AI – Prof. R.G.Mehta

- Brief introduction to AI
- Goal-based agents
- Representing states and operators
- Example problems
- Generic state-space search algorithm

What is AI?

- A branch of Science which deals with helping machines finding solutions to complex problems in a more human-like fashion.
- generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way.
- generally associated with Computer Science, but it has many important links with other fields such as Maths, Psychology, Cognition, Biology and Philosophy, among many others.

Components of AI

- Hardware
- Software
- Architectural

Hardware Components of AI

- Pattern Matching
- Logic Representation
- Symbolic Processing
- Numeric Processing
- Problem Solving
- Heuristic Search
- Natural Language processing
- Knowledge Representation
- Expert System
- Neural Network
- Learning
- Planning
- Semantic Network

Software Components of AI

- Machine Language
- Assembly language
- High level Language
- LISP Language / PROLOG
- Fourth generation Language
- Object Oriented Language
- Distributed Language
- Natural Language
- Particular Problem Solving Language

Architectural Components AI

- Uniprocessor
- Multiprocessor
- Special Purpose Processor
- Array Processor
- Vector Processor
- Parallel Processor
- Distributed Processor

Some Definitions of AI

- AI is the study of how to make computers do things which at the moment people do better. This is ephemeral as it refers to the current state of computer science and it excludes a major area ; problems that cannot be solved well either by computers or by people at the moment.
- **AI is a field of study that encompasses computational techniques for performing tasks that apparently require intelligence when performed by humans.**
- AI is the branch of computer science that is concerned with the automation of intelligent behaviour. A I is based upon the principles of computer science namely data structures used in knowledge representation, the algorithms needed to apply that knowledge and the languages and programming techniques used in their implementation.
- **AI is the field of study that seeks to explain and emulate intelligent behaviour in terms of computational processes.**

Some definitions of AI

- **AI is about generating representations and procedures that automatically or autonomously solve problems heretofore solved by humans.**
- AI is the part of computer science concerned with designing intelligent computer systems, that is, computer systems that exhibit the characteristics we associate with intelligence in human behaviour such as understanding language, learning, reasoning and solving problems.
- **AI is the study of mental faculties through the use of computational models.**
- AI is the study of the computations that make it possible to perceive, reason, and act.
- **AI is the exciting new effort to make computers think machines with minds, in the full and literal sense.**
- AI is concerned with developing computer systems that can store knowledge and effectively use the knowledge to help solve problems and accomplish tasks.

Area of AI

- **Perception**
  - **Machine Vision:**
    - interface a TV camera to a computer and get an image into memory;
    - Then understand the image.. What does it represent?
    - Vision takes lots of computation
      - in humans, roughly 10% of all calories consumed are burned in vision computation
  - **Speech Understanding:**
    - Some systems must be trained for the individual user
    - May require pauses between words while trainning
    - Understanding continuous speech with a larger vocabulary is harder.
  - **Touch(tactile or haptic) Sensation**:
    - Important for robot assembly tasks.

Areas of AI

- **Robotics :**
  - Industrial robots have been expensive
  - Now a days economical robot hardware also available
  - Most important is perception and intelligence to tell the robot what to do
  - ``blind'' robots are limited to very well-structured tasks (like spray painting car bodies).
- **Planning:**
  - Planning attempts to order actions to achieve goals
  - Planning applications include logistics, manufacturing scheduling, planning manufacturing steps to construct a desired product.
  - There are huge amounts of money to be saved through better planning.

Areas of AI

- **Expert Systems**
  - attempt to capture the knowledge of a human expert and make it available through a computer program.
  - Expert systems provide the following benefits
    - Reducing skill level needed to operate complex devices
    - Diagnostic advice for device repair
    - Interpretation of complex data
    - `Cloning'' of scarce expertise
    - Capturing knowledge of expert who is about to retire
    - Combining knowledge of multiple experts.

Areas of AI

- **Natural Language Processing**
  - Natural Language Understanding
  - Speech Understanding
  - Language Generation
  - Machine Translation

Areas of AI

- **Game Playing**
  - GP are good vehicles for research because they are well formalized, small, and self-contained
  - GP can be easily programmed
  - Games can be good models of competitive situations, so principles discovered in game-playing programs may be applicable to practical problems.
- **Others areas**
  - Theorem proving
  - Symbolic mathematics

Knowledge in AI

- AI Technique Intelligence requires knowledge but knowledge possesses less desirable properties such as
  - It is voluminous
  - it is difficult to characterise accurately
  - it is constantly changing
  - it differs from data by being organised in a way that corresponds to its application

Characteristics of Knowledge in AI

- AI technique exploits the represented knowledge so that
  - The knowledge captures generalisations
    - situations that share properties, are grouped together, rather than being allowed separate representation.
  - It can be understood by people who must provide it
    - In many AI domains people must supply the knowledge to programs in a form the people understand and in a form that is acceptable to the program
    - although for many programs the bulk of the data may come automatically, such as from readings
  - It can be easily modified
    - to correct errors and reflect changes in real conditions
  - It can be widely used even if it is incomplete or inaccurate
  - It can be used to narrow the range of possibilities (knowledge) that must be usually considered from the available bulk of knowledge

Problem-solving in AI

- The problem of AI is directly associated with the nature of humans and their activities. So we need a number of finite steps to solve a problem which makes human easy works.
- These are the following steps which require to solve a problem :
  - **Goal Formulation:**
    - the first and simple step in problem-solving
    - It organizes finite steps to formulate a target/goals which require some action to achieve the goal
    - the formulation of the goal is based on AI agents.
  - **Problem formulation:**
    - It is one of the core steps of problem-solving which decides what action should be taken to achieve the formulated goal
    - In AI this core part is dependent upon software agent which consisted of the following components to formulate the associated problem.

Problem Formulation

- Building a system to solve a problem requires the following steps
  - Define the problem precisely
    - including detailed specifications
    - Including an acceptable solution
  - Analyse the problem thoroughly
    - some features may have a dominant affect on the chosen method of solution
  - Isolate and represent the background knowledge needed in the solution of the problem
  - Choose the best problem solving techniques in the solution
    - A state represents a status of the solution at a given step of the problem solving procedure
    - The solution of a problem, thus, is a collection of the problem states.
    - operator is applied to current state to get the next state
    - The process is continued until the goal (desired) state is derived.
- **This Problem solving technique is referred to as state space approach**

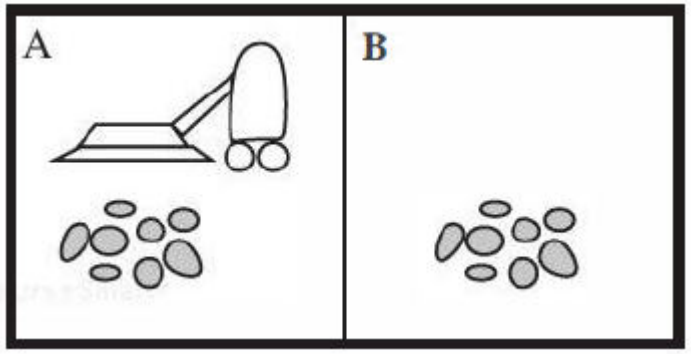
Problem Formulation Simple example

- in order to solve play a game (e.g. two person table or board games) we need
  - the rules of the game
  - the targets for winning
  - means of representing positions in the game
- The opening position can be defined as the initial state and a winning position as a goal state,
- There can be more than one. legal moves allow for transfer from initial state to other states leading to the goal state.
- Rules are far too copious in most games especially like chess and cannot in general be supplied accurately and computer programs cannot easily handle them.
- The storage also presents another problem but searching can be achieved by hashing.
- The number of rules that are used must be minimised and the set can be produced by expressing each rule in as general a form as possible.

Components of Problem Formulation

- **Initial State:**
  - This state requires an initial state for the problem which starts the AI agent towards a specified goal. In this state new methods also initialize problem domain solving by a specific class.
- **Action:**
  - This stage of problem formulation works with function with a specific class taken from the initial state and all possible actions done in this stage.
- **Transition:**
  - This stage of problem formulation integrates the actual action done by the previous action stage and collects the final stage to forward it to their next stage.
- **Goal test:**
  - This stage determines that the specified goal achieved by the integrated transition model or not, whenever the goal achieves stop the action and forward into the next stage to determines the cost to achieve the goal.
- **Path costing:**
  - This component of problem-solving numerical assigned what will be the cost to achieve the goal. It requires all hardware software and human working cost.

