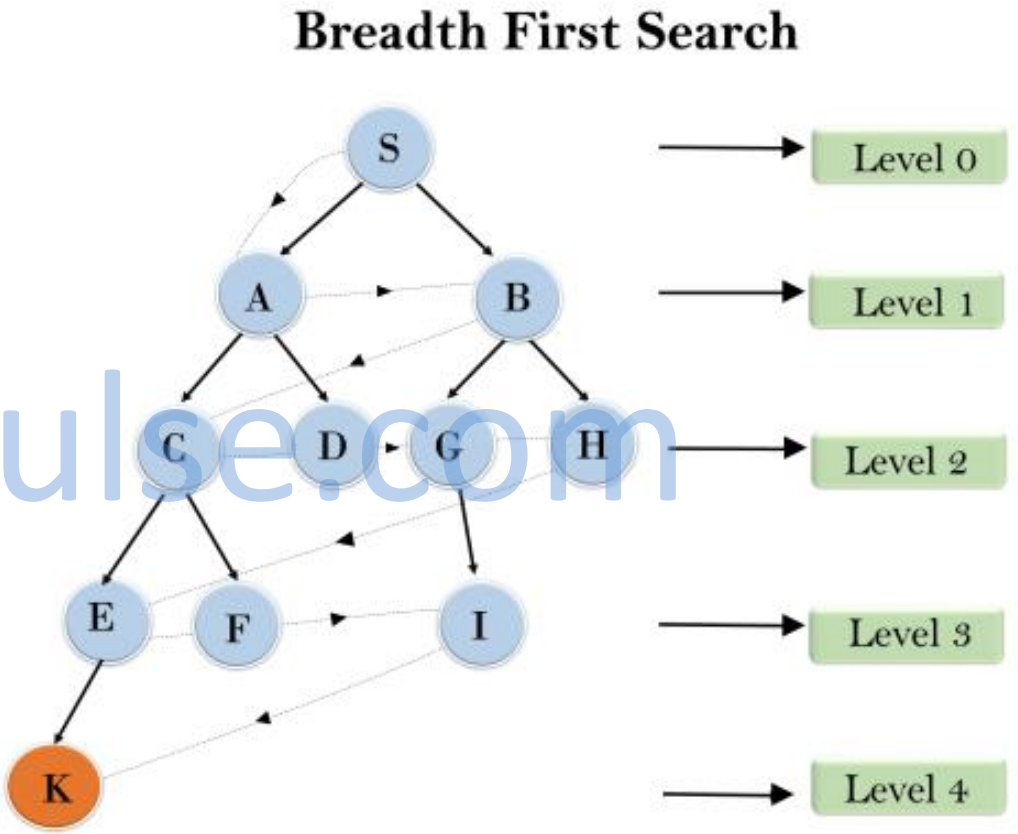


Breadth-First Search

- **Breadth first search** is:
- One of the simplest search strategies
- Complete. If there is a solution, BFS is guaranteed to find it.
- If there are multiple solutions, then a minimal solution will be found
- The algorithm is optimal (i.e., admissible) if all operators have the same cost. Otherwise, breadth first search finds a solution with the shortest path length.
- **Advantages**: Finds the path of minimal length to the goal.
- **Disadvantages**:
- Requires the generation and storage of a tree whose size is exponential the depth of the shallowest goal node.
- The breadth first search algorithm cannot be effectively used unless the search space is quite small.

Search Strategies

- Breadth-first search
  Expand all the nodes of one level first.



Search Strategies - Depth-First Search

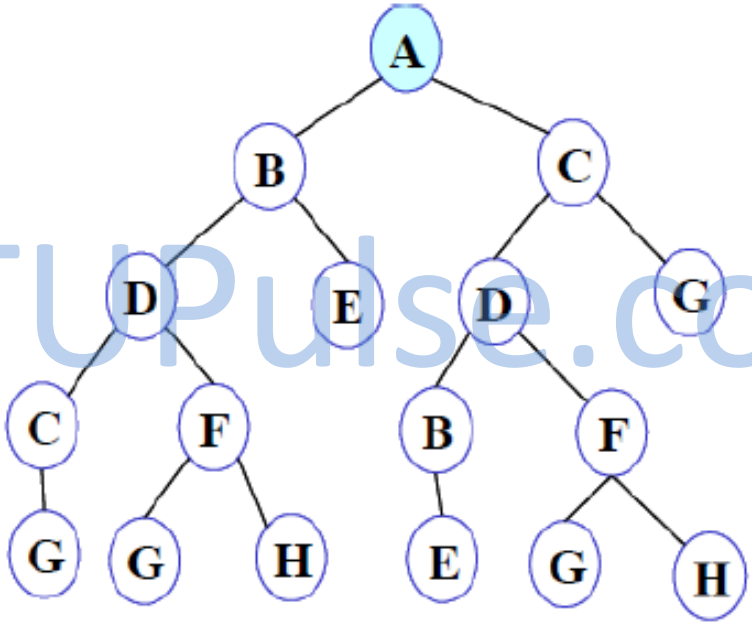
Algorithm: Depth-First Search

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.

Depth-First Search - Example

- **Step 1:** Initially NODE-LIST contains only one node corresponding to the source state A.

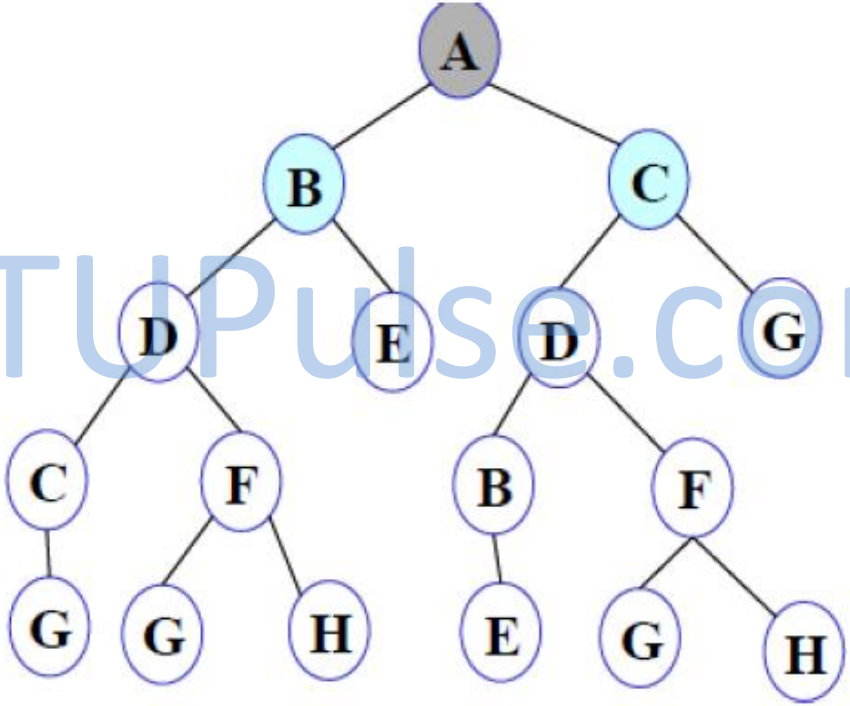
- **NODE-LIST: A**



Depth-First Search - Example

- **Step 2:** A is removed from NODE-LIST . A is expanded and its children B and C are put in front of NODE-LIST .

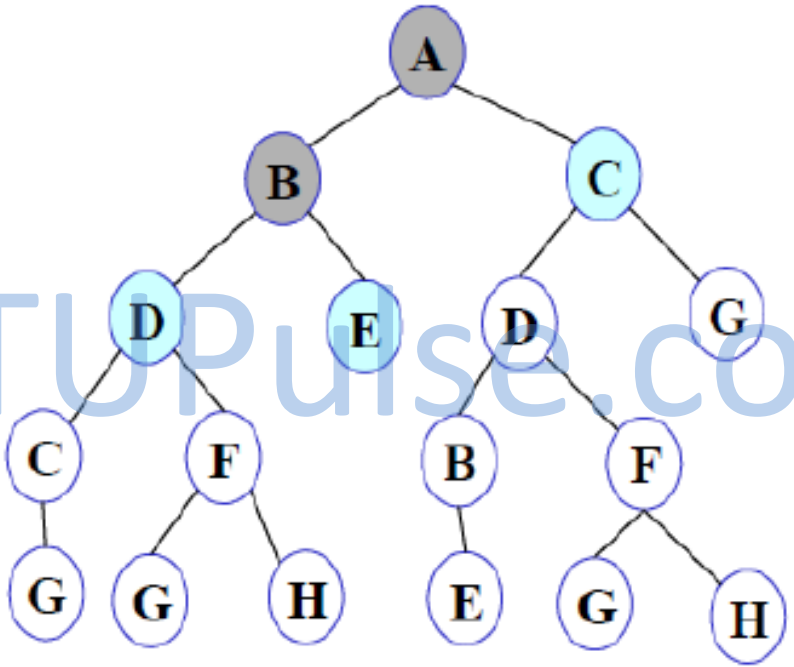
- **NODE-LIST: B C**



Depth-First Search - Example

- **Step 3:** Node B is removed from NODE-LIST , and its children D and E are pushed in front of NODE-LIST

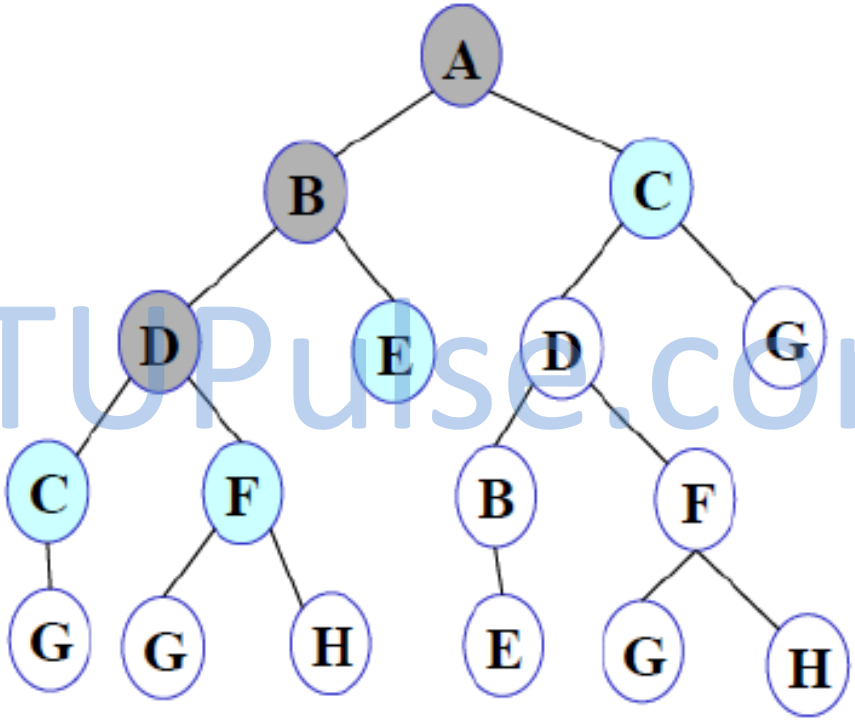
- **NODE-LIST: D E C**



Depth-First Search - Example

- **Step 4:** Node D is removed from NODE-LIST . C and F are pushed in front of NODE-LIST .

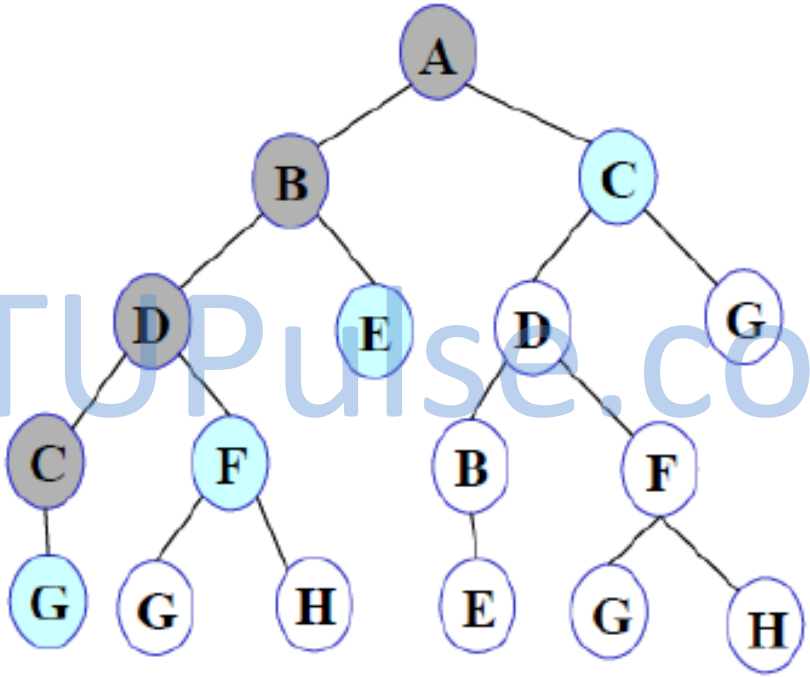
- **NODE-LIST: C F E**



Depth-First Search - Example

- **Step 5:** Node C is removed from NODE-LIST . Its child G is pushed in front of NODE-LIST .

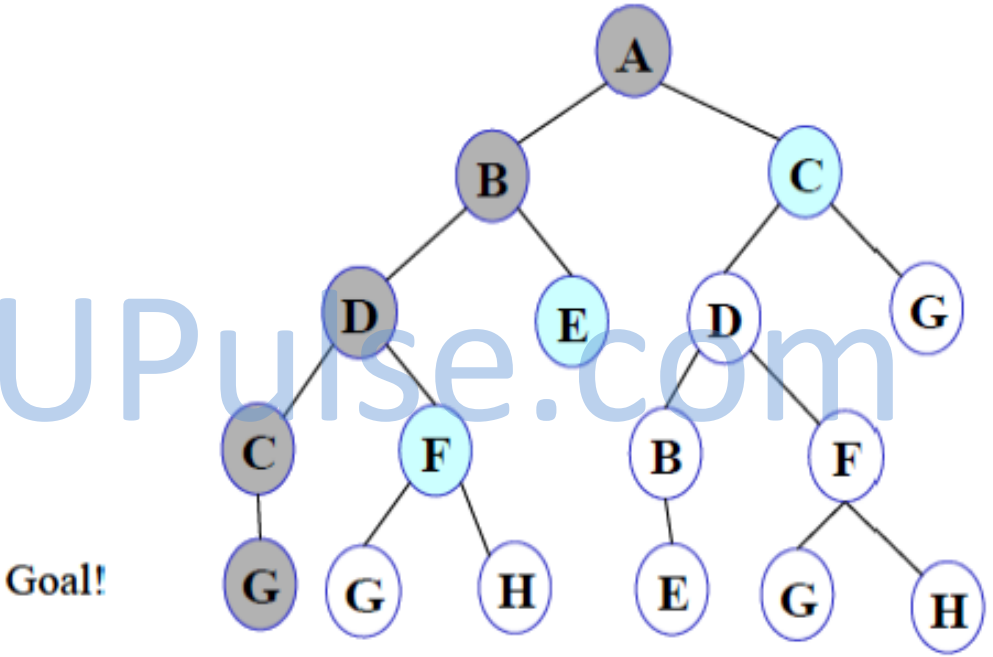
- **NODE-LIST: *G* F E C**



Depth-First Search - Example

- **Step 6:** Node G is expanded and found to be a goal node.

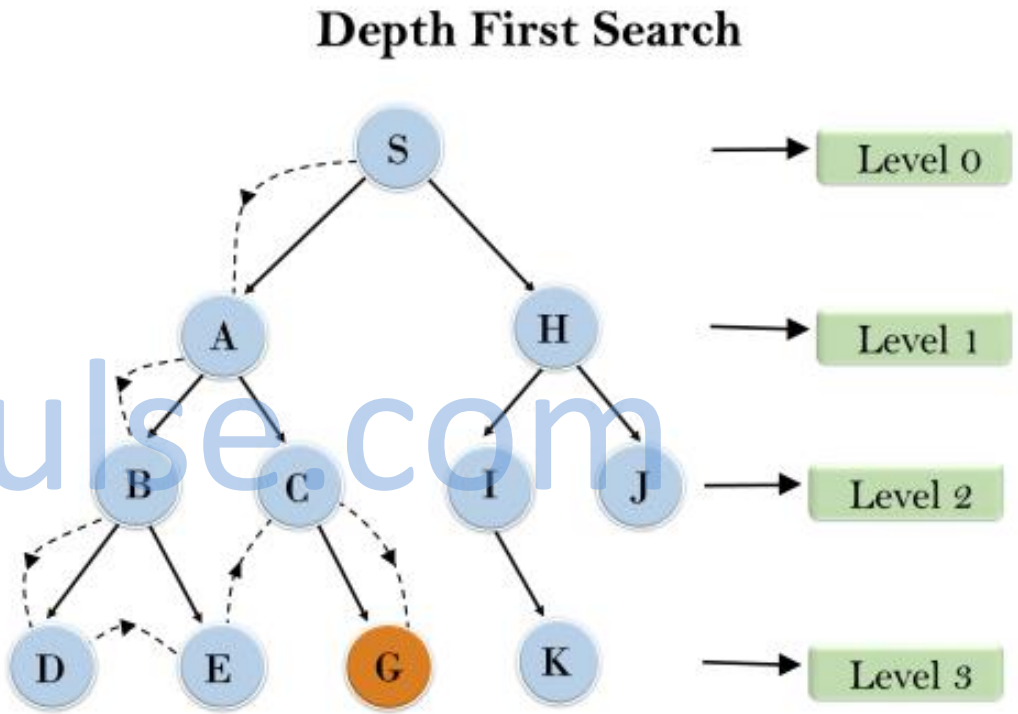
- **NODE-LIST: *G* F E C**



- The solution path **A-B-D-C-G** is returned and the algorithm terminates.

Depth-First Search - Example

- Depth-first search
  Expand one of the nodes at the deepest level.



Search Strategies

Advantages of Depth-First Search

- Depth-first search requires less memory since only the nodes on the current path are stored. This contrasts with breadth-first search, where all of the tree that has so far been generated must be stored.
- Depth-first search may find a solution without examining much of the search space at all. This contrasts with breadth-first search in which all parts of the tree must be examined to level n before any nodes on level n + i can be examined. This is particularly significant if many acceptable solutions exist. Depth-first search can stop when one of them is found.