Intelligent Agents



A vacuum-cleaner world with just two locations.

Percept sequence	Action
[A, Clean]	Right
[A, Dirty]	Suck
[B, Clean]	Left
[B, Dirty]	Suck
[A, Clean], [A, Clean]	Right
[A, Clean], [A, Dirty]	Suck
⋮	⋮
[A, Clean], [A, Clean], [A, Clean]	Right
[A, Clean], [A, Clean], [A, Dirty]	Suck
⋮	⋮

Partial tabulation of a simple agent function for the vacuum-cleaner world

The water jug problem

- Problem description:
  - There are two jugs called X and Y
  - X holds a maximum of four(4) gallons
  - Y holds a maximum of three (3) gallons.
  - Initial : Both Jugs are empty
  - Goal : How can we get 2 gallons in the jug X?
    - (2,n) where n is a don't care but is limited to X holding from 0 to 3 gallons.
  - [
  - The state space is a set of ordered pairs giving the number of gallons in the pair of jugs at any time ie (X,Y) where X= 0, 1, 2, 3, 4 and Y= 0, 1, 2, 3.
  - The **major production** rules for solving this problem are shown below:

State space for Water-Jug

SrNo	Current State	Control strategy	Nest state	action
1	(X,Y)	if X < 4	(4,Y)	fill X from tap
2	(X,Y)	if Y< 3	(X,3)	fill Y from tap
3	(X,Y)	If X > 0	(0,Y)	empty X into drain
4	(X,Y)	if Y > 0	(X,0)	empty Y into drain
5	(X,Y)	if X+Y<4	(X+Y,0)	empty Y into X
6	(X,Y)	if X+Y<3	(0,X+Y)	empty X into Y
7	(0,Y)	If Y>0	(Y,0)	empty Y into X
8	(X,0)	if X>0	(0,X)	empty X into Y
9	(0,2)	(2,0)		empty Y into X
10	(2,0)	(0,2)		empty X into Y
11	(X,Y)	if X<4	(4,Y-diff)	pour diff, 4-X, into X from Y
12	(Y,X)	if Y<3	(X-diff,3)	pour diff, 3-Y, into Y from X

State space for Water-Jug

- Initial condition goal comment
- 1 (X,Y) if X < 4 (4,Y) fill X from tap
- **1.1 (1,y) (4,Y)**
- **1.2 (2,Y) (4,Y)**
- **1.3 (3,Y) (4,Y)**
- 2 (X,Y) if Y< 3 (X,3) fill Y from tap
- 3 (X,Y) If X > 0 (0,Y) empty X into drain
- **3.1 (1,Y) (0,y)**
- **3.2 (2,y) (0,Y)**
- **3.3 (3,Y) (0,Y)**
- 4 (X,Y) if Y > 0 (X,0) empty Y into drain
- 5 (X,Y) if X+Y<4 (X+Y,0) empty Y into X
- 6 (X,Y) if X+Y<3 (0,X+Y) empty X into Y
- 7 (0,Y) If Y>0 (Y,0) empty Y into X
- 8 (X,0) if X>0 (0,X) empty X into Y

State space for Water-Jug

Initial condition	goal	comment		
1	(X,Y)	if X < 4	(4,Y)	fill X from tap
2	(X,Y)	if Y< 3	(X,3)	fill Y from tap
3	(X,Y)	If X > 0	(0,Y)	empty X into drain
4	(X,Y)	if Y > 0	(X,0)	empty Y into drain
5	(X,Y)	if X+Y<4	(X+Y,0)	empty Y into X
6	(X,Y)	if X+Y<3	(0,X+Y)	empty X into Y
7	(0,Y)	If Y>0	(Y,0)	empty Y into X
8	(X,0)	if X>0	(0,X)	empty X into Y
9	(0,2)	(2,0)		empty Y into X
10	(2,0)	(0,2)		empty X into Y
11	(X,Y)	if X<4	(4,Y-diff)	pour diff, 4-X, into X from Y
12	(Y,X)	if Y<3	(X-diff,3)	pour diff, 3-Y, into Y from X

Solution

X	Y	Rule
0	0	
0	3	2
3	0	7
3	3	2
4	2	11
0	2	3
2	0	10

Control strategies

- Characteristics of a good control strategy
  - It must causes motion.
    - In a game playing program the pieces move on the board and in the water jug problem water is used to fill jugs.
  - it must be systematic
    - it would not be sensible
    - to fill a jug and empty it repeatedly
    - to move a piece round and round the board in a cyclic way.
  - Systematic approaches of **searching**

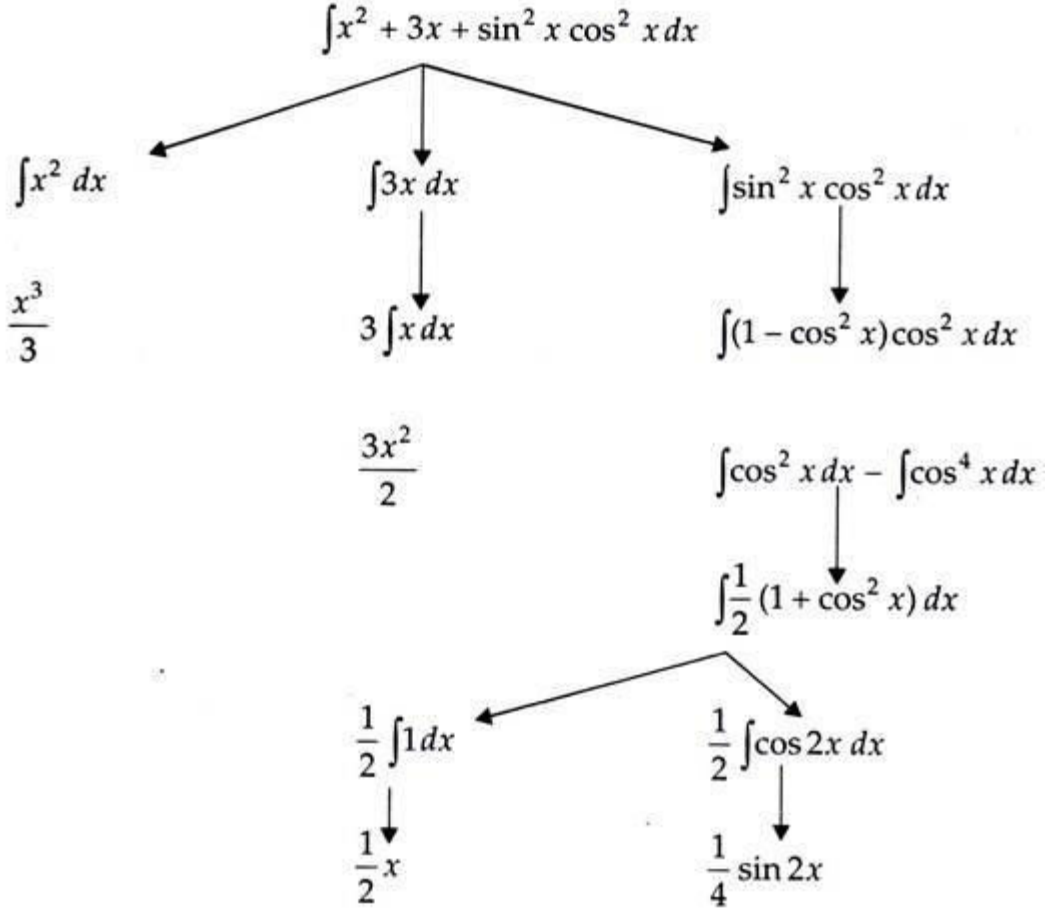
PROBLEM CHARACTERISTICS

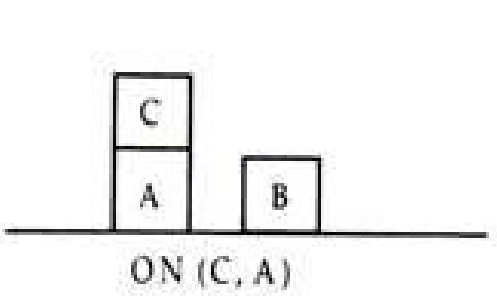
- The problem may have different type of representation and explanation
- Key dimensions of the problems are:
  - Is the problem decomposable into set of sub problems?
  - Can the solution step ignorable / recoverable or non-recoverable?
  - Is the problem universally predictable?
  - Is a good solution to the problem obvious without comparison to all the possible solutions?
  - Is the desire solution a state of world or a path to a state?
  - Is a large amount of knowledge absolutely required to solve the problem?
  - Will the solution of the problem required interaction between the computer and the person?