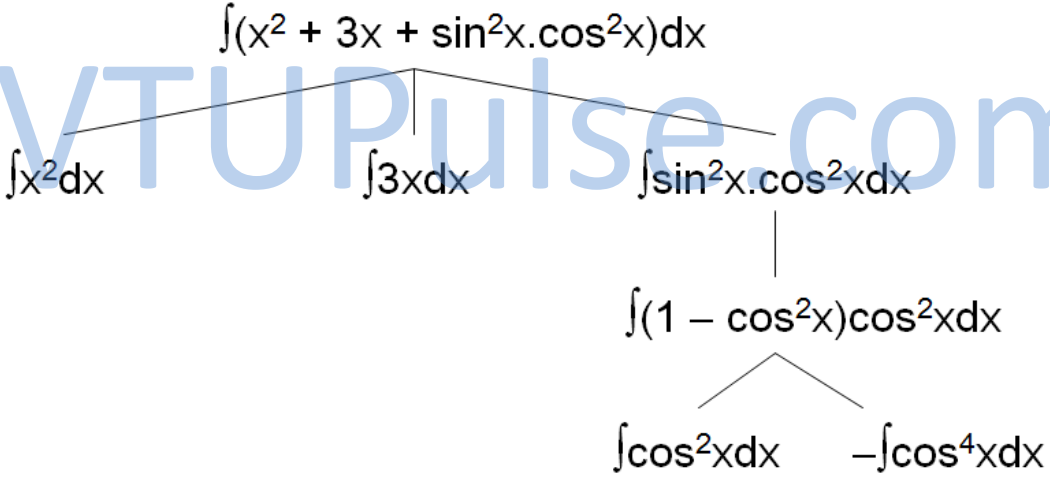
Problem Characteristics

To choose an appropriate method for a particular problem:

- Is the problem decomposable?
- Can solution steps be ignored or undone?
- Is the universe predictable?
- Is a good solution absolute or relative?
- Is the solution a state or a path?
- What is the role of knowledge?
- Does the task require human-interaction?

Is the problem decomposable?

- Can the problem be broken down to smaller problems to be solved independently?
- Decomposable problem can be solved easily.



Can solution steps be ignored or undone?

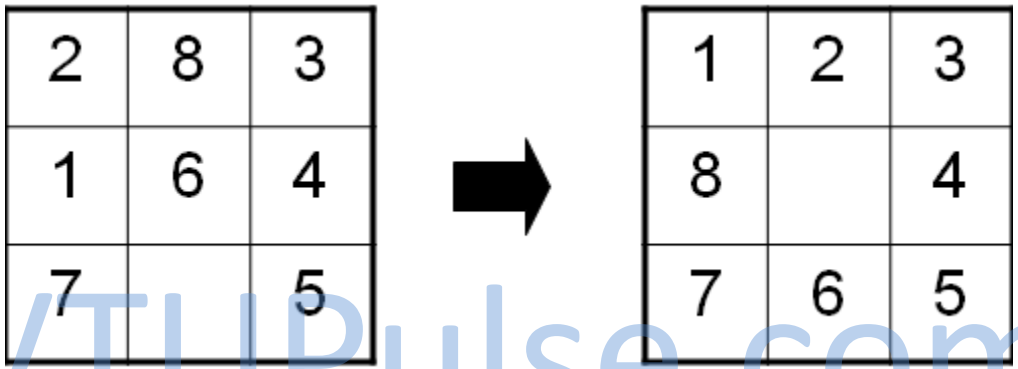
Theorem Proving

A lemma that has been proved can be ignored for next steps.

Ignorable!

Can solution steps be ignored or undone?

The 8-Puzzle



Moves can be undone and backtracked.

Recoverable!

Can solution steps be ignored or undone?

Playing Chess

Moves cannot be retracted.

Irrecoverable!

Can solution steps be ignored or undone?

- Ignorable problems can be solved using a simple control structure that never backtracks.
- Recoverable problems can be solved using backtracking.
- Irrecoverable problems can be solved by recoverable style methods via planning.

Is the universe predictable?

Playing Bridge

- We cannot know exactly where all the cards are or what the other players will do on their turns.

Uncertain outcome!

Is the universe predictable?

- For certain-outcome problems, planning can used to generate a sequence of operators that is guaranteed to lead to a solution.
- For uncertain-outcome problems, a sequence of generated operators can only have a good probability of leading to a solution. Plan revision is made as the plan is carried out and the necessary feedback is provided.

Is a good solution absolute or relative?

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 A.D.
6. No mortal lives longer than 150 years.
7. It is now 2016 A.D.

Is Marcus alive?

Is a good solution absolute or relative?

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 A.D.	axiom 3
7. It is now 1991 A.D.	axiom 7
9. Marcus' age is 1951 years.	3, 7
6. No mortal lives longer than 150 years.	axiom 6
10. Marcus is dead.	

OR

7. It is now 1991 A.D.	axiom 7
5. All Pompeians died in 79 A.D.	axiom 5
11. All Pompeians are dead now.	5, 7
2. Marcus was a Pompeian.	axiom 2
12. Marcus is dead.	

Is a good solution absolute or relative?

- The Travelling Salesman Problem
- We have to try all paths to find the shortest one.

- Any-path problems can be solved using heuristics that suggest good paths to explore.

- For best-path problems, much more exhaustive search will be performed.

Is the solution a state or a path?

Finding a consistent interpretation for

"The bank president ate a dish of pasta salad with the fork".

- "bank" refers to a financial situation or to a side of a river?
- "dish" or "pasta salad" was eaten?
- Does "pasta salad" contain pasta, as "dog food" does not contain "dog"?
- Which part of the sentence does "with the fork" modify?

  What if "with vegetables" is there?

No record of the processing is necessary.

Is the solution a state or a path?

- The Water Jug Problem
- The path that leads to the goal must be reported.

- A path-solution problem can be reformulated as a state-solution problem by describing a state as a partial path to a solution.

- The question is whether that is natural or not.

What is the role of knowledge

Playing Chess

- Consider again the problem of playing chess. Suppose you had unlimited computing power available.
- How much knowledge would be required by a perfect program? The answer to this question is very little—just the rules for determining legal moves and some simple control mechanism that implements an appropriate search procedure.
- Additional knowledge about such things as good strategy and tactics could of course help considerably to constrain the search and speed up the execution of the program.
- Knowledge is important only to constrain the search for a solution.

What is the role of knowledge

Reading Newspaper

- Now consider the problem of scanning daily newspapers to decide which are supporting the Democrats and which are supporting the Republicans in some upcoming election.
- Again assuming unlimited computing power, how much knowledge would be required by a computer trying to solve this problem? This time the answer is a great deal.
- It would have to know such things as:
  - The names of the candidates in each party.
  - The fact that if the major thing you want to see done is have taxes lowered, you are probably supporting the Republicans.
  - The fact that if the major thing you want to see done is improved education for minority students, you are probably supporting the Democrats.
  - The fact that if you are opposed to big government, you are probably supporting the Republicans.
  - And so on ...
- Knowledge is required even to be able to recognize a solution.

Does the task require human-interaction?

- 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 the interaction between the computer and its human users is **problem-in solution-out**.
- But increasingly we are building programs that require intermediate interaction with people, both to provide additional input to the program and to provide additional reassurance to the user.

Does the task require human-interaction?

- Solitary problem, in which there is no intermediate communication and no demand for an explanation of the reasoning process.

- Conversational problem, in which intermediate communication is to provide either additional assistance to the computer or additional information to the user.

Problem Classification

- There is a variety of problem-solving methods, but there is no one single way of solving all problems.

- Not all new problems should be considered as totally new. Solutions of similar problems can be exploited.

Production system

- We have just examined a set of characteristics that distinguish various classes of problems

- It has also been shown that production systems are a good way to describe the operations that can be performed in a search for good solution

Classes / Categories of Production systems

Monotonic Production System: the application of a rule never prevents the later application of another rule that could also have been applied at the time the first rule was selected

Non-Monotonic Production system: is one in which this is not true

Partially commutative Production system: property that if application of a particular sequence of rules transforms state x to state y, then permutation of those rules allowable, also transforms state x into state y.

Commutative Production system: both monotonic and Partially commutative

Relationship between classes of systems

	Monotonic	Non-Monotonic
Partially Commutative	Theorem Proving	Robot Navigation
Not Partially commutative	Chemical Synthesis	Bridge

Partially Commutative and Monotonic

- These production systems are useful for solving ignorable problems.
- **Example: Theorem Proving**
- They can be implemented without the ability to backtrack to previous states when it is discovered that an incorrect path has been followed.
- This often results in a considerable increase in efficiency, particularly because since the database will never have to be restored, It is not necessary to keep track of where in the search process every change was made.
- They are good for problems where things do not change; new things get created.

Partially Commutative and Non-Monotonic

- Useful for problems in which changes occur but can be reversed and in which order of operations is not critical.
- **Example: Robot Navigation, 8-puzzle, blocks world**
- Suppose the robot has the following ops: go North (N), go East (E), go South (S), go West (W).
- To reach its goal, it does not matter whether the robot executes the N-N-E or N-E-N.

Not Partially Commutative

- Problems in which irreversible change occurs
- **Example: chemical synthesis**
- The ops can be :Add chemical x to the pot, Change the temperature to t degrees.
- These ops may cause irreversible changes to the potion being brewed.
- The order in which they are performed can be very important in determining the final output.
- (X+y) +z is not the same as (z+y) +x

Issues in the design of search programs

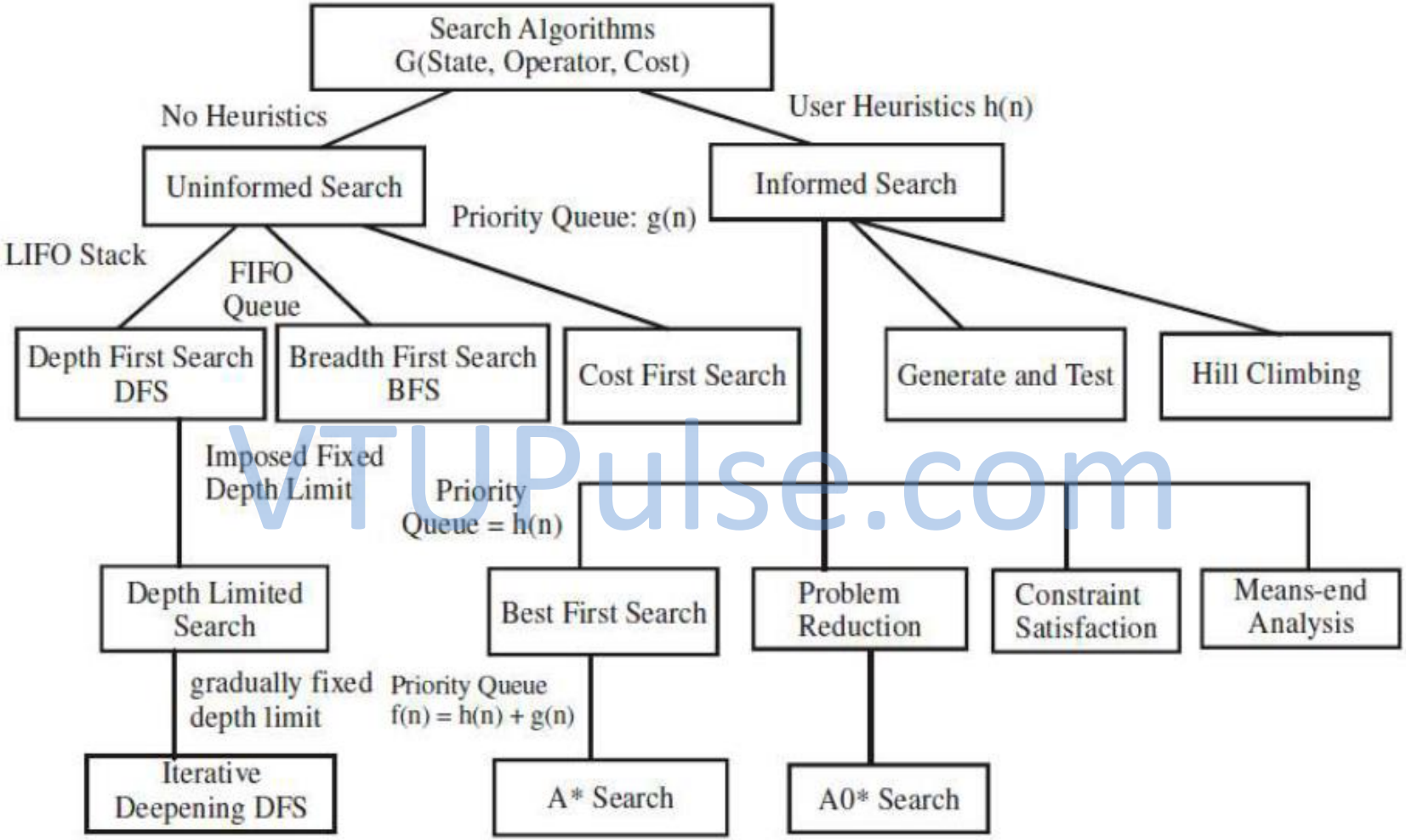
- The direction in which to conduct the search **(forward versus backward reasoning)**. We can search forward through the state space from the start state to a goal state, or we can search backward from the goal.
- How to select applicable rules **(matching).** Production systems typically spend most of their time looking for rules to apply, so it is critical to have efficient procedures for matching rules against states.
- How to represent each node of the search process **(the knowledge representation problem and the frame problem)**.
  - For problems like chess, a node can be fully represented by a simple array.
  - In more complex problem solving, however, it is inefficient and/or impossible to represent all of the facts in the world and to determine all of the side effects an action may have.

Search Algorithms / Techniques

- ***Uninformed search algorithms*** or *Brute-force algorithms*, search through the search space all possible candidates for the solution checking whether each candidate satisfies the problem's statement.
- ***Informed search algorithms*** use heuristic functions that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

Search Algorithms / Techniques

- A good heuristic will make an informed search dramatically outperform any uninformed search: for example, the Traveling Salesman Problem (TSP), where the goal is to find is a good solution instead of finding the best solution.
- In such problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution.
- Such techniques help in finding a solution within reasonable time and space (memory).



Requirement of Search Algorithms / Techniques

- *The first requirement is that it causes **motion***, in a game playing program, it moves on the board and in the water jug problem, filling water is used to fill jugs. It means the control strategies without the motion will never lead to the solution.
- *The second requirement is that it is **systematic***, that is, it corresponds to the need for global motion as well as for local motion. This is a clear condition that neither would it be rational to fill a jug and empty it repeatedly, nor it would be worthwhile to move a piece round and round on the board in a cyclic way in a game. We shall initially consider two systematic approaches for searching. Searches can be classified by the order in which operators are tried: depth-first, breadth-first, bounded depth-first.

Informed search Algorithms / Techniques

- Many Informed search Algorithms techniques are developed, using ***heuristic functions.***
- The algorithms that use *heuristic functions* are called ***heuristic algorithms***.
- Heuristic algorithms are not really intelligent; they appear to be intelligent because they achieve better performance.
- Heuristic algorithms are more efficient because they take advantage of feedback from the data to direct the search path.

Heuristics Search

- To find a solution in proper time rather than a complete solution in unlimited time we use heuristics. ‘A heuristic function is a function that maps from problem state descriptions to measures of desirability, usually represented as numbers’. Heuristic search methods use knowledge about the problem domain and choose promising operators first. These heuristic
- search methods use heuristic functions to evaluate the next state towards the goal state. For finding a solution, by using the heuristic technique, one should carry out the following steps:
  1. Add domain—specific information to select what is the best path to continue searching along.
  2. Define a heuristic function h(n) that estimates the ‘goodness’ of a node n. Specifically, h(n) = estimated cost(or distance) of minimal cost path from n to a goal state.
  3. The term, heuristic means ‘serving to aid discovery’ and is an estimate, based on domain specific information that is computable from the current state description of how close we are to a goal.

Heuristics Search

- Finding a route from one city to another city is an example of a search problem in which different search orders and the use of heuristic knowledge are easily understood.

1. **State**: The current city in which the traveller is located.
2. **Operators**: Roads linking the current city to other cities.
3. **Cost Metric**: The cost of taking a given road between cities.
4. **Heuristic information**: The search could be guided by the direction of the goal city from the current city, or we could use airline distance as an estimate of the distance to the goal.

Heuristic search techniques

- For complex problems, the traditional algorithms, presented above, are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using ***heuristic functions.***
  - Blind search is not always possible, because it requires too much time or Space (memory).
  - Heuristics are ***rules of thumb***; they do not guarantee a solution to a problem.
  - Heuristic Search is a weak technique but can be effective if applied correctly; it requires domain specific information.

Characteristics of heuristic search

- Heuristics are knowledge about domain, which help search and reasoning in its domain.
- Heuristic search incorporates domain knowledge to improve efficiency over blind search.
- Heuristic is a function that, when applied to a state, returns value as estimated merit of state, with respect to goal.
  - Heuristics might (for reasons) *underestimate* or *overestimate* the merit of a state with respect to goal.
  - Heuristics that underestimate are desirable and called admissible.
- Heuristic evaluation function estimates likelihood of given state leading to goal state.
- Heuristic search function estimates cost from current state to goal, presuming function is efficient.

Heuristic search compared with other search

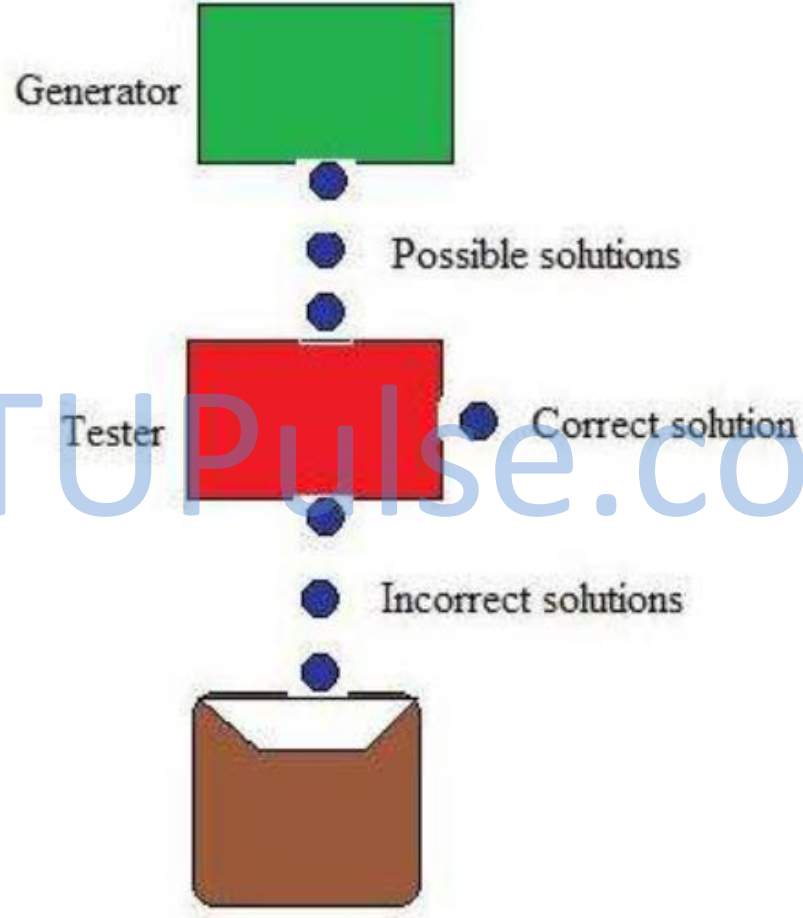
- The Heuristic search is compared with Brute force or Blind search techniques below:
- **Comparison of Algorithms**

Brute force / Blind search	Heuristic search
Can only search what it has knowledge about already	Estimates 'distance' to goal state through explored nodes
No knowledge about how far a node node from goal state	Guides search process toward goal
	Prefers states (nodes) that lead close to and not away from goal State

Heuristic Searches - GENERATE-AND-TEST

- The generate-and-test strategy is the simplest of all the approaches. It consists of the following steps:
- **Algorithm: Generate-and-Test**
  1. Generate a possible solution. For some problems. this means generating a particular point in the problem space. For others, it means generating a path from a start state.
  2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
  3. If a solution has been found, quit. Otherwise, return to step 1

Heuristic Searches - GENERATE-AND-TEST



Heuristic Searches - GENERATE-AND-TEST

- Generate-and-test, like depth-first search, requires that complete solutions be generated for testing.
- In its most systematic form, it is only an exhaustive search of the problem space.
- Solutions can also be generated randomly but solution is not guaranteed.
- This approach is what is known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

Heuristic Searches - GENERATE-AND-TEST

Systematic Generate-And-Test

- While generating complete solutions and generating random solutions are the two extremes there exists another approach that lies in between. The approach is that the search process proceeds systematically but some paths that unlikely to lead the solution are not considered. This evaluation is performed by a heuristic function.
- Depth-first search tree with backtracking can be used to implement systematic generate-and-test procedure. As per this procedure, if some intermediate states are likely to appear often in the tree, it would be better to modify that procedure to traverse a graph rather than a tree.

Heuristic Searches - GENERATE-AND-TEST

Generate-And-Test And Planning

- Exhaustive generate-and-test is very useful for simple problems. But for complex problems even heuristic generate-and-test is not very effective technique. But this may be made effective by combining with other techniques in such a way that the space in which to search is restricted. An AI program DENDRAL, for example, uses plan-Generate-and-test technique. First, the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated substructures. Then the generate-and-test procedure uses the lists generated and required to explore only a limited set of structures. Constrained in this way, generate-and-test proved highly effective. A major weakness of planning is that it often produces inaccurate solutions as there is no feedback from the world. But if it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

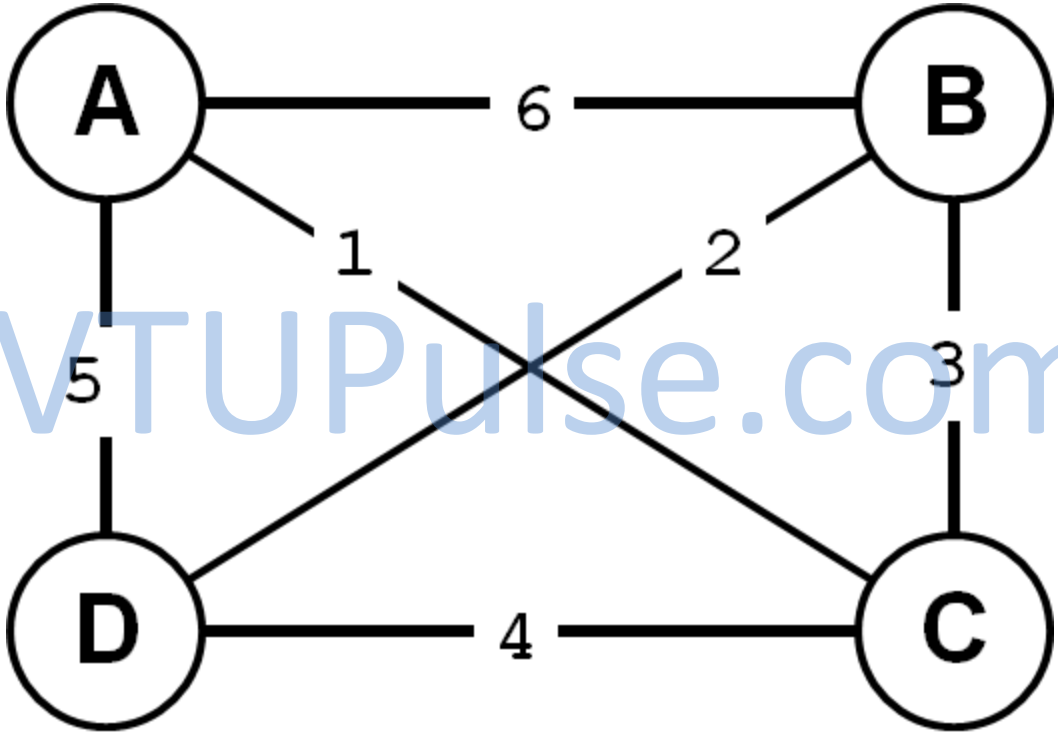
Heuristic Searches - GENERATE-AND-TEST

Example - Traveling Salesman Problem (TSP)

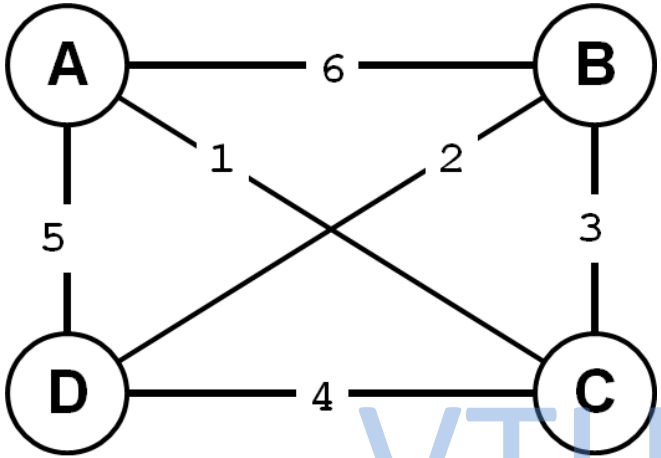
A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

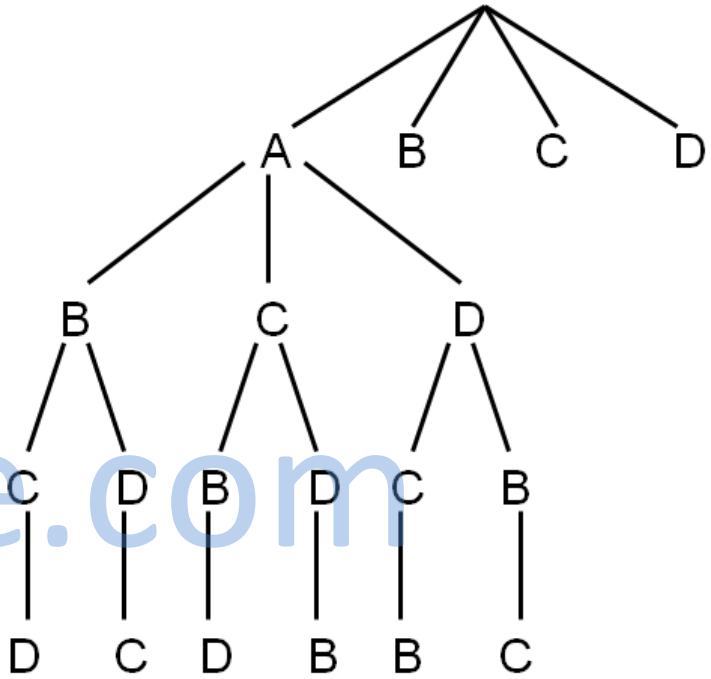
- Traveler needs to visit n cities.
- Know the distance between each pair of cities.
- Want to know the shortest route that visits all the cities once.

Heuristic Searches - GENERATE-AND-TEST



Heuristic Searches - GENERATE-AND-TEST





Search flow with Generate and Test

Search for	Path	Length of Path
1	ABCD	19
2	ABDC	18
3	ACBD	12
4	ACDB	13
5	ADBC	16
Dst…..		

Heuristic Searches - Simplest Hill Climbing

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

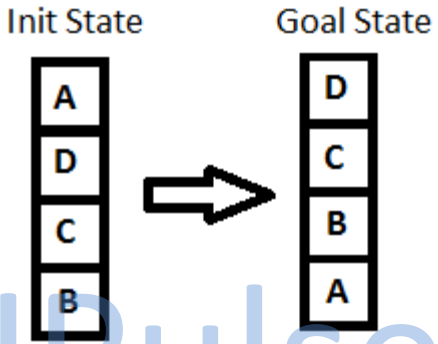
- So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

Heuristic Searches - Simplest Hill Climbing

1. Evaluate the initial state. If it is also goal state then return it, otherwise continue with the initial state as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state
   a) Select an operator that has not yet been applied to the current state and apply it to produce new state
   b) Evaluate the new state
      i. If it 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 as current state
      iii. If it is not better than the current state, then continue in loop.

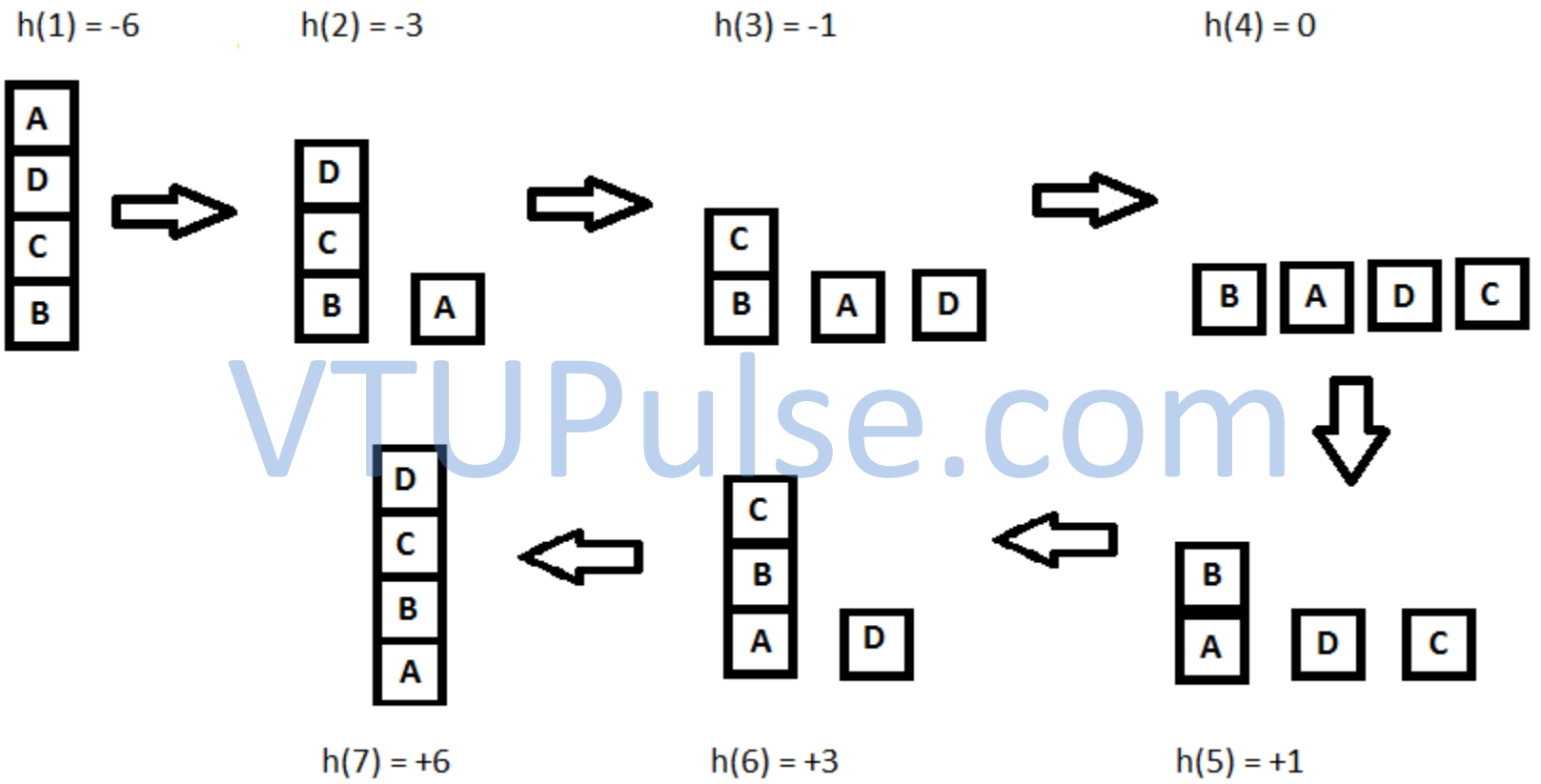
Heuristic Searches - Simplest Hill Climbing

- To understand the concept easily, we will take up a very simple example



- Key point while solving any hill-climbing problem is to choose an appropriate heuristic function.
- Let's define such function *h:*
- ***h(x)* = +1 for all the blocks in the support structure if the block is correctly positioned otherwise -1 for all the blocks in the support structure.**

Heuristic Searches - Simplest Hill Climbing



Heuristic Searches - Steepest-Ascent Hill Climbing

- A variation on simple hill climbing.
- Instead of moving to the *first* state that is *better,* move to the best possible state that is one move away.
- The order of operators does not matter.
- Not just climbing to a better state, climbing up the *steepest* slope.

Heuristic Searches - Steepest-Ascent Hill Climbing

- Considers all the moves from the current state.
- Selects the best one as the next state.
- Basic hill climbing first applies one operator n gets new state. If it is better that becomes current state whereas steepest climbing tests all possible solutions n chooses best

Heuristic Searches - Steepest-Ascent Hill Climbing

Algorithm

1. Evaluate the initial state. If it is also a goal state then return it and quit. Otherwise continue with the initial state as the 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 and generate a new state.
      ii. Evaluate the new state. If it is a goal state, then return it and quit. If not compare it to SUCC. If it is better, then set SUCC to this state. If it is not better, leave SUCC alone.
   c) IF the SUCC is better than current state, then set current state to SUCC.

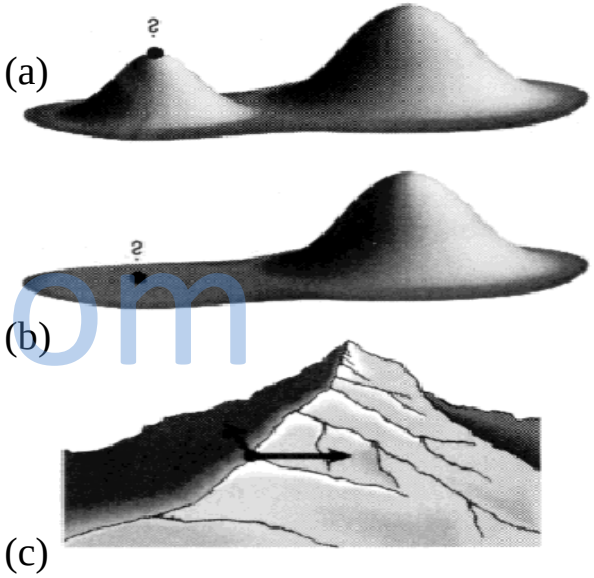
Hill-climbing

This simple policy has three well-known drawbacks:

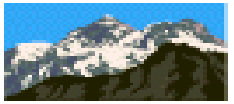
1. **Local Maxima**: a local maximum as opposed to global maximum.

2. **Plateaus**: An area of the search space where evaluation function is flat, thus requiring random walk.

3. **Ridge**: Where there are steep slopes and the search direction is not towards the top but towards the side.

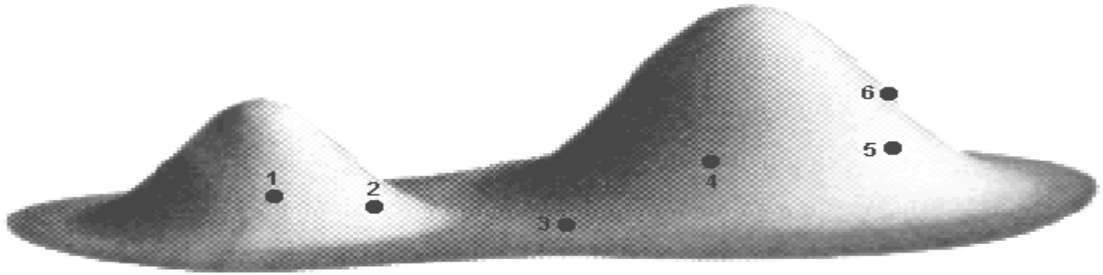


Hill-climbing



- In each of the previous cases (local maxima, plateaus & ridge), the algorithm reaches a point at which no progress is being made.

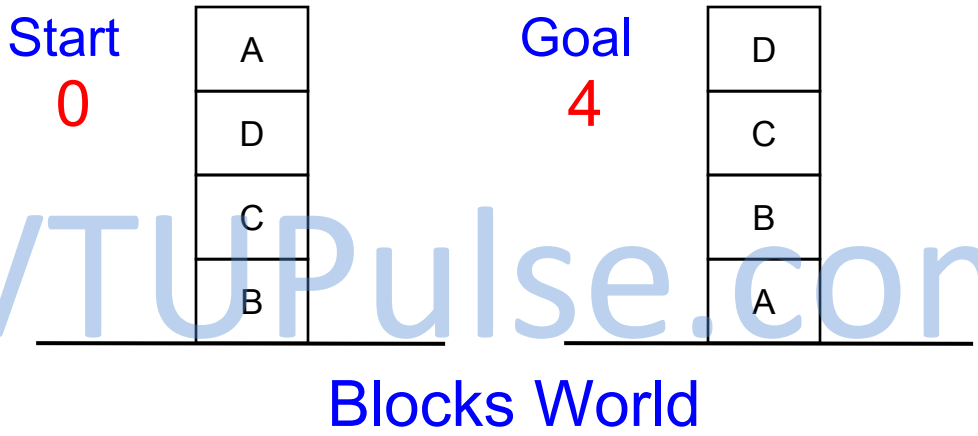
- A solution is to do a **random-restart hill-climbing** - where random initial states are generated, running each until it halts or makes no discernible progress. The best result is then chosen.



Hill Climbing: Disadvantages

- Hill climbing is a local method:
  Decides what to do next by looking only at the “immediate” consequences of its choices.
- Will terminate when at local optimum.
- The order of application of operators can make a big difference.
- Global information might be encoded in heuristic functions.