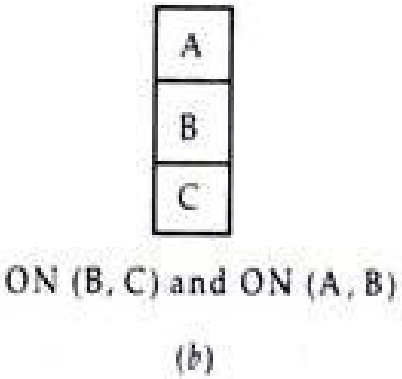
  The above characteristics of a problem are called as 7-problem characteristics under which the solution must take place.

Decomposable vs Non-decomposable

- **Is the problem Decomposable?**







Ignorable or Recoverable

- **Ignorable problem:**
  - solution steps can be ignored
  - **Theorem Proving :**

  E.g.
  - We proceed by first proving a lemma that we think will be useful.
  - Eventually, we realize that the lemma is not help at all.Every thing we need to know to prove theorem is still true and in memory, if it ever was.
  - Any rule that could have been applied at the outset can still be applied. All we have lost is the effort that was spent exploring the blind alley.

Ignorable or Recoverable

- **Recoverable problem:**
  - in which solution steps can be undone.
  - Ex:- **The 8-Puzzle**

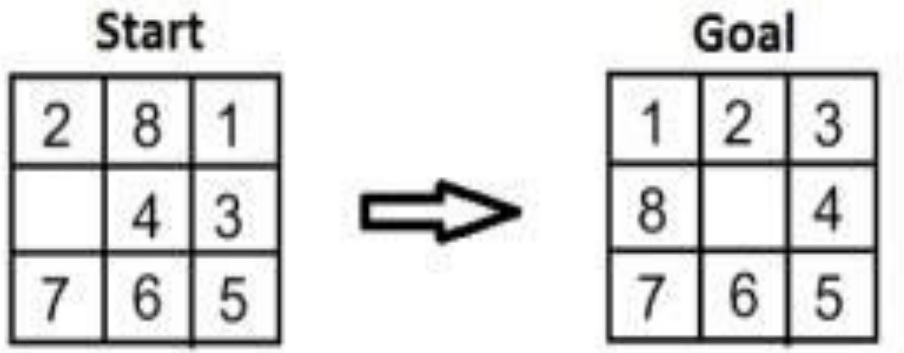


Figure : 8 Puzzle Problem

- While attempting to solve the 8-puzzle problem, mistakenly we make a wrong move and realize that mistake
- Now to correct our mistake we need to undo incorrect steps
- To undo incorrect steps the control mechanism must keep track of the order in which steps are performed, so that we can backtrack to the initial state and start with some correct move.

Recoverable vs Irrecoverable

- **S**olution steps cannot be undone. (e.g., Chess).
- a wrong move is realized after couple of moves, cannot simply be ignored or undone.
- Here, once we make a move we never recover from that step. Only we can try to give the best of the current situation.

Predictable or Non-Predictable Universe

- Predictable problem (8-Puzzle)
  - Every time we make a move, we know exactly what will happen.
  - possible to plan entire sequence of moves and
  - Surety of the resulting state will be.
- Uncertain-outcome problem(play Bridge)
  - One of the decisions we will have to make is which card to play on the first trick.
  - What we would like to do is to plan entire hand before making the 1st hand.
  - But now it is not possible to do such planning with certainty since we cannot know exactly where all the cards are or what the other players will do on their turn.

Absolute or Relative Solution

- Any-path problem (Answer-question System0
  - Consider the problem of answering the question based on following facts:

1. Marcus was a man.
2. Marcus was a Pompean.
3. Marcus was born in 40 AD.
4. All men are mortal.
5. All Pompeans died when volcano erupted in 79 AD.
6. No mortal lives longer than 150 years.
7. Now it is 1991 AD.

- **Question : " Is Marcus alive?"**

- Marcus was a man - Axiom1
- All men are mortal -Axiom4
- Marcus is Mortal – 1,4
- Marcus was born in 40 AD -Axiom3
- Now it is 1991 AD -Axiom7
- Marcus age is 1951 years – 3,7
- No mortal lives longer than 150 years -Axiom6
- Marcus is dead -6,8,9

- goal is to answer to question,
- it does not matter which path we follow

Absolute or Relative Solution

- Best-path problem
  - Ex: Traveling Salesman Problem
    - Given a road map of n cities, find the shortest tour which visits every city on the map exactly once and then return to the original city
  - Best-path problems are, in general, computationally harder than any-path problems.
  - heuristic that choose the best solution should be used
    - much more exhaustive search will be performed
- Any-path problems
  - can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore.
  - If the heuristics are not perfect, the search for a solution may not be as optimum as possible (but that does not matter)

Solution as State or Path

- Solution as Path
  - Ex: Water jug problem
  - Here is not sufficient to report that we have solved the problem and the final state is (2,0).
  - Here we must report is not the final state but the path that we found to that state.
  - Thus a statement of solution to this problem must be a sequence of operations (some time called apian) that produce the final state.
- Solution is a state of world
  - Ex: Natural language understanding
  - To solve the problem of finding the interpretation we need to produce interpretation itself.
  - No record of processing by which the interpretation was found is necessary.

Role of Knowledge

- Knowledge is important only to **constrain the search for solution**
- Knowledge is required even to be **able to recognize a solution**

Knowledge to constrain the search for solution

- Ex: playing chess
  - Suppose you have ultimate computing power available. How much knowledge would be required by a perfect program?
  - just the rule for determining legal moves and some simple control mechanism that implement an appropriate search procedure
  - Tic-ta-Toe game
  - Chess
  - 8 puzzle

Knowledge to recognize a solution

- Example:
  - Scanning daily news paper to decide which group of the community is supporting the democrats and which are supporting the republicans in upcoming elections
  - **The knowledge contain**
    - The name of candidates in each party
    - The knowledge abut people
  - The knowledge must be able to identify
    - Supporters of republicans…. Check for people who have lowered taxes
    - Supporters of democrats….. check for minority students with improved education
    - And so on……

Interactive Task

- Non interactive
  - the computer is given a problem description and produces an answer
  - no intermediate communication
  - no demand for an explanation for the process.
- Interactive
  - Level of interaction b/w computer and user is problem-in solution-out.
  - EX: Theorem Proving Conversational
    - there is intermediate communication between a person and the computer, either to provide additional assistance to computer
  - Ex.: Medical diagnosis
    - provide additional information to user and computer both.