Hill Climbing: Disadvantages



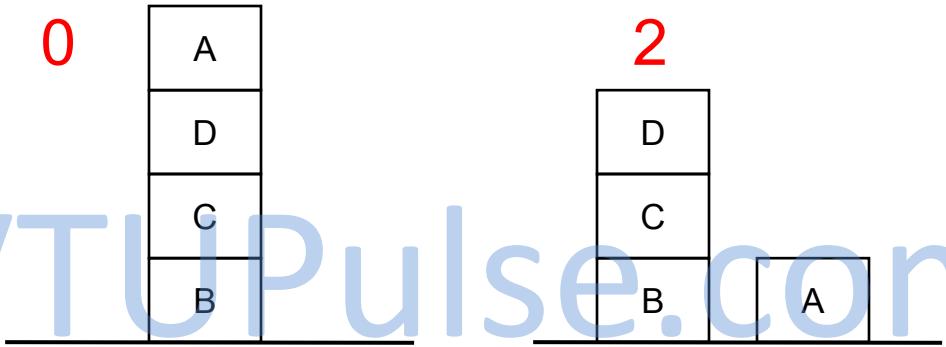
Blocks World

Local heuristic:

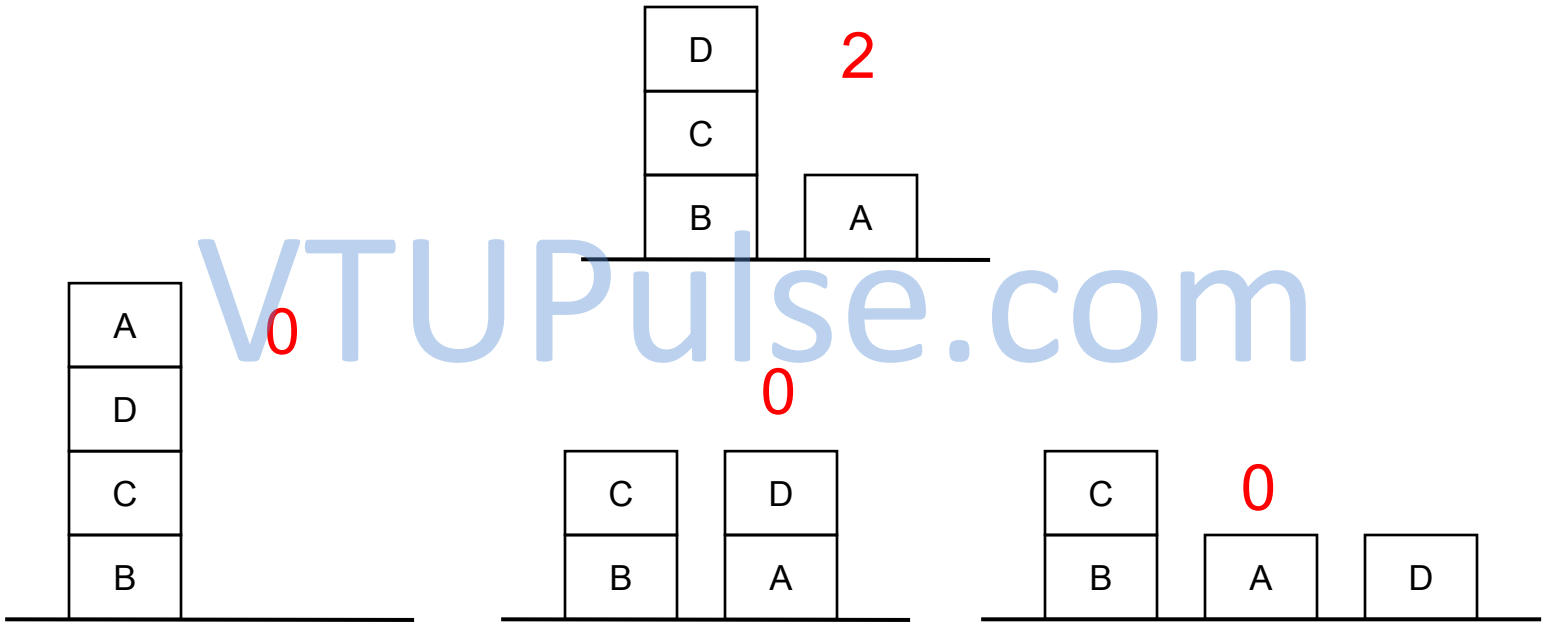
+1 for each block that is resting on the thing it is supposed to be resting on.

–1 for each block that is resting on a wrong thing.

Hill Climbing: Disadvantages



Hill Climbing: Disadvantages



Hill Climbing: Disadvantages

Blocks World

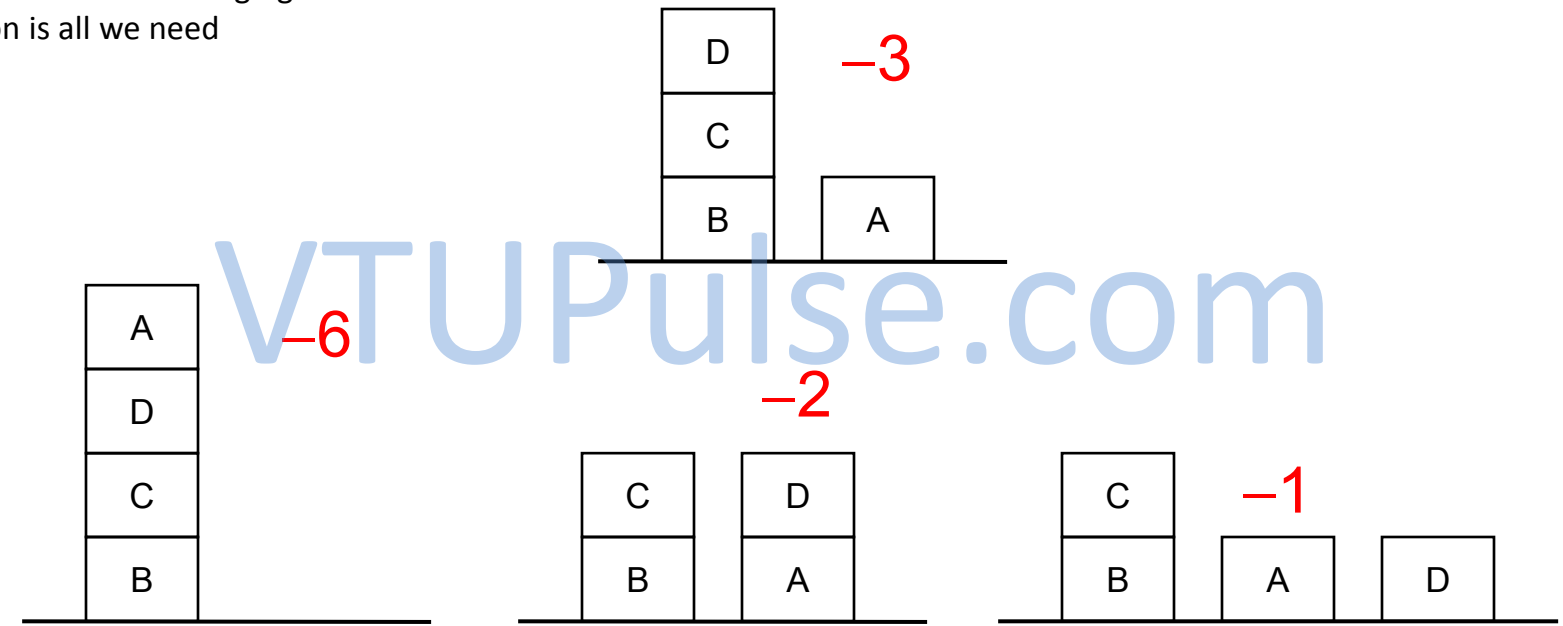
Global heuristic:

For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: –1 to every block in the support structure.

Hill Climbing: Disadvantages

There is no local maximum!
Moral: sometimes changing the heuristic function is all we need



Hill Climbing: Conclusion

- Can be very inefficient in a large, rough problem space.
- Global heuristic may have to pay for computational complexity.
- Often useful when combined with other methods, getting it started right in the right general neighbourhood.

Simulated Annealing

- A variation of hill climbing in which, at the beginning of the process, some downhill moves may be made.

- Idea is to do enough exploration of the whole space early on, so that the final solution is relatively insensitive to the starting state.
- Lowering the chances of getting caught at a local maximum, or plateau, or a ridge.

Simulated Annealing

- Hill climbing with a twist:
  - allow some moves downhill (to worse states)
  - start out allowing large downhill moves (to much worse states) and gradually allow only small downhill moves.

Based on physical process of annealing a metal to get the best (minimal energy) state.

Simulated Annealing

Physical Annealing

- Physical substances are melted and then gradually cooled until some solid state is reached.
- The goal is to produce a minimal-energy final state.
- This process is one of valley descending where the objective function is the energy level

Simulated Annealing

- The rate at which the system is cooled is called annealing schedule
- Annealing schedule: if the temperature is lowered sufficiently slowly, then the goal will be attained(global minimum).
- If cooled rapidly local minimum but not global minimum is reached
- If too slow time is wasted
- Nevertheless, there is some probability for a transition to a higher energy state: $e^{-\Delta E/T}$.

Simulated Annealing

- The search initially jumps around a lot, exploring many regions of the state space.
- The jumping is gradually reduced and the search becomes a simple hill climb (search for local optimum).
- The simulated annealing process lowers the temperature by slow stages until the system ``freezes" and no further changes occur.

Simulated Annealing

Algorithm

1. Evaluate the initial state. If it is goal state then return
2. Initialize BEST-SO-FAR to the current state
3. Set T according to an annealing schedule
4. Loop until a solution is found or there are no new operators left to be applied:
   - Selects and applies a new operator
   - Evaluate the new state:

     goal $\rightarrow$ quit

     compute ΔE = Val(current state) – Val(new state)

     $\Delta E < 0 \rightarrow$ new current state-if is is not goal state but better than the current state then make it current. Set this state as BEST-SO-FAR

Simulated Annealing

Algorithm

else → If it is not better than the current state make it the current state with probability $p=e^{-\Delta E/T}$.

Randomly generate number between [0 1], if p>number generated, move is accepted else do nothing

- Revise T as necessary according to annealing schedule

5. Return BEST-SO-FAR as answer

Heuristic Searches - Best-First Search

- Combines the advantages of both DFS and BFS into a single method.
- DFS is good because it allows a solution to be found without all competing branches having to be expanded.
- BFS is good because it does not get branches on dead end paths.
- One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.

Heuristic Searches - Best-First Search

- At each step of the BFS search process, we select the most promising of the nodes we have generated so far.
- This is done by applying an appropriate heuristic function to each of them.
- We then expand the chosen node by using the rules to generate its successors

Heuristic Searches - Best-First Search

- It proceeds in steps, expanding one node at each step, until it generates a node that corresponds to a goal state.
- At each step, it picks the most promising of the nodes that have so far been generated but not expanded.
- It generates the successors of the chosen node, applies the heuristic function to them, and adds them to the list of open nodes, after checking to see if any of them have been generated before.
- By doing this check, we can guarantee that each node only appears once in the graph, although many nodes may point to it as a successor.

Heuristic Searches - Best-First Search

- To implement such a graph-search procedure, we will need to use two lists of nodes:
  - OPEN — nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined (i.e., had their successors generated).
  - CLOSED — nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated, we need to check whether it has been generated before.

Heuristic Searches - Best-First Search

- We will also need a heuristic function that estimates the merits of each node we generate. This will enable the algorithm to search more promising paths first.
- Call this function f’.
- For many applications, it is convenient to define this function as the sum of two components that we call g and h'.
- The function g is a measure of the cost of getting from the initial state to the current node.

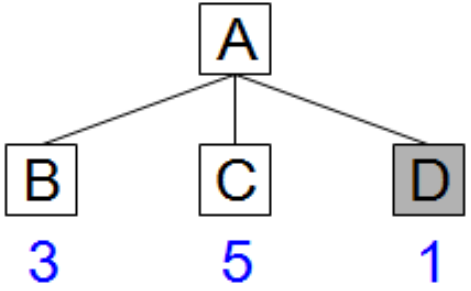
Heuristic Searches - Best-First Search

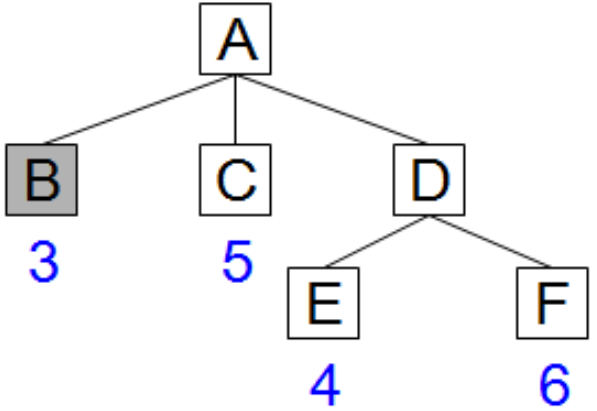
- Note that g is not an estimate of anything; it is known to be the exact sum of the costs of applying each of the rules that were applied along the best path to the node.
- The combined function then represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the current node.
- If more than one path generated the node, then the algorithm will record the best one.
- Note that because g and h' must be added, it is important that h'

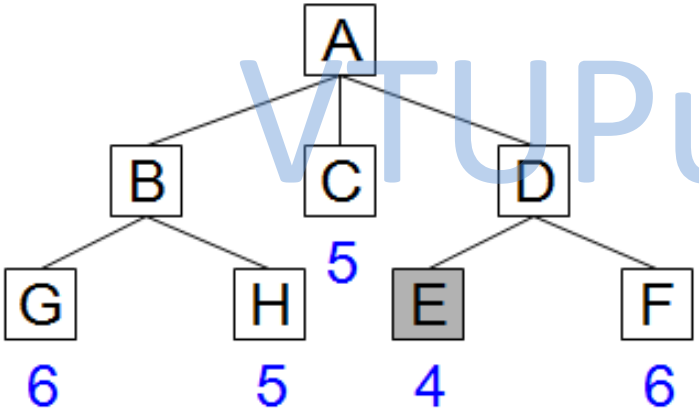
Heuristic Searches - Best-First Search

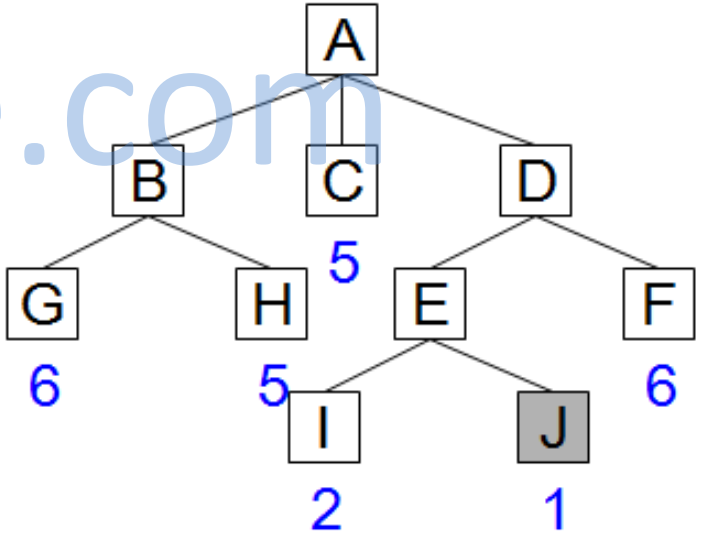
Algorithm: Best-First Search

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left on OPEN do:
   a) Pick them best node on OPEN.
   b) Generate its successors.
   c) For each successor do:
      i. if it has not been generated before, evaluate it, add it to OPEN, and record its parent.
      ii. If it has been generated before, change the parent if this new path is better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.







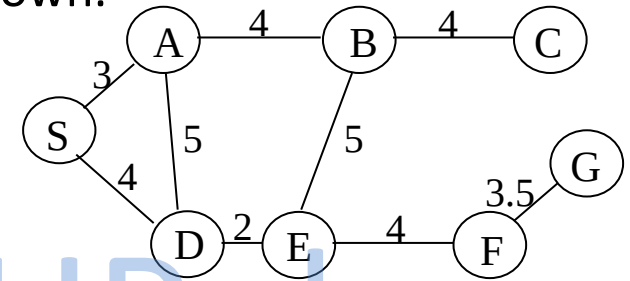


Heuristic Searches – A* Search

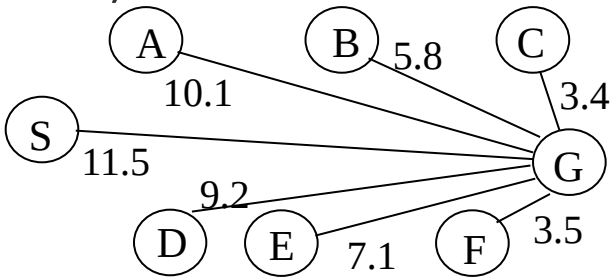
1. Start with OPEN containing only the initial node. Set that node's **g** value to **0**, its **h'** value to whatever it is, and its **f'** value to **h' + 0**, or **h'**. Set CLOSED to the empty list.
2. Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN. report failure. Otherwise, pick the node on OPEN with the lowest **f'** value. Call it BESTNODE. Remove it from OPEN. Place it on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution. Otherwise, generate the successors of BESTNODE but do not set BESTNODE to point to them yet. For each such SUCCESSOR, do the following:
   a) Set SUCCESSOR to point back to BESTNODE. These backwards links will make it possible to recover the path once a solution is found.
   b) Compute g(SUCCESSOR) = g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.
   c) if SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)

A* Algorithm Solved Example

A simple search problem: S is the start node, G is the goal node, the real distances are shown.

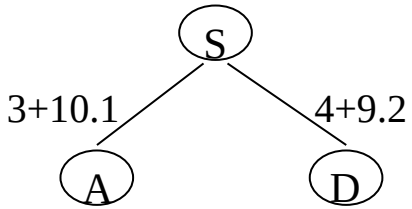


A lower-bound estimate of the distance to G could be as follows (note that we do not need it for F):



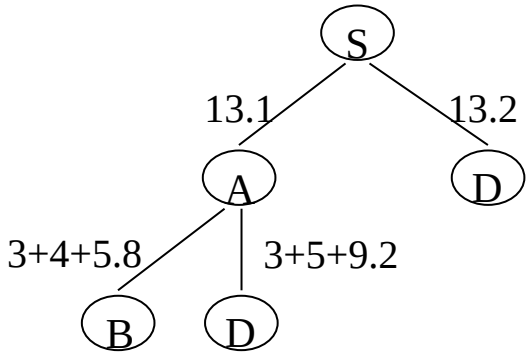
A* Algorithm

A*, step 1



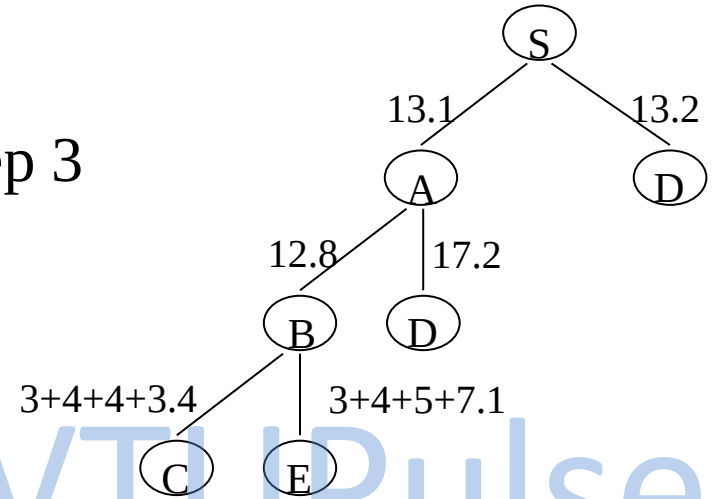
A* Algorithm

A*, step 2



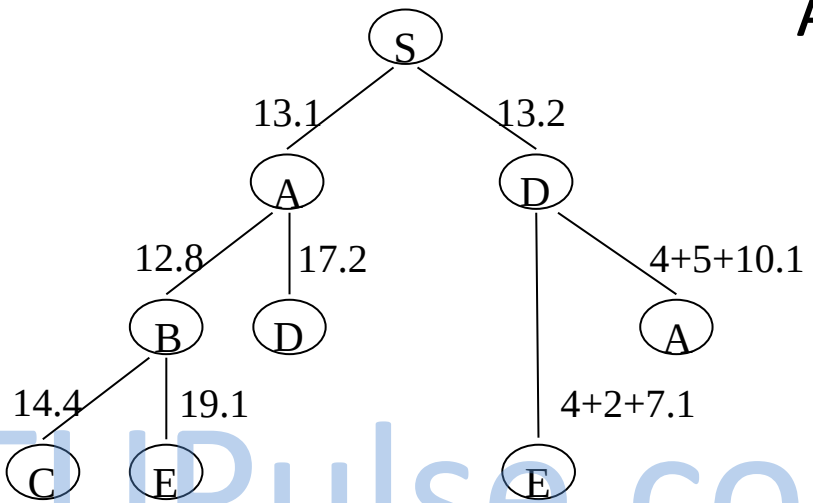
A* Algorithm

A*, step 3



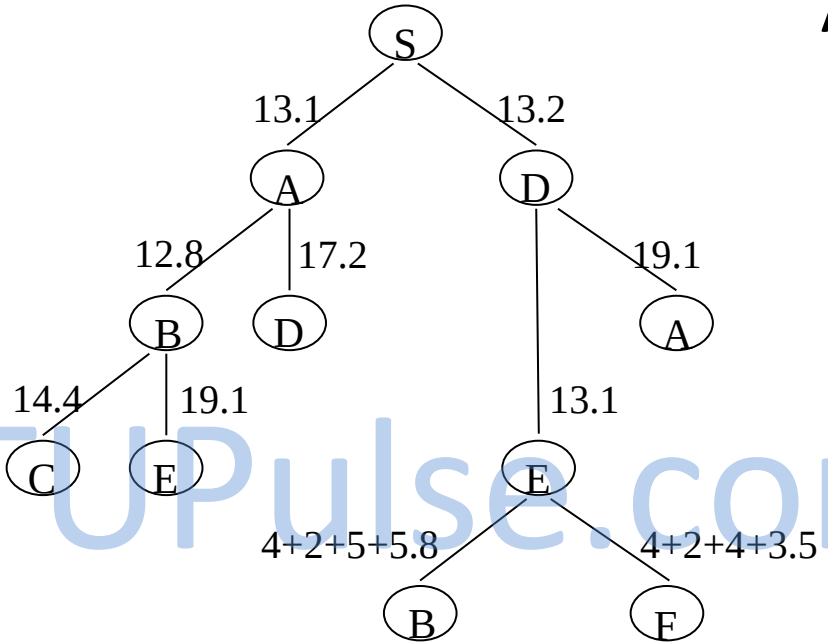
A* Algorithm

A*, step 4



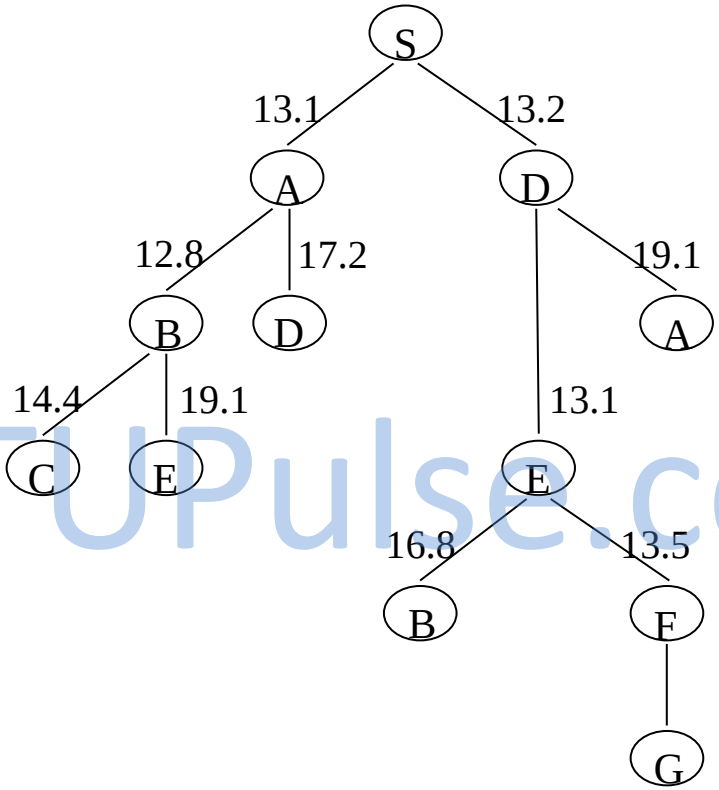
A* Algorithm

A*, step 5

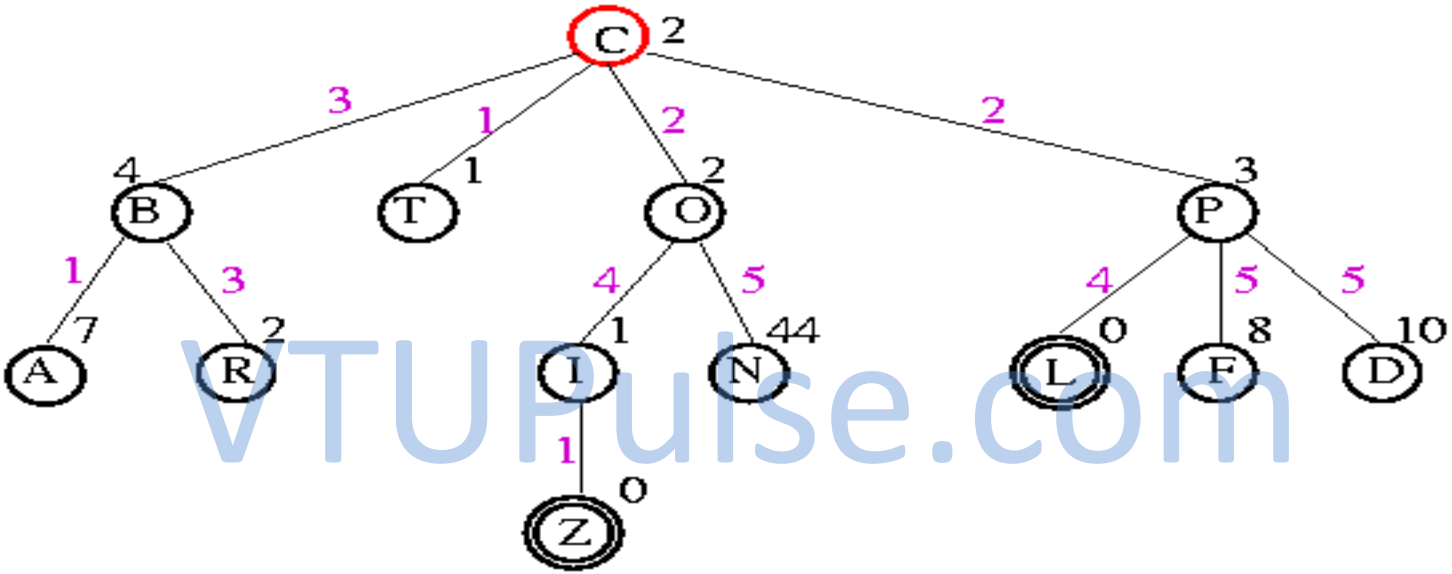


A* Algorithm

A*, step 6

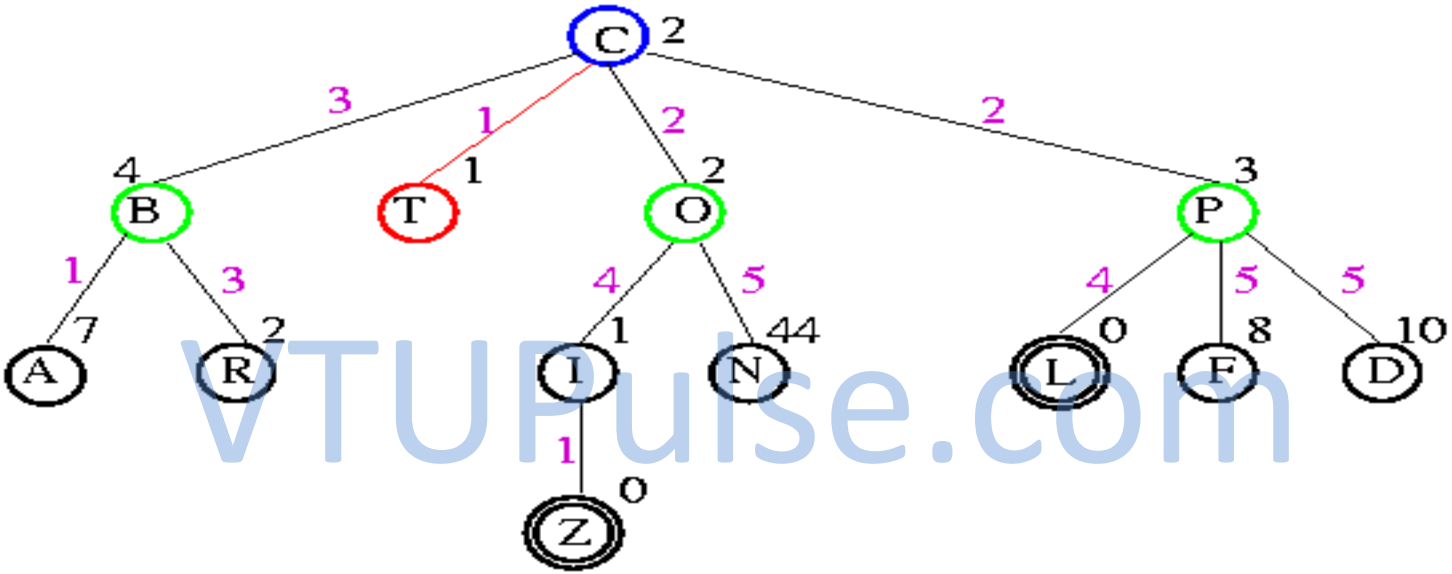


Example



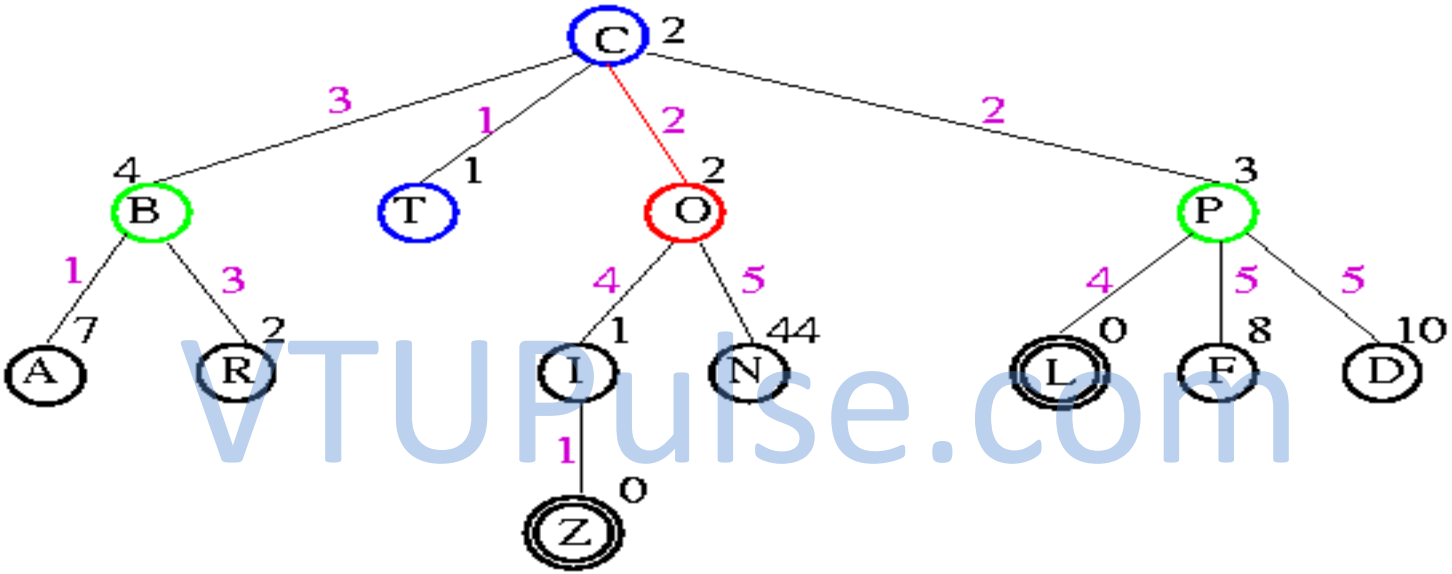
Open List = C (0+2=2)

Example



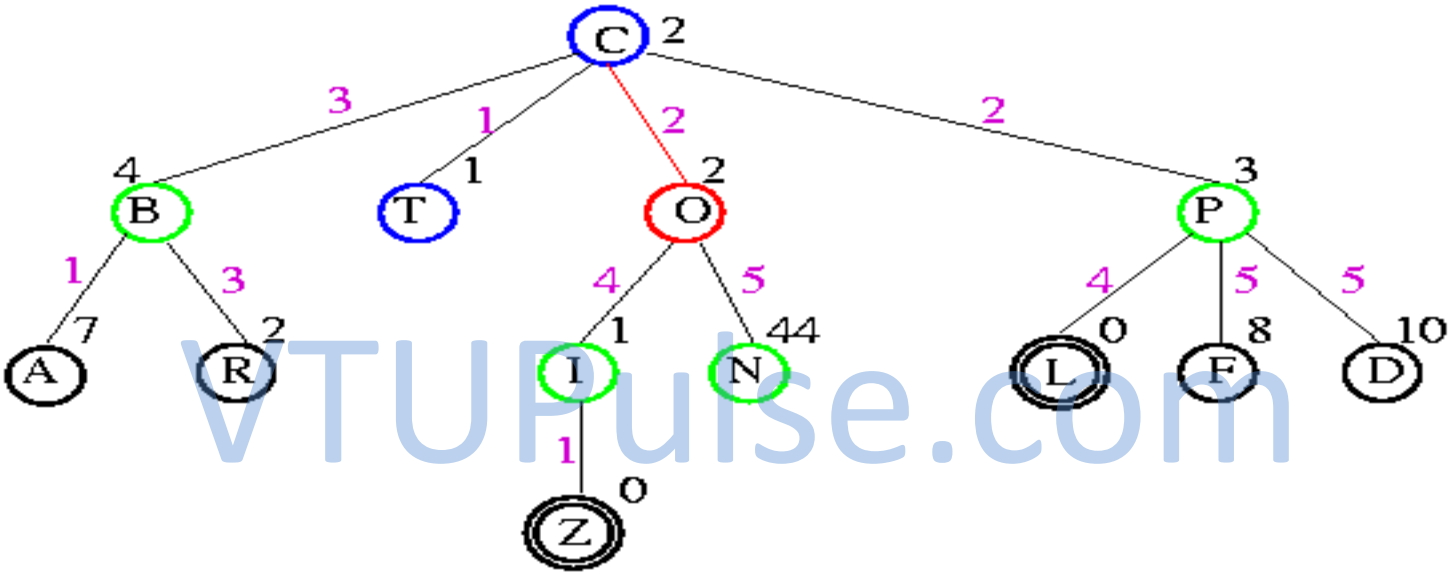
Open List = T (1+1=2), O (2+2=4), P (2+3=5), B(3+4=7)

Example



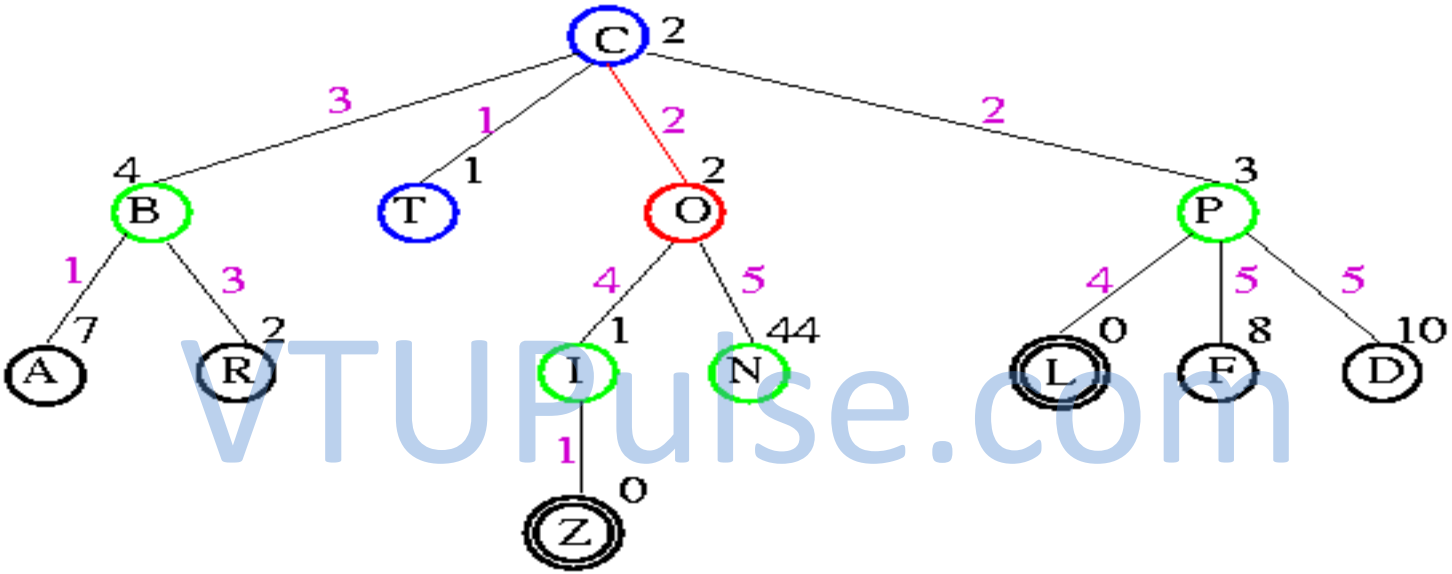
Open List = O (2+2=4), P (2+3=5), B(3+4=7)