Some Game Problems

Water Jug Problem

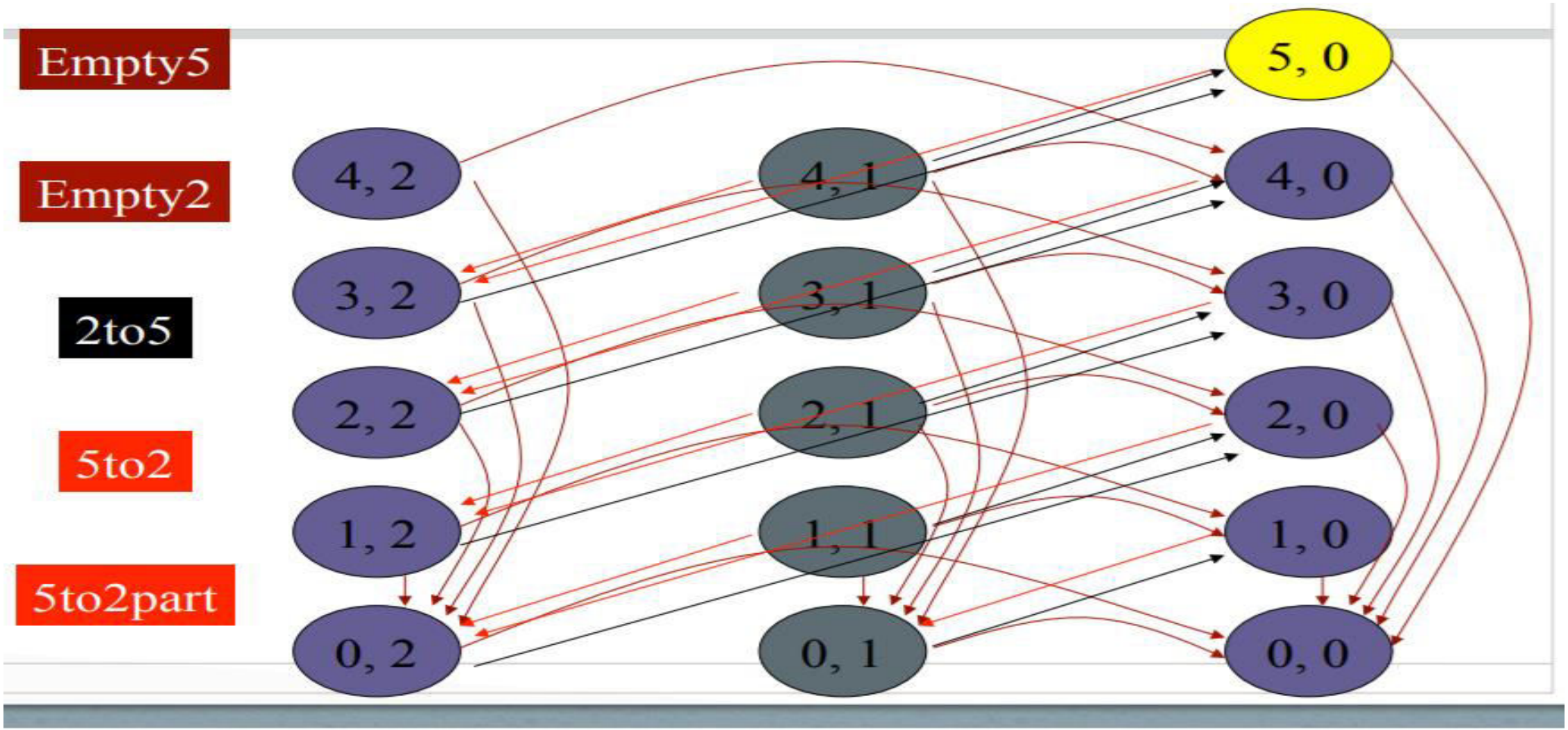
Given a full 5-gallon jug and an empty 2-gallon jug, the goal is to fill the 2-gallon jug with exactly one gallon of water.

- **State** = (x,y), where x is the number of gallons of water in the 5-gallon jug and y is # of gallons in the 2-gallon jug
- **Initial State** = (5,0)
- **Goal State** = (*,1), where * means any amount

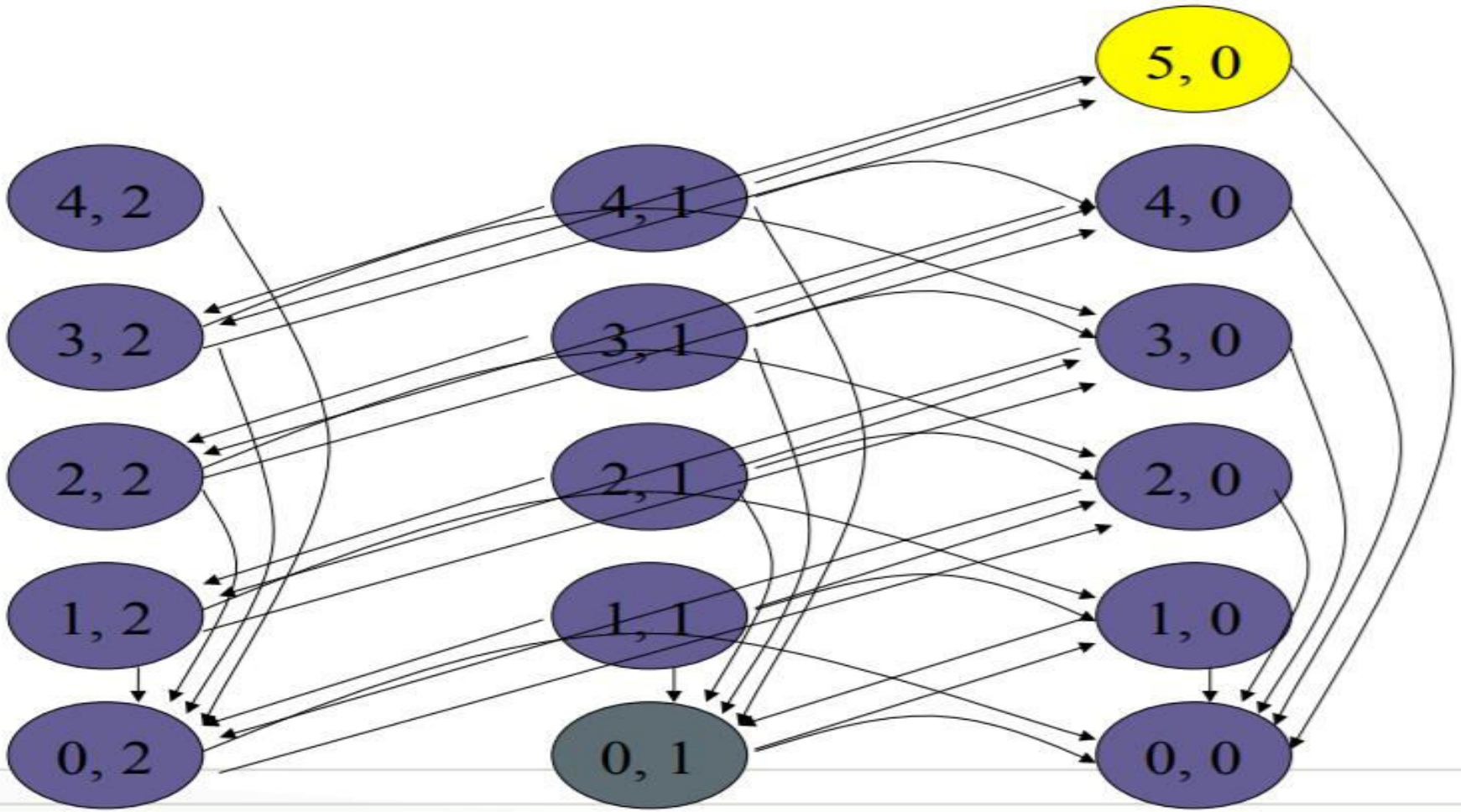
Operator table

Name	Cond.	Transition	Effect
Empty5	–	(x,y)→(0,y)	Empty 5-gal. jug
Empty2	–	(x,y)→(x,0)	Empty 2-gal. jug
2to5	x ≤ 3	(x,2)→(x+2,0)	Pour 2-gal. into 5-gal.
5to2	x ≥ 2	(x, 0)→(x-2,2)	Pour 5-gal. into 2-gal.
5to2part	y < 2	(1,y)→(0,y+1)	Pour partial 5- gal. into 2-gal.