Example



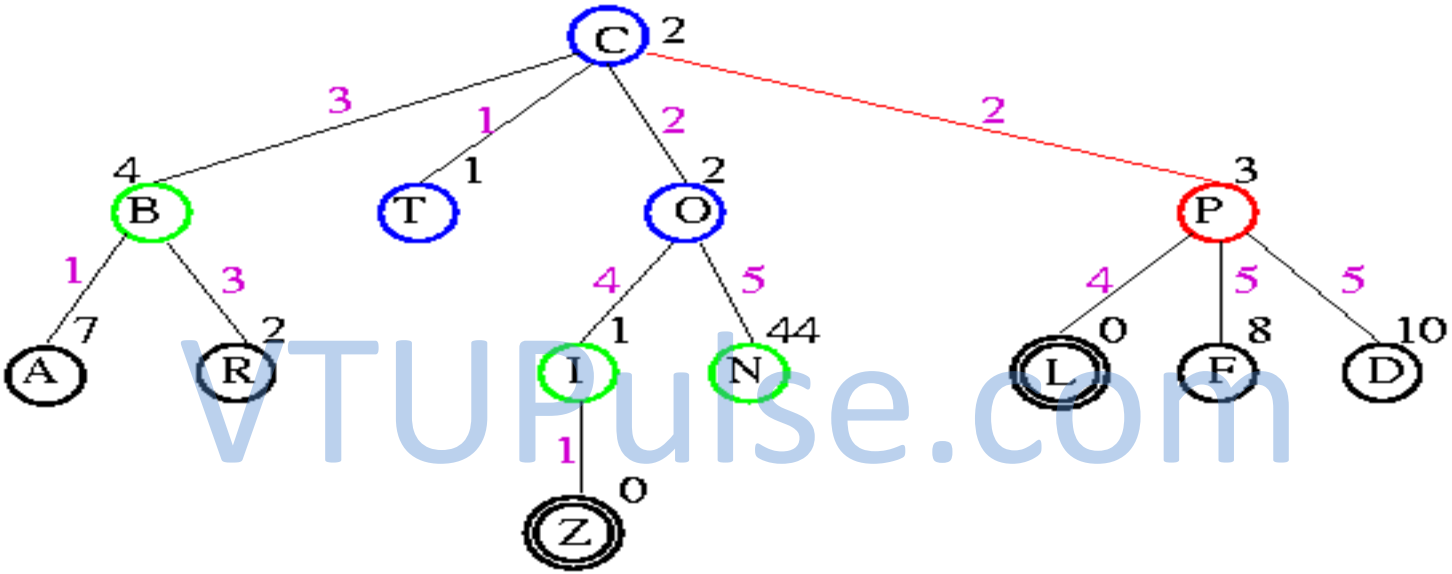
Open List = O (2+2=4), P (2+3=5), B(3+4=7)

Example



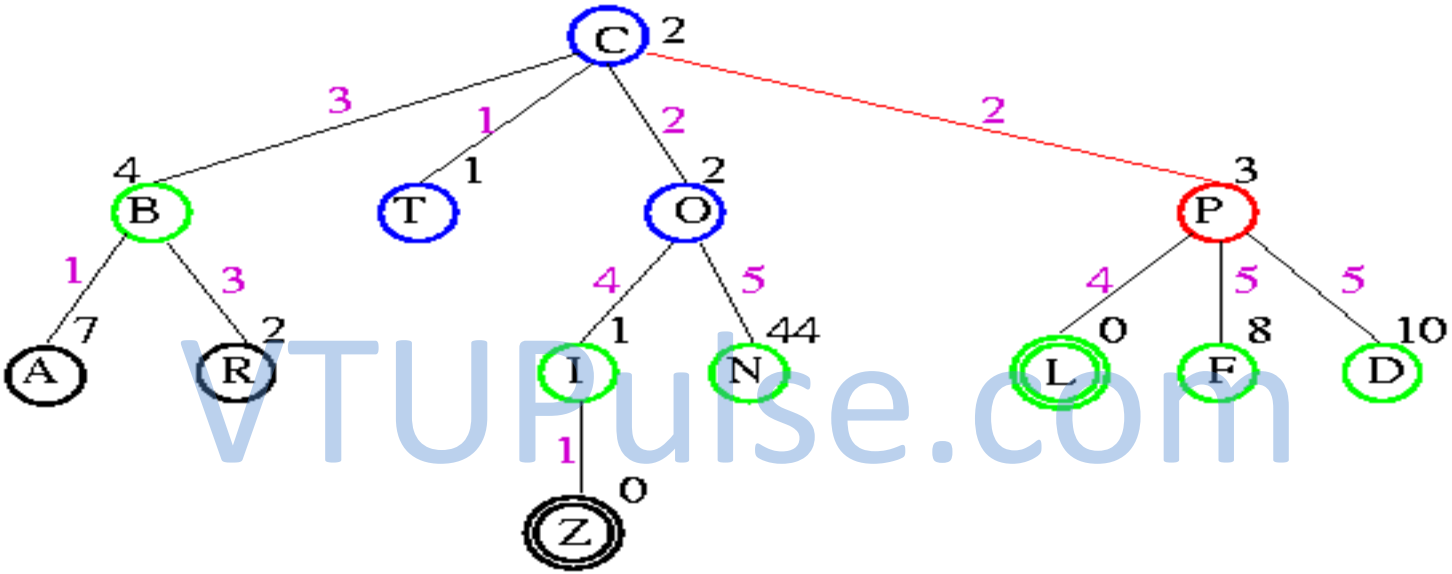
Open List = O (2+2=4), P (2+3=5), B(3+4=7)
I (6+1=7), N (7+44=51)

Example



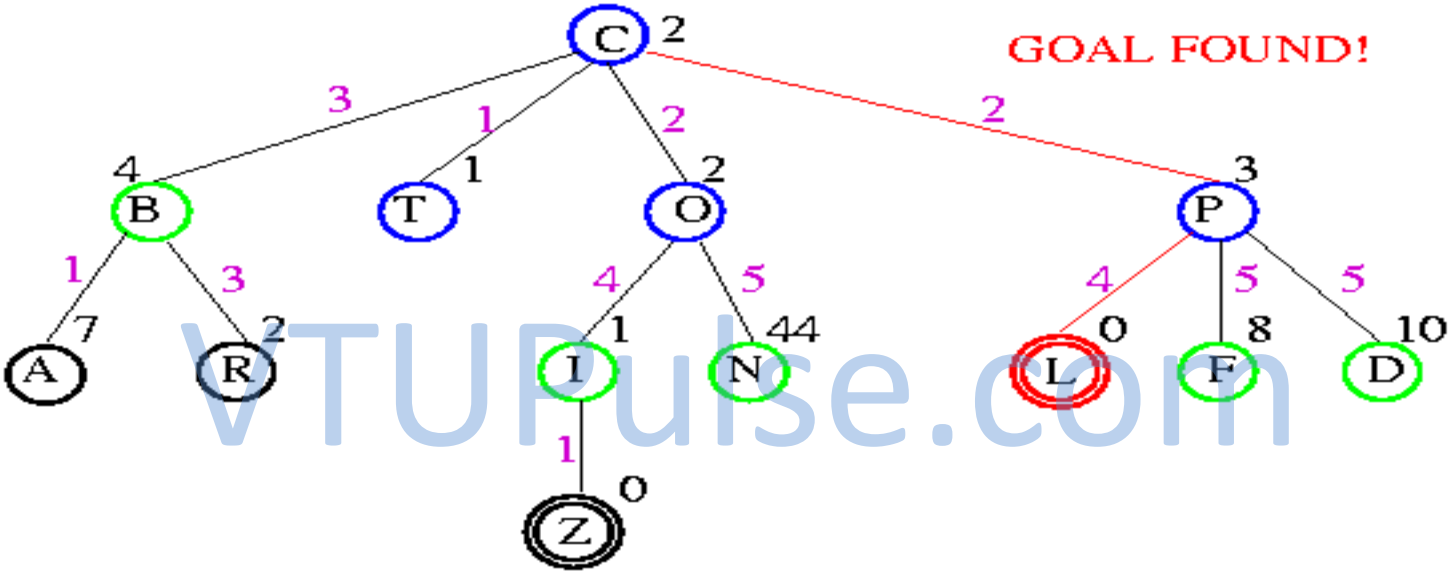
Open List = P (2+3=5), B(3+4=7)
I (6+1=7), N (7+44=51)

Example



Open List = P (2+3=5), L (6+0=6), B (3+4=7)
I (6+1=7), F (7+8=15), D (7+10=17), N (7+44=51)

Example



Open List = L (6+0=6), B (3+4=7)
I (6+1=7), F (7+8=15), D (7+10=17), N (7+44=51)

Problem Reduction

- Sometimes problems only seem hard to solve.
- A hard problem may be one that can be reduced to a number of simple problems and, when each of the simple problems is solved, then the hard problem has been solved.
- This is the basic intuition behind the method of problem reduction.

Problem Reduction

- If we are looking for a sequence of actions to achieve some goal, then one way to do it is to use state-space search, where each node in your search space is a state of the world, and you are searching for a sequence of actions that get you from an initial state to a final state.

Problem Reduction

- Another way is to consider the different ways that the goal state can be decomposed into simpler subgoals.
- For example, when planning a trip to Bangalore you probably don't want to search through all the possible sequences of actions that might get you to Bangalore .
- You're more likely to decompose the problem into simpler ones - such as getting to the station, then getting a train to Bangalore .

Problem Reduction

- There may be more than one possible way of decomposing the problem - an alternative strategy might be to get to the airport, fly to Mysore, and get the train from Mysore into Bangalore.
- These different possible plans would have different costs (and benefits) associated with them, and you might have to choose the best plan.

Problem Reduction

- The simple state-space search techniques could be represented using a tree where each successor node represents an *alternative* action to be taken.

- The graph structure being searched is referred to as an *OR* graph.
- In *OR* graph we want to find a single path to a goal.

Problem Reduction

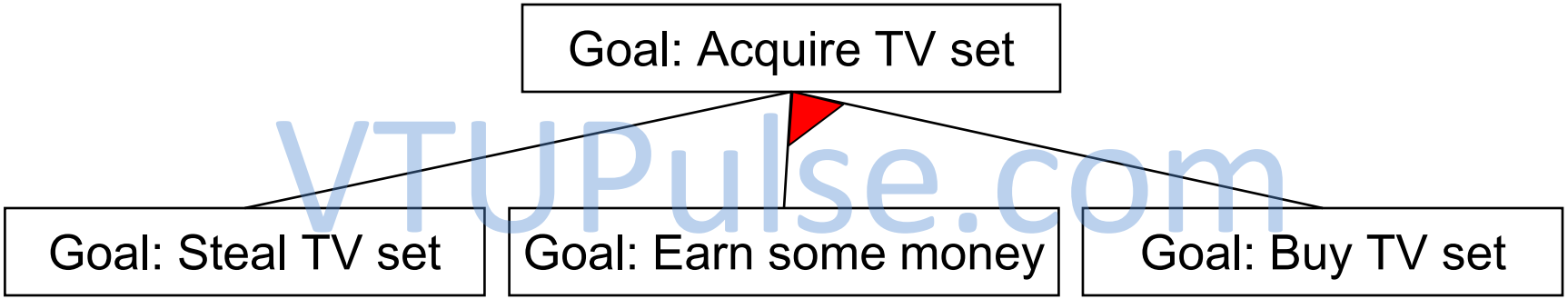
- This is due to the fact that we will know how to get to from a node to a goal state if we can discover how to get from that node to a goal state along any one of the branches leaving it.
- To represent problem reduction techniques we need to use an AND-OR graph/tree.

Problem Reduction

- Breaking a problem down into smaller sub-problems (or **sub-goals**).
- Can be represented using **goal trees** (or **and-or trees**).
- Nodes in the tree represent sub-problems.
- The root node represents the overall problem.
- Some nodes are **and nodes**, meaning **all** their children must be solved.

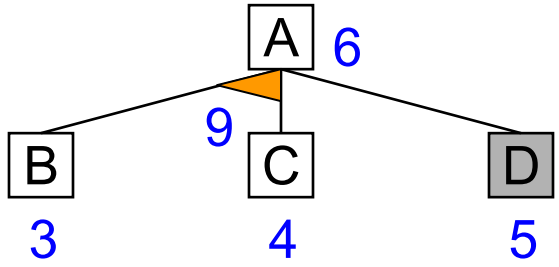
Problem Reduction

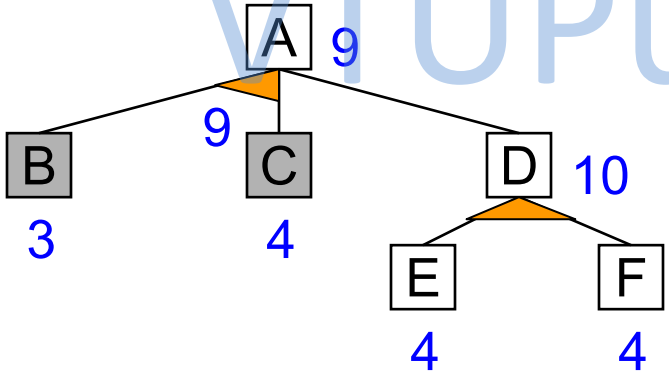
- Algorithm AO* (Martelli & Montanari 1973, Nilsson 1980)

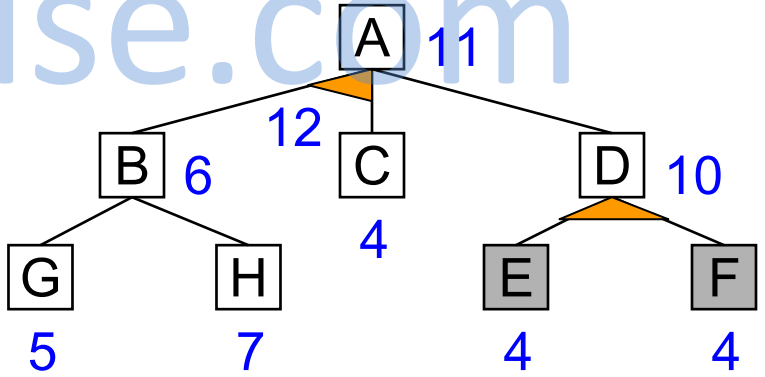


AND-OR Graphs

Problem Reduction

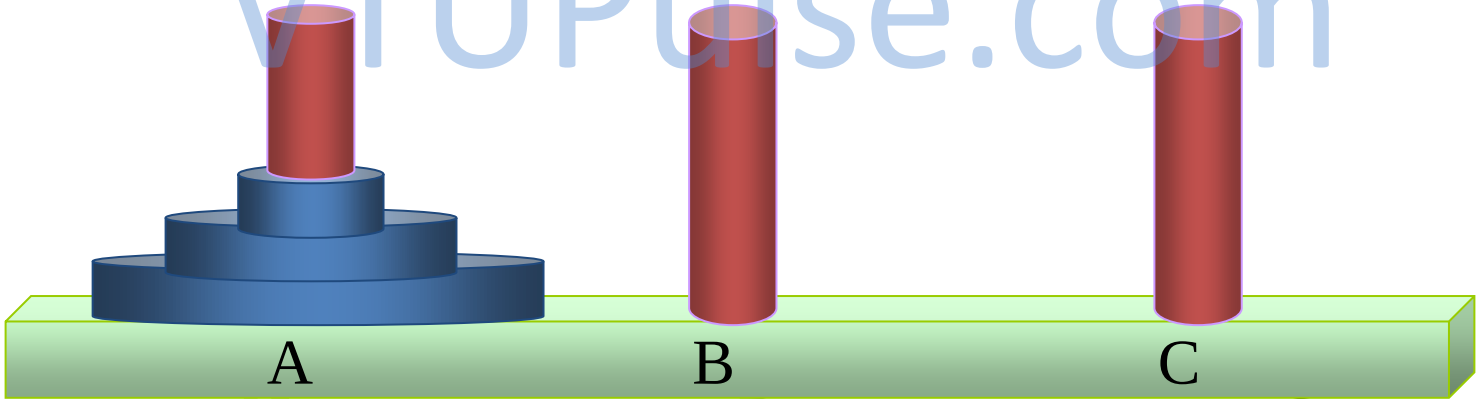






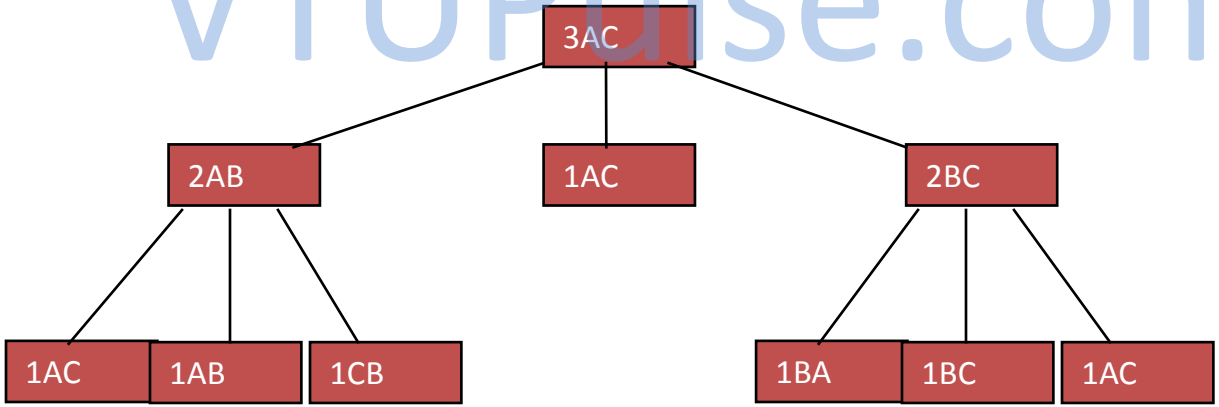
Problem Reduction

- In a problem reduction space, the nodes represent problems to be solved or goals to be achieved, and the edges represent the decomposition of the problem into subproblems.
- This is best illustrated by the example of the Towers of Hanoi problem.