Water Jug Problem state Space(Graph)



Water jug solution



Water Jug Rpblem (Tree structure)

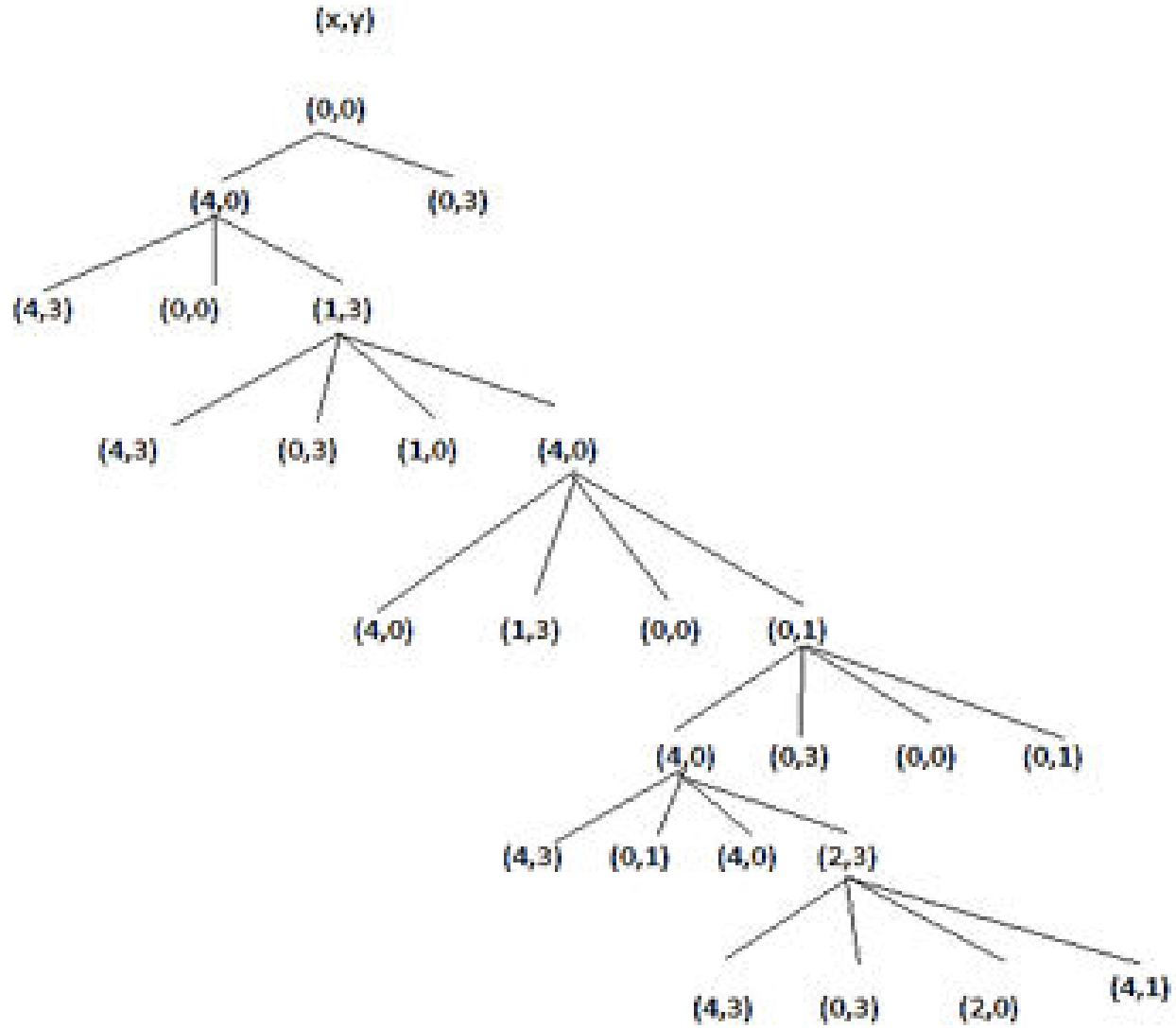
- Rule set:
- (x,y)->(4,y) fill the 4 gallon jug If x<4
- (x,y)->(x,3) fill the 3 gallon jug If x<3
- (x,y)->(x-d,y) pour some water out of the 4-gallon jug.If x>0 (x,y)->(x-d,y)
- pour some water out of the 3-gallon jug.If y>0
- (x,y)->(0,y) empty the 4-gallon jug on the ground If x>0
- (x,y)->(x,0) empty the 3-gallon jug on the ground If y>0
- (x,y)->(4,y-(4-x)) pour water from the 3-gallon jug into the 4-gallon If x+y>=4 and y>0 jug until the 4-gallon jug is full
- (x,y)->(x-(3-y),3)) pour water from the 4-gallon jug into the 3-gallon If x+y>=3 and x>0 jug until the 3-gallon jug is full.
- (x,y)->(x+y,0) pour all the water from the 3-gallon jug into If x+y<=4 and y>0 the 3-gallon jug.
- (x,y)->(0,x+y) pour all the water from the 4-gallon jug into If x+y<=3 and x>0 the 3-gallon jug.
- (0,2)->(2,0) pour the 2-gallon from the 3-gallon jug into the 4-gallon jug.
- (2,y)->(0,x) empty the 2 gallon in the 4 gallon on the ground

Production of the Water Jug Problem

Gallons in the 4-gallon jug	Gallons in the 3-gallon	Rule Applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

One solution to the water jug problem.

Search in the statespace



8 Puzzle problem

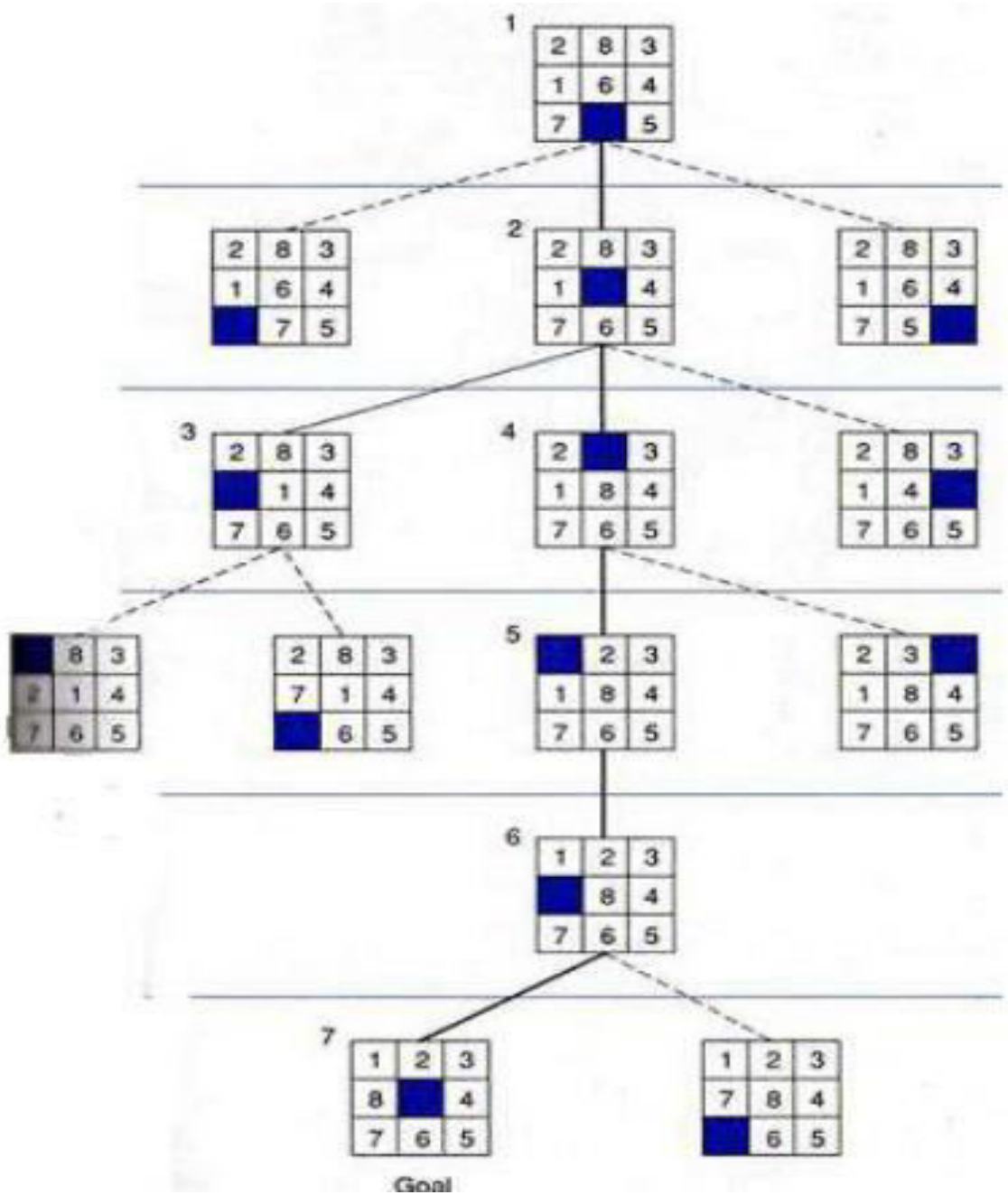
- The program is to change the initial configuration into the goal configuration.
- A solution to the problem is an appropriate sequence of moves, such as “move tile 5 to the right, move tile 7 to the left, move tile 6 to the down” etc…
- To solve a problem, we must specify the
  - Global database(Problem staes)
  - Rules(Moves and control strategy)
  - Goal
- In this problem each tile configuration is a state.
- The set of all possible configuration in the problem space, consists of 3,62,880 different configurations of the 8 tiles and blank space.
- For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers.