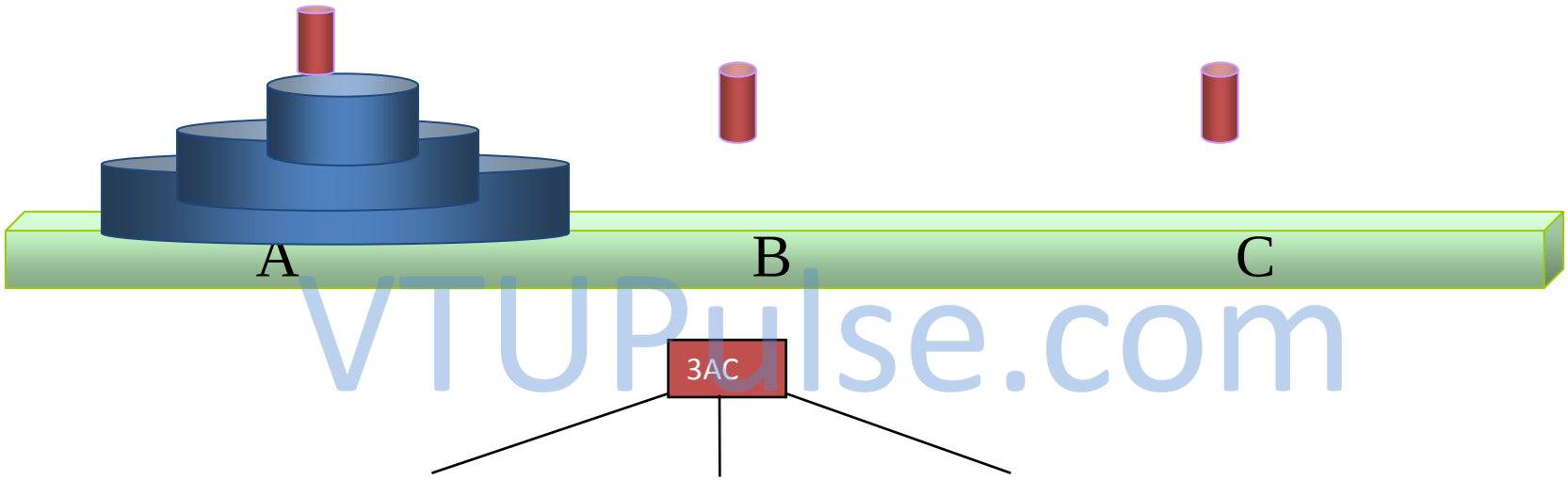


Problem Reduction Space

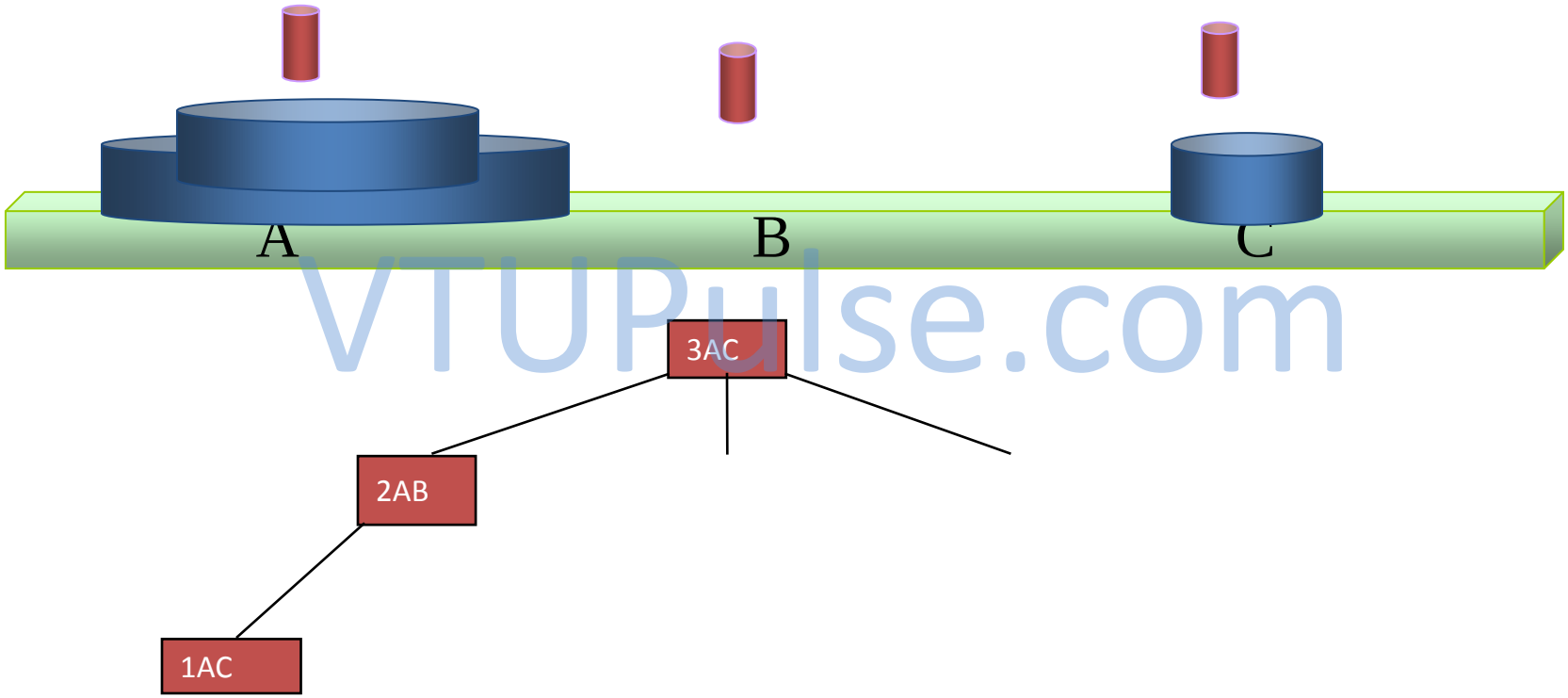
- The root node, labeled “3AC” represents the original problem of transferring all 3 disks from peg A to peg C.
- The goal can be decomposed into three subgoals: 2AB, 1AC, 2BC. In order to achieve the goal, all 3 subgoals must be achieved.