8 puzzle problem

- The 8-puzzle is conveniently interpreted as having the following moves
  - Move empty space (blank) to the left, move blank up, move blank to the right and move blank down.
- These moves are modeled by production rules that operate on the state descriptions in the appropriate manner
- The goal condition forms the basis for the termination.
- The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced.
- It also keeps track of rules that have been applied so that it can compose them into sequence representing the problem solution.

8 puzzle game

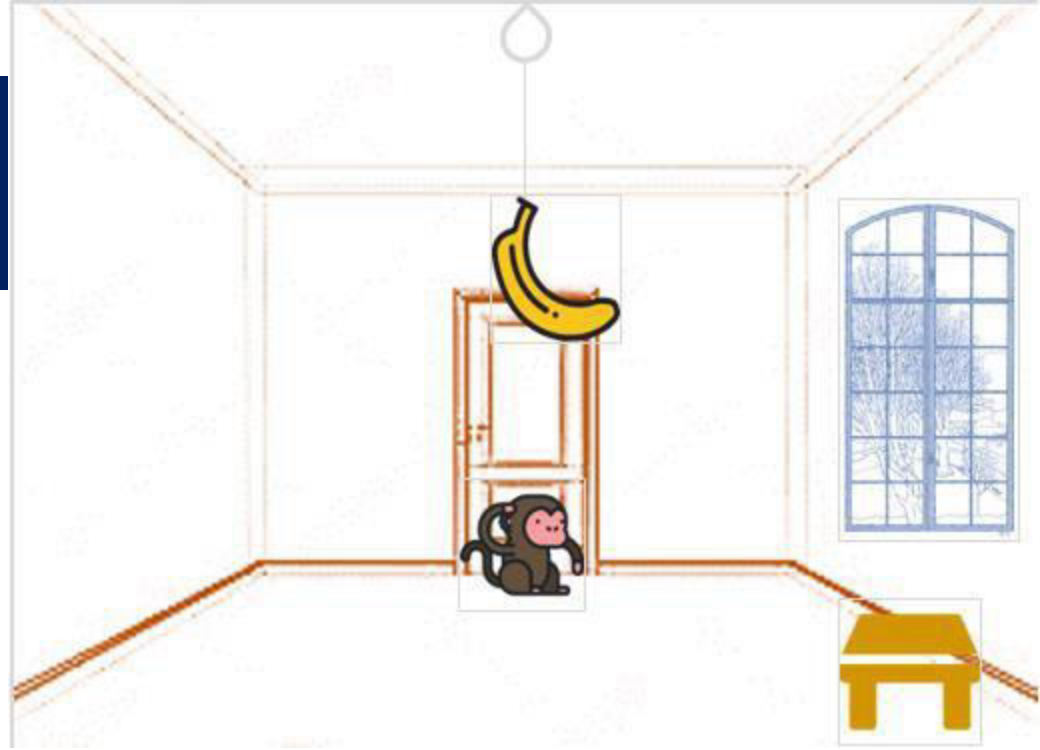
- Sample Solution



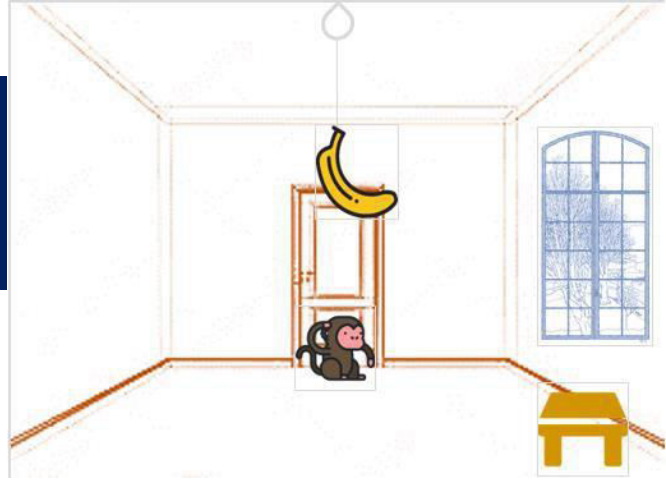
Monkey banana Problem

Problem Statement

- A hungry monkey is in a room, and he is near the door
- The monkey is on the floor
- Banana is hung from the center of the ceiling of the room
- There is a block (or chair) present in the room near the window
- The monkey wants the banana, but cannot reach it



The Monkey Banana Problem



Strategy

Monkey can come to the block, drag the block to the center, climb on it, and get the banana.

Below are few observations in this case −

- Monkey can reach the block, if both of them are at the same level.
  - From the above image, we can see that both the monkey and the block are on the floor.
- If the block position is not at the center, then monkey can drag it to the center.
- If monkey and the block both are on the floor, and block is at the center, then the monkey can climb up on the block.
  - So the vertical position of the monkey will be changed.
- When the monkey is on the block, and block is at the center, then the monkey can get the bananas.

The Monkey Banana Problem (State Desccription)

- Initial State
  - Monkey is on the ground
  - Monkey is Not in the Center
  - Box is on the ground
  - Box is not in the center
  - State (0,P1,0,P2)
- Goal State
  - Monkey is on the Box
  - Box is in the center
  - Monkey has banana
- Action
  - Monkey Walk to the Box
  - Monkey drag the box from Position1 to Center
  - Monkey Climb the Box
  - Monkey has banana

The Monkey Banana Problem (State Description)

- Sate() as a predicate
  - Monkey Position in the room (Middle, or not)
  - Monkey Position related to box (On the Box or not)
  - Box Position (Middle, or not)
  - Banana status (has, or hasnot)
  - E.g. state(MIDDLE,1,MIDDLE,0,hasnot)
- Grasp() as a predicate
  - E.g. grasp(stae(stae(middle,onbox,middle,hasnot),stae(middle,onbox,middle,has))

Prolog Predicates to the Monkey Banana Problem

Here Four different moves are defined with Single predicate “move”

- state(middle,onbox,middle,hasnot)
- state(middle,onbox,middle,has)
- move(state(middle,onbox,middle,hasnot), grasp, state(middle,onbox,middle,has))
- move(state(P,onfloor,P,H), climb, state(P,onbox,P,H)).
- move(state(P1,onfloor,P1,H), drag(P1,P2), state(P2,onfloor,P2,H)).
- move(state(P1,onfloor,B,H), walk(P1,P2), state(P2,onfloor,B,H)).
- canget(state(_,_,_,has)).
- canget(State1) :- move(State1,_,State2), canget(State2).

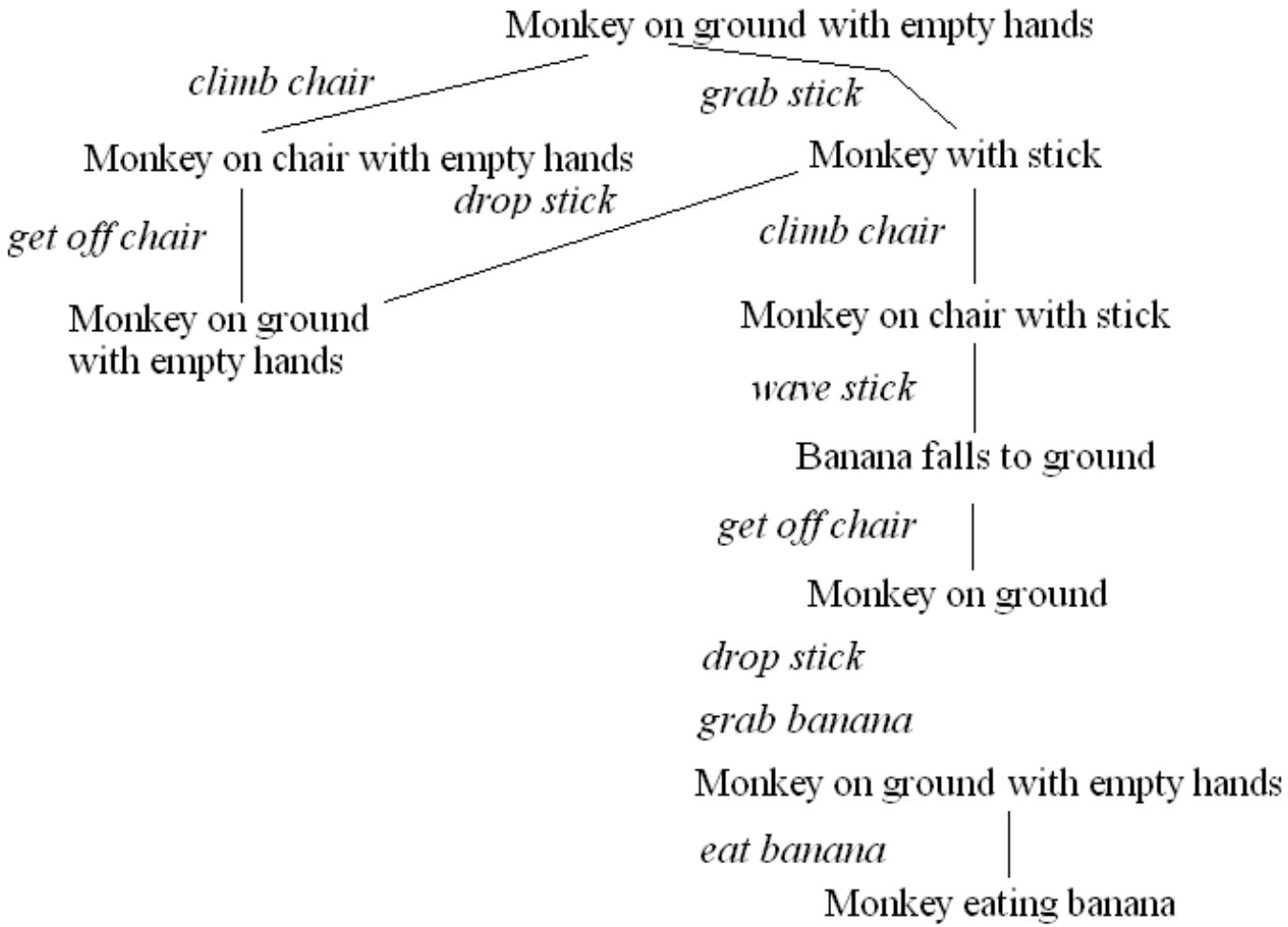
Monkey banana Problem : 2

- A monkey is in a cage and bananas are suspended from the ceiling, the monkey wants to eat a banana but cannot reach them
  - in the room are a chair and a stick
  - if the monkey stands on the chair and waves the stick, he can knock a banana down to eat it
  - what are the actions the monkey should take?

- Initial state:
  - monkey on ground
  - with empty hand
  - bananas suspended
- Goal state:
  - monkey eating
- Actions:
  - climb chair/get off
  - grab X
  - wave X
  - eat X

Monkey banana Problem : 2(Search Space)

- Define state space (HW)



Missionaries and Cannibals

- 3 missionaries and 3 cannibals are on one side of the river with a boat that can take exactly 2 people across the river
  - how can we move the 3 missionaries and 3 cannibals across the river
  - with the constraint that
    - the cannibals never outnumber the missionaries on either side of the river (lest the cannibals start eating the missionaries!)??
    - a boat that can take exactly 2 people across the river

Missionaries and Cannibals(strategy 1)

- We can represent a state as a 6-item tuple:
  - (a, b, c, d, e, f)
    - a/b = number of missionaries/cannibals on left shore
    - c/d = number of missionaries/cannibals in boat
    - e/f = number of missionaries/cannibals on right shore
    - where a + b + c + d + e + f = 6 and c + d <= 2, c + d >= 1 to move the boat
    - a >= b unless a = 0, c >= d unless c = 0, e >= f unless e = 0

Missionaries and Cannibals

- Legal operations (moves) are
  - 0, 1, 2 missionaries get into boat (c + d must be <= 2)
  - 0, 1, 2 missionaries get out of boat
  - 0, 1, 2 cannibals get into boat (c + d must be <= 2)
  - 0, 1, 2 missionaries get out of boat
  - boat sails from left shore to right shore (c + d must be >= 1)
  - boat sails from right shore to left shore (c + d must be >= 1)
    - drawing the state space will be left as a homework problem

Missionaries and Cannibals (Strategy 2)

- **Objects of the State World:**
  - M M M C C C B
  - 3 missionaries, 3 cannibals, 1 boat, a left river bank, and a right river bank.
  - C represents a cannibal, M represents a missionary, and B represents the location of the boat
- **Representation of a State of the World:**
  - L<M C B>
  - R<M C B>
  - A state of the world is represented as 2 lists :
    - L is the left bank.
    - R is the right bank.
    - C represents the location’s amount of cannibals.
    - M represents the location's amount of missionaries.
    - B is 1 when the boat is on the shore and 0 when it is on the opposite shore.

The state space

The State Space Description:

Initial State: L<3 3 1> R<0 0 0>

Goal State: L<0 0 0> R<3 3 1>

State Space Operands:

1 Cannibal goes left:
L1C L<M C B> R<M C B> => L<M (C-1) (B-1)> R<M (C+1) (B+1)>

2 Cannibals go left:
L2C L<M C B> R<M C B> => L<M (C-2) (B-1)> R<M (C+2) (B+1)>

1 Missionary goes left:
L1M L<M C B> R<M C B> => L<(M-1) C (B-1)> R<(M+1) C (B+1)>

2 Missionaries go left:
L2M L<M C B> R<M C B> => L<(M-2) C (B-1)> R<(M+2) C (B+1)>

1 Cannibal goes right:
R1C L<M C B> R<M C B> => L<M (C+1) (B+1)> R<M (C-1) (B-1)>

2 Cannibals goes right:
R2C L<M C B> R<M C B> => L<M (C+2) (B+1)> R<M (C-2) (B-1)>

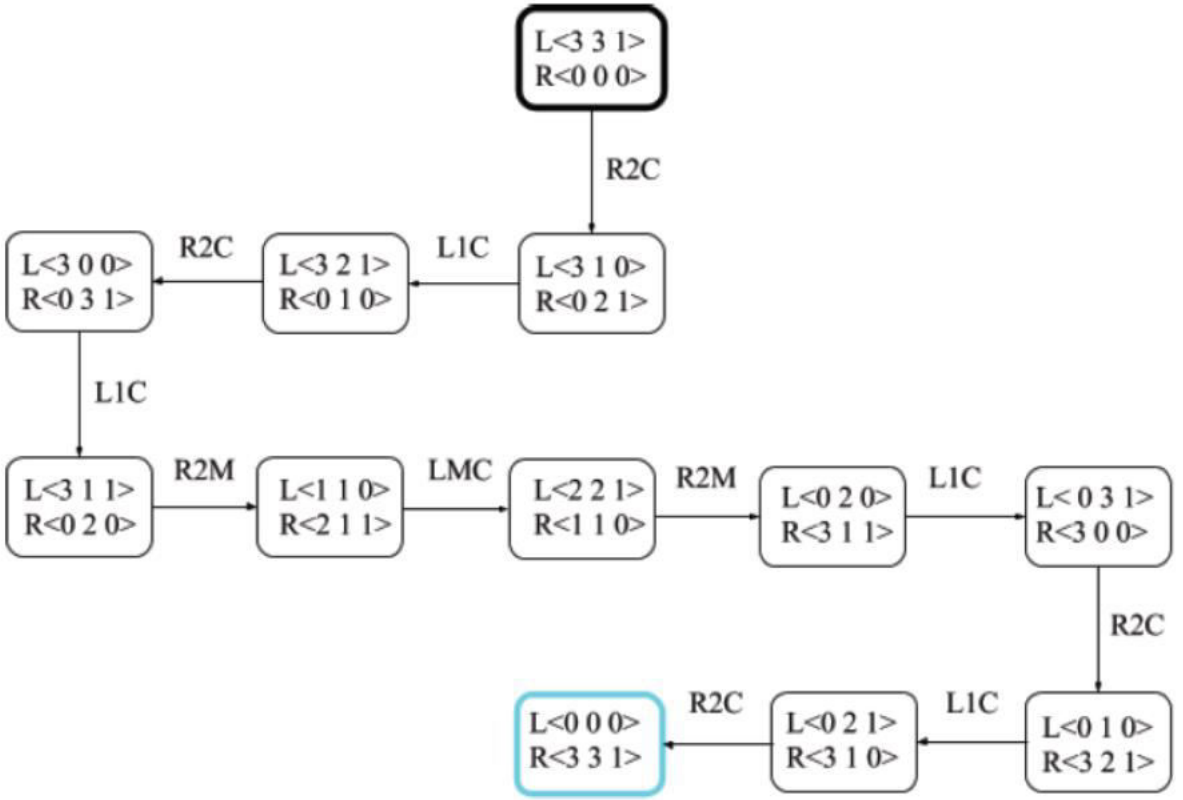
1 Missionary goes right:
R1M L<M C B> R<M C B> => L<(M+1) C (B+1)> R<(M-1) C (B+1)>