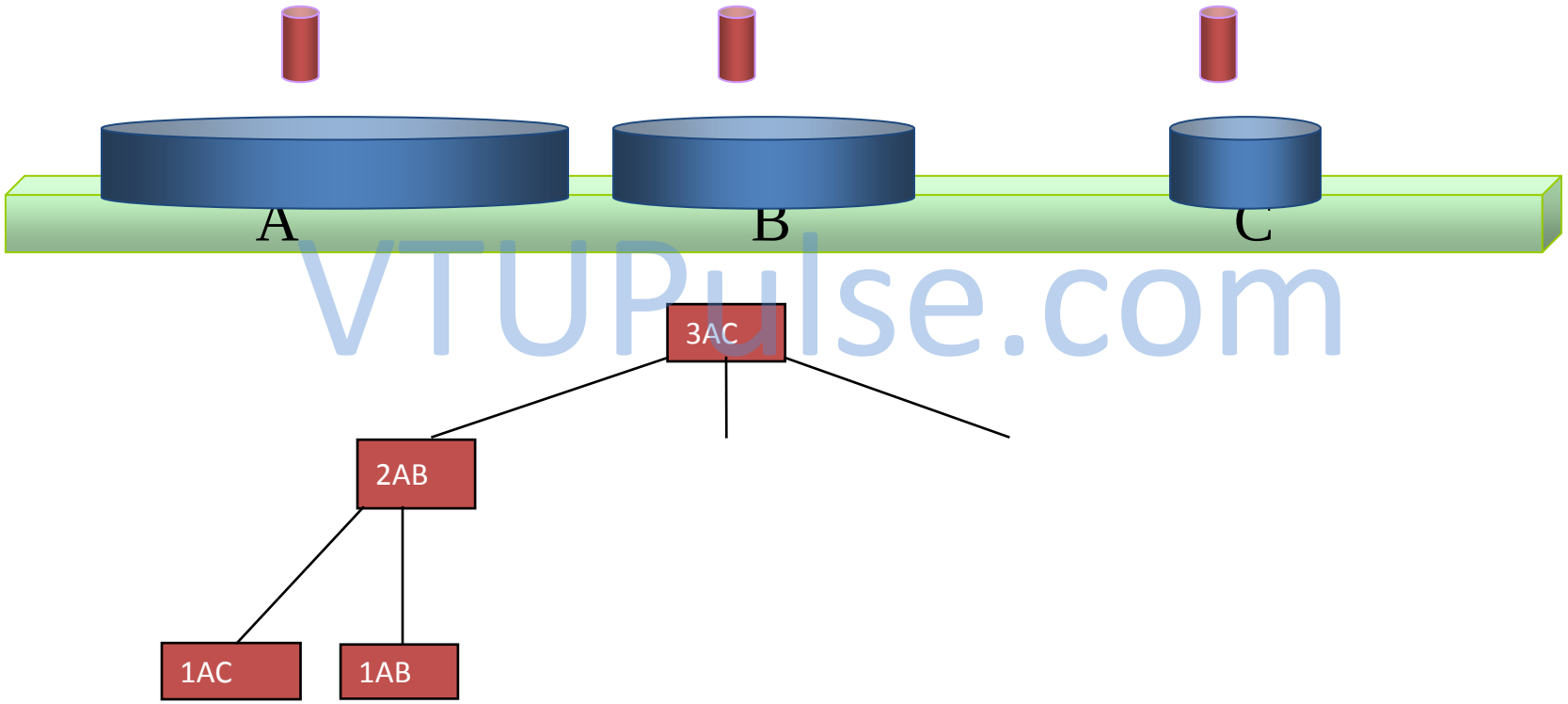
Problem Reduction Space



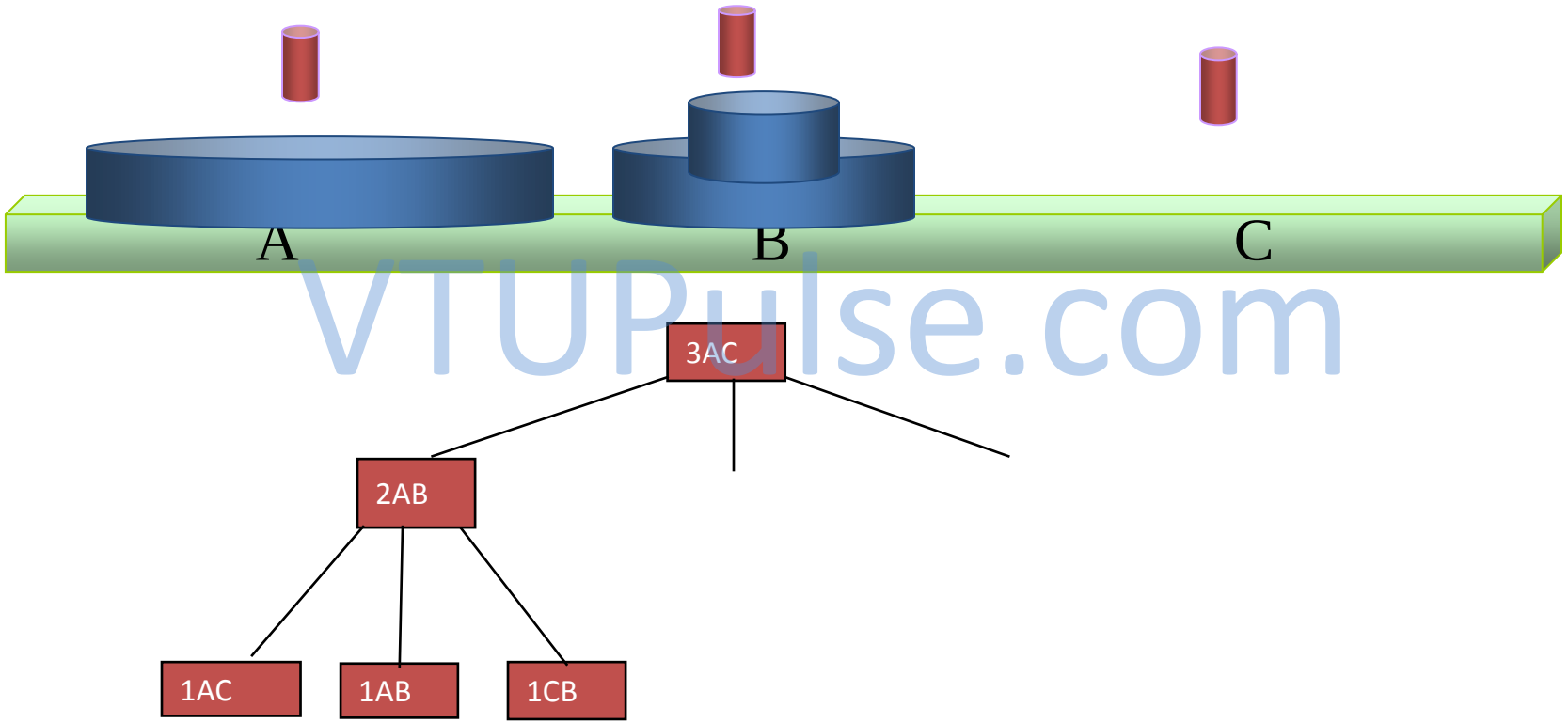
Problem Reduction Space



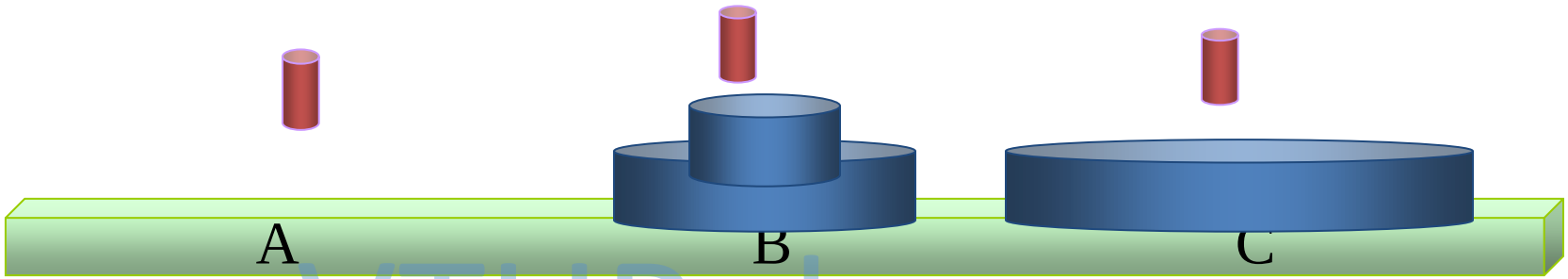
Problem Reduction Space

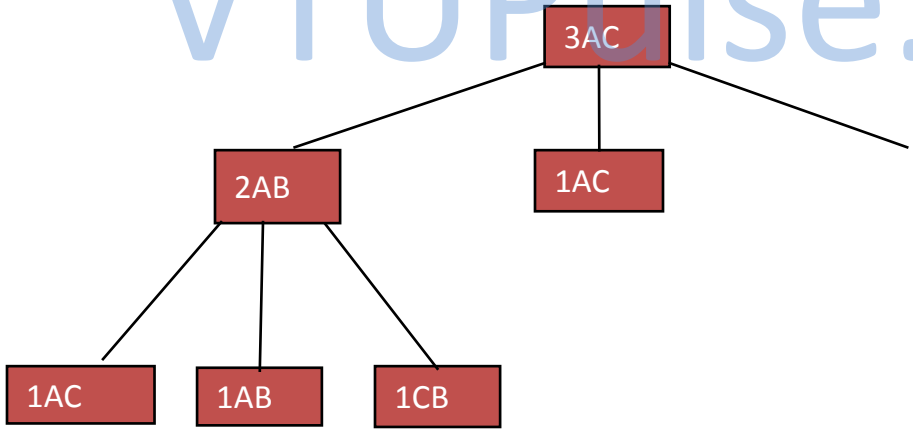


Problem Reduction Space

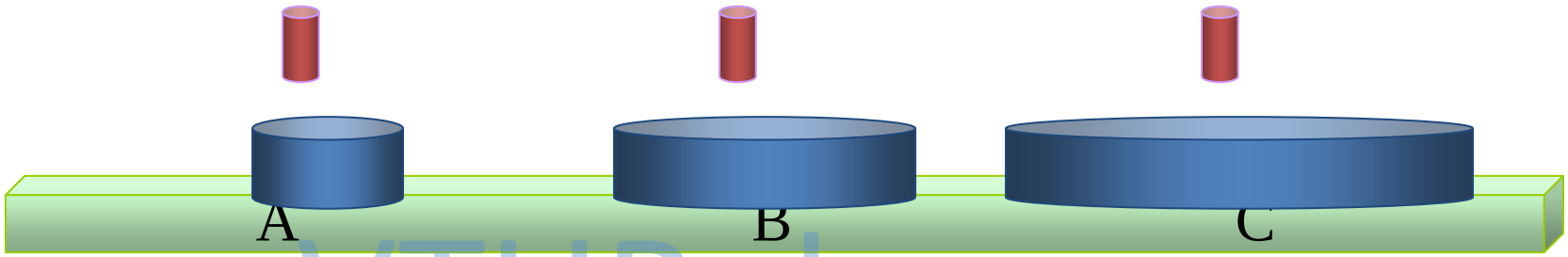


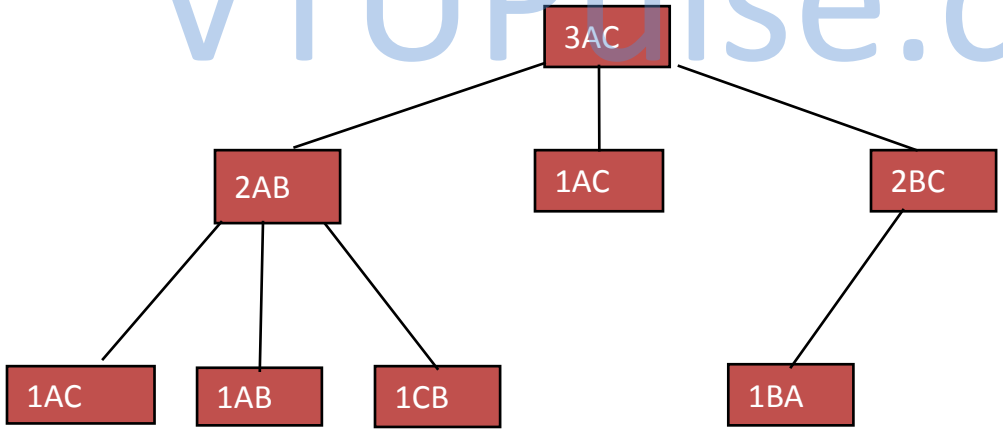
Problem Reduction Space



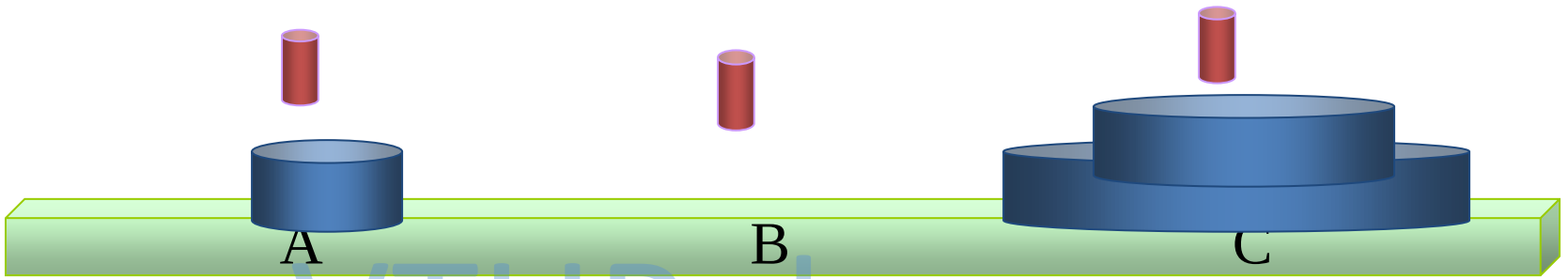


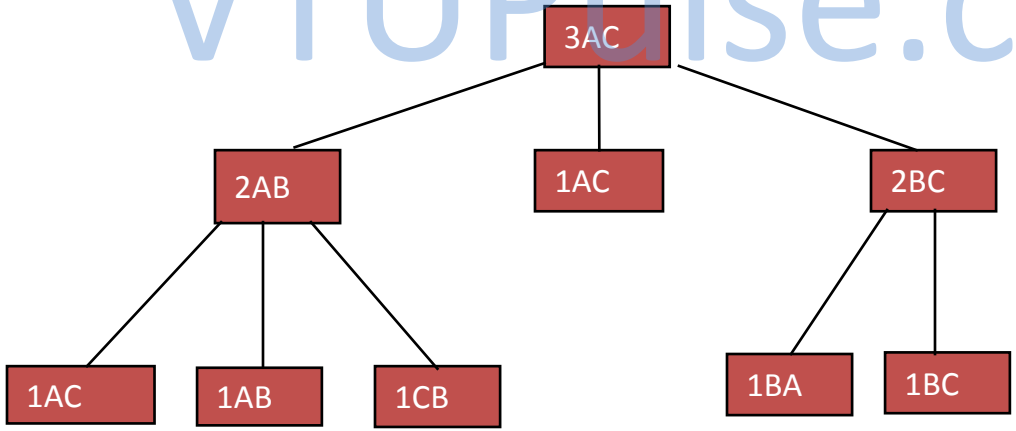
Problem Reduction Space



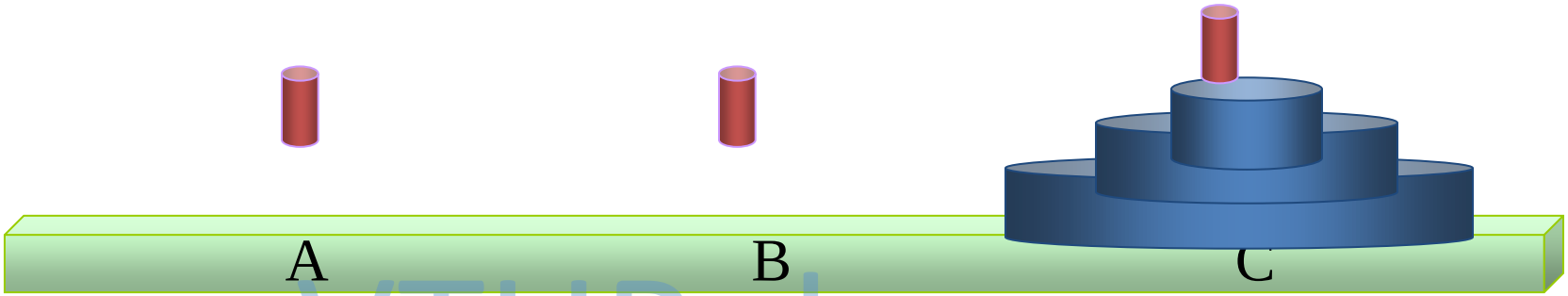


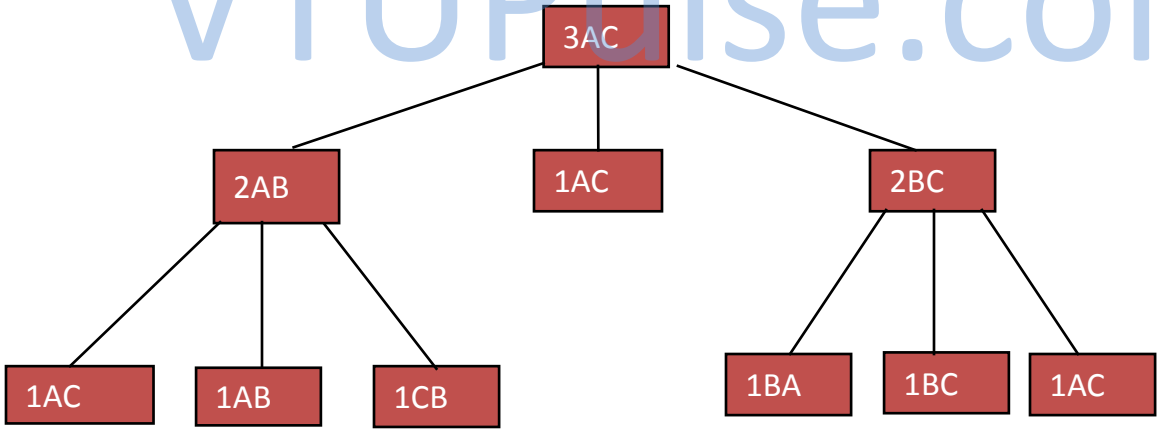
Problem Reduction Space





Problem Reduction Space

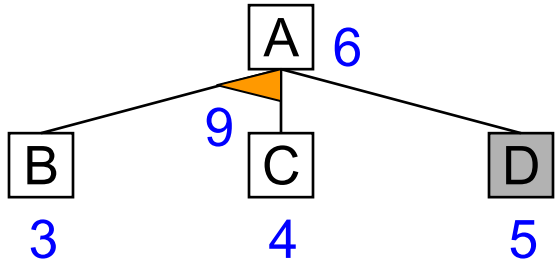


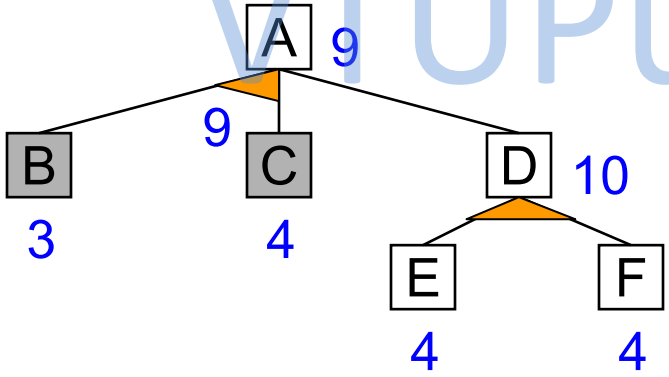


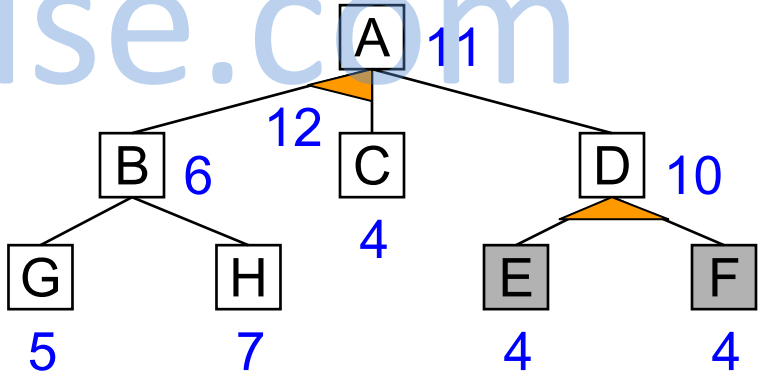
Problem Reduction or AO* Algorithm

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:
    1. Traverse the graph, starting at the initial node following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded or labeled solved.
    2. Pick up one of those unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise add the successors to the graph and each of this compute f' (use only h' and ignore g).
    3. If f' of any node is "0", mark the node as SOLVED. Change the value of f' for the newly created node to reflect its successors by back propagation. Wherever possible use the most promising routes and if a node is marked as SOLVED then mark the parent node as SOLVED.

Problem Reduction







AO* (Elaborated) Algorithm

1. Let GRAPH consist only of the node representing the initial state. (Call this node min:) Compute h'(INIT)
2. Until INIT is labeled SOLVED or until INIT's h' value becomes greater than FUTILITY. repeat the following procedure:
   a) Trace the labeled arcs from INIT and select for expansion one of the as yet unexpanded nodes that occurs on this path. Call the selected node NODE.
   b) Generate the successors of NODE. If there arc none, then assign FUTILITY as the h' value of NODE. This is equivalent to saying that NODE is not solvable. If there are successors, then for each one (called SUCCESSOR) that is not also an ancestor of NODE do the following:
      i. Add SUCCESSOR to GRAPH.
      ii. If SUCCESSOR is a terminal node, label it SOLVED and assign it an It' value of 0.
      iii. If SUCCESSOR is not a terminal node, compute its h' value.

AO* (Elaborated) Algorithm

c) Propagate the newly discovered information up the graph by doing the following: Let S be a set of nodes that have been labeled SOLVED or whose h' values have been changed and so need to have values propagated back to their parents. Initialize 5 to NODE. Until S is empty, repeat the, following procedure:

i. If possible, select from S a node none of whose descendants in GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT. and remove it from S.

ii. Compute the cost of each of the arcs emerging from CURRENT: The cost of each arc is equal to the sum of the h' values of each of the nodes at the end of the arc plus whatever the cost of the arc itself is. Assign as CURRENT'S new le value the minimum of the costs just computed for the arcs emerging from it.

iii. Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

iv. Mark CURRENT SOLVED if all of the nodes connected to it through the new labeled arc have been labeled SOLVED.

v. If CURRENT has been labeled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of CURRENT to S.

Constraint Satisfaction

- Many AI problems can be viewed as problems of constraint satisfaction.

Cryptarithmetic puzzle:


```
  SEND
+ MORE
------
 MONEY
```

Constraint Satisfaction

- As compared with a straightforward search procedure, viewing a problem as one of constraint satisfaction can reduce substantially the amount of search.

- Operates in a space of constraint sets.
- Initial state contains the original constraints given in the problem.
- A goal state is any state that has been constrained “enough”.

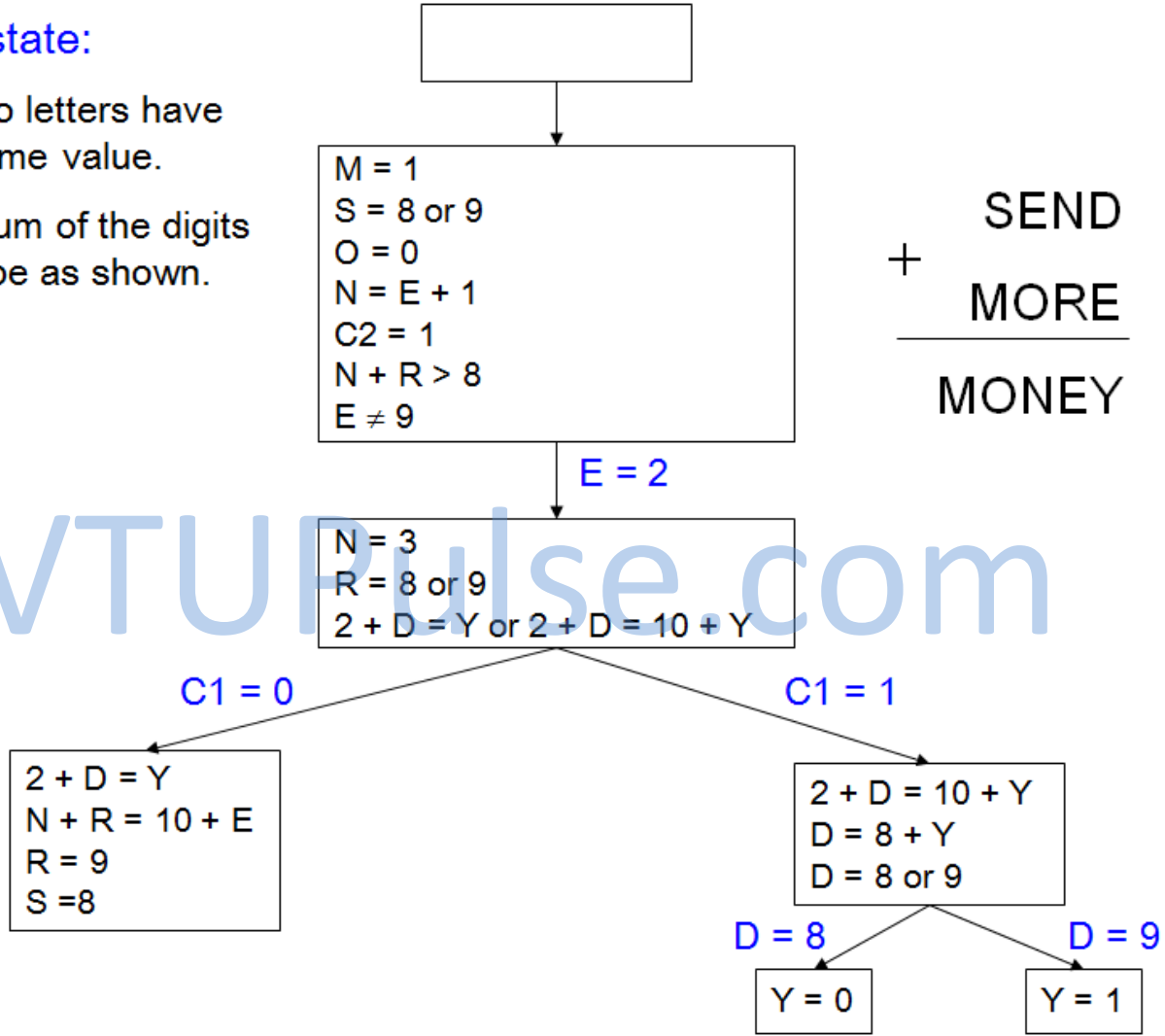
Constraint Satisfaction

Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Initial state:

- No two letters have the same value.
- The sum of the digits must be as shown.



SEND
\+ MORE
MONEY

Constraint Satisfaction

1) M = 1, because M =/= 0 and …

- the carry over of the addition of two
- digits (plus previous carry) is at most 1.

2) O = 0. Because M=1 and we have to have a carry to the next column. S + 1 + C3 is either 10 or 11. So, O equals 0 or 1. 1 is taken. So, O = 0.

3) S = 9. There cannot be a carry to the 4th column (if there were, N would also have to be 0 or 1. Already taken.). So, S = 9.

Constraint Satisfaction

4. If there were no carry in column 3 then E = N, which is impossible. Therefore there is a carry and N = E + 1.
5. If there were no carry in column 2, then (N + R) mod 10 = E, and N = E + 1, so (E + 1 + R) mod 10 = E which means (1 + R) mod 10 = 0, so R = 9. But S = 9, so there must be a carry in column 2 so **R = 8**.
6. To produce a carry in column 2, we must have D + E = 10 + Y.
7. Y is at least 2 so D + E is at least 12.
8. The only two pairs of available numbers that sum to at least 12 are (5,7) and (6,7) so either E = 7 or D = 7.
9. Since N = E + 1, E can't be 7 because then N = 8 = R so **D = 7**.
10. E can't be 6 because then N = 7 = D so **E = 5** and **N = 6**.
11. D + E = 12 so **Y = 2**.

Constraint Satisfaction

Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

Constraint Satisfaction Algorithm

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:
   a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.
   b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to OPEN all objects that share any constraints with OB.
   c) Remove OB from OPEN.
2. If the union of the constraints discovered above defines a solution, then quit and report the solution.
3. If the union of the constraints discovered above defines a contradiction, then return failure.
4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:
   a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.
   b) Recursively invoke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

Constraint Satisfaction

Two kinds of rules:

1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

Means-Ends Analysis

- We have studied the strategies which can reason either in forward or backward, but a mixture of the two directions is appropriate for solving a complex and large problem. Such a mixed strategy, make it possible that first to solve the major part of a problem and then go back and solve the small problems arise during combining the big parts of the problem. Such a technique is called **Means-Ends Analysis**.
- It is a mixture of Backward and forward search technique.
- The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS).
- The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

Means-Ends Analysis

How means-ends analysis Works:

- The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving.
- Following are the main Steps which describes the working of MEA technique for solving a problem.

1) First, evaluate the difference between Initial State and final State.
2) Select the various operators which can be applied for each difference.
3) Apply the operator at each difference, which reduces the difference between the current state and goal state.

Means-Ends Analysis

Operator Subgoaling

- In the MEA process, we detect the differences between the current state and goal state.
- Once these differences occur, then we can apply an operator to reduce the differences.
- But sometimes it is possible that an operator cannot be applied to the current state.
- So we create the subproblem of the current state, in which operator can be applied, such type of backward chaining in which operators are selected, and then sub goals are set up to establish the preconditions of the operator is called **Operator Subgoaling**.

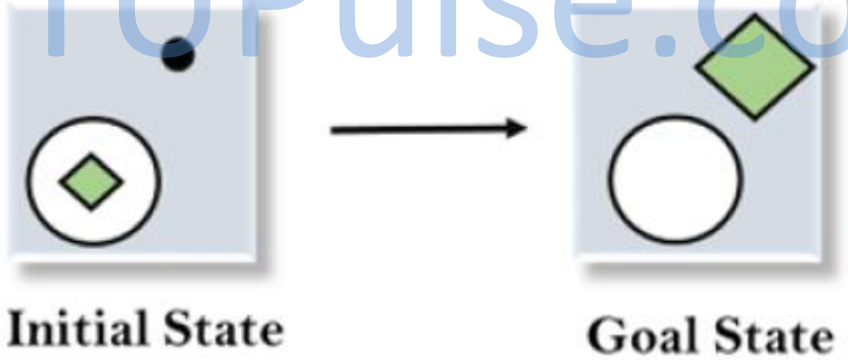
Means-Ends Analysis Algorithm

Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.

- **Step 1:** Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.
- **Step 2:** Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.
  a) Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
  b) Attempt to apply operator O to CURRENT. Make a description of two states.
  i) O-Start, a state in which O?s preconditions are satisfied.
  ii) O-Result, the state that would result if O were applied In O-start.
  c) If
  (First-Part <------ MEA (CURRENT, O-START)
  And
  (LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

Means-Ends Analysis - Example

- Let's take an example where we know the initial state and goal state as given below.
- In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.



Means-Ends Analysis - Example

- To solve the above problem, we will first find the differences between initial states and goal states, and for each difference, we will generate a new state and will apply the operators.
- The operators we have for this problem are:
  - Move
  - Delete
  - Expand

Means-Ends Analysis - Example

1. Evaluating the initial state:

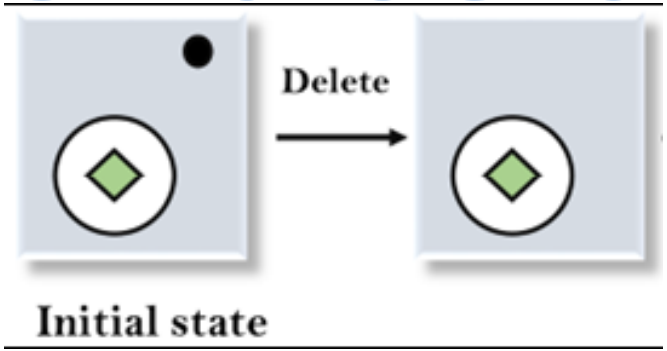
In the first step, we will evaluate the initial state and will compare the initial and Goal state to find the differences between both states.

Initial state

Means-Ends Analysis - Example

2. Applying Delete operator:

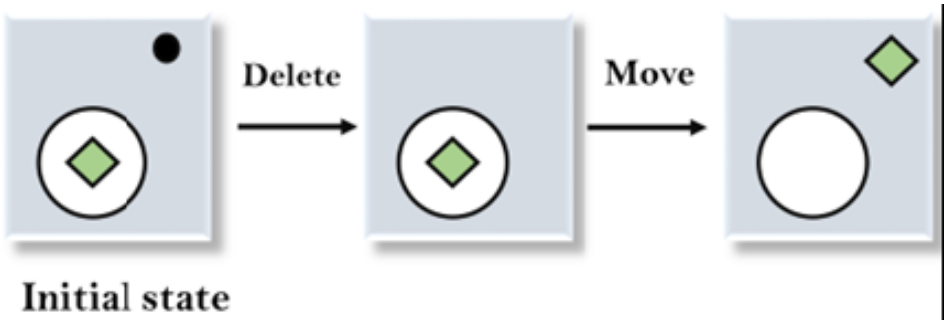
As we can check the first difference is that in goal state there is no dot symbol which is present in the initial state, so, first we will apply the Delete operator to remove this dot.



Means-Ends Analysis - Example

3. Applying Move Operator:

After applying the Delete operator, the new state occurs which we will again compare with goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the Move Operator.



Means-Ends Analysis - Example

4. Applying Expand Operator:

Now a new state is generated in the third step, and we will compare this state with the goal state.

After comparing the states there is still one difference which is the size of the square, so, we will apply Expand operator, and finally, it will generate the goal state.