2 Missionaries go right:
R2M L<M C B> R<M C B> => L<(M+2) C (B+1)> R<(M-2) C (B+1)>

1 Cannibal and 1 Missionary go left:
LMC L<M C B> R<M C B> => L<(M-1) (C-1) (B-1)> R<(M+1) (C+1) (B+1)>

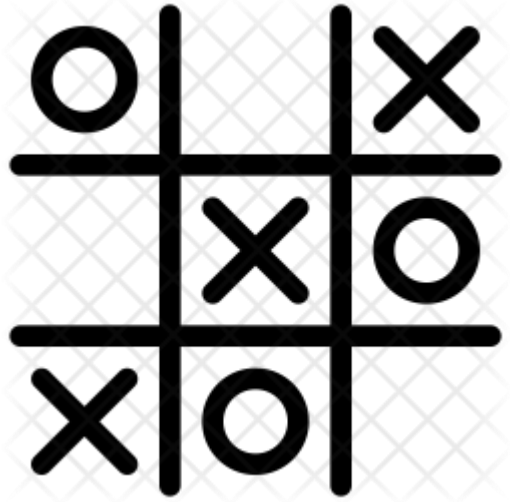
1 Cannibal and 1 Missionary go right:
RMC L<M C B> R<M C B> => L<(M+1) (C+1) (B+1)> R<(M+1) (C+1) (B+1)>

State Space Graph (Including At Least One Solution):



<R2C|L1C R2C L1C R2M LMC R2M L1C R2C L1C R2C>

Tic Tac Toe(Board Game)

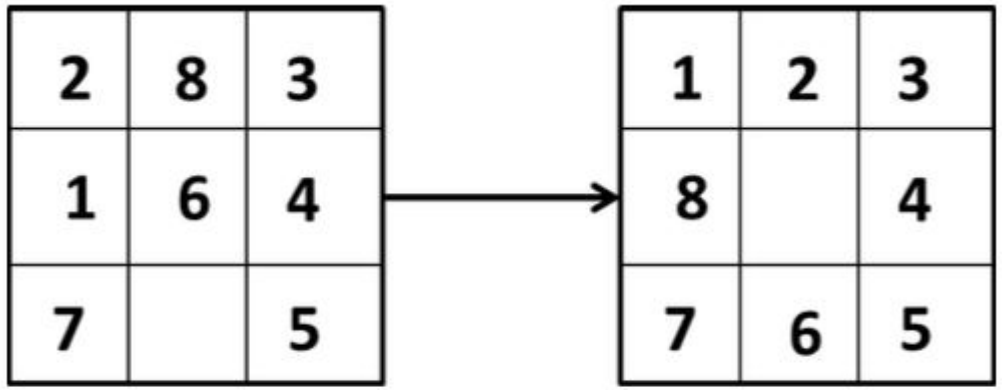


- consists of a nine element vector called BOARD
- it represents the numbers 1 to 9 in three rows
- An element contains the value 0 for blank, 1 for X and 2 for O.
- A MOVETABLE vector consists of 19,683 elements (3^9) and is needed where each element is a nine element vector.
- The contents of the vector are especially chosen to help the algorithm.

Index	Current Board position	New Board position
0	000000000	000010000
1	000000001	020000001
2	000000002	000100002
3	000000010	002000010
	⋮	
	⋮	

8 Puzzle problem

- consists of eight numbered, movable tiles set in a 3x3 frame
- One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell



Chess

- To build a program that could play chess, we have to specify:
- The starting position of the chess board
  - The starting position can be described by an 8 X 8 array square in which each element square (X, Y), (x varying from 1 to 8 & y varying from 1 to 8) describes the board position of an appropriate piece in the official chess opening position.
- The rules that define legal moves
  - The legal moves provide the way of getting from initial state of final state.
  - The legal moves can be described as a set of rules consisting of two parts: A left side that gives the current position and the right side that describes the change to be made to the board position.
- The board position that represent a win.
  - The goal is any board position in which the opponent does not have a legal move and his or her “king” is under attack.

Chess

- An example
- **Current Position:** While pawn at square (5, 2), AND Square (5, 3) is empty, AND Square (5, 4) is empty.
- **Changing Board Position:** Move pawn from Square (5, 2) to Square (5, 4).
- The current position of a chess coin on the board is **its state** and the set of all possible states is **state space.**
- State space representation seems natural for playing chess problem because the set of states, which corresponds to the set of board positions, is well organised.
- One or more states where the problem terminates are goal states.
- Chess has approximately 10^{120} game paths. These positions comprise the problem search space.
- Using above formulation, the problem of playing chess is defined as a problem of moving around in a state space, where each state corresponds to a legal position of the board

Types of Environmnt in AI

An environment in artificial intelligence is the surrounding of the agent. The agent takes input from the environment through sensors and delivers the output to the environment through actuators. There are several types of environments:

- Fully Observable vs Partially Observable
- Deterministic vs Stochastic
- Competitive vs Collaborative
- Single-agent vs Multi-agent
- Static vs Dynamic
- Discrete vs Continuous

Fully Observable vs Partially Observable

- When an agent sensor is capable to sense or access the complete state of an agent at each point in time, it is said to be a fully observable environment else it is partially observable.
- Maintaining a fully observable environment is easy as there is no need to keep track of the history of the surrounding.
- An environment is called **unobservable** when the agent has no sensors in all environments.

Example:

- **Chess –** the board is fully observable, so are the opponent's moves
- **Driving –** the environment is partially observable because what's around the corner is not know.

Deterministic vs Stochastic

- When a uniqueness in the agent's current state completely determines the next state of the agent, the environment is said to be deterministic.
- The stochastic environment is random in nature which is not unique and cannot be completely determined by the agent.

- **Example:**

  - **Chess –** there would be only a few possible moves for a coin at the current state and these moves can be determined

  - **Self Driving Cars –** the actions of a self-driving car are not unique, it varies time to time

Competitive vs Collaborative

- An agent is said to be in a competitive environment when it competes against another agent to optimize the output.
- The game of chess is competitive as the agents compete with each other to win the game which is the output.
- An agent is said to be in a collaborative environment when multiple agents cooperate to produce the desired output.
- When multiple self-driving cars are found on the roads, they cooperate with each other to avoid collisions and reach their destination which is the output desired.

Single-agent vs Multi-agent

An environment consisting of only one agent is said to be a single-agent environment.
A person left alone in a maze is an example of the single-agent system.
An environment involving more than one agent is a multi-agent environment.
The game of football is multi-agent as it involves 11 players in each team.

Static vs Dynamic

An environment that keeps constantly changing itself when the agent is up with some action is said to be dynamic.
A roller coaster ride is dynamic as it is set in motion and the environment keeps changing every instant.
An idle environment with no change in its state is called a static environment.
An empty house is static as there’s no change in the surroundings when an agent enters.

Discrete vs Continuous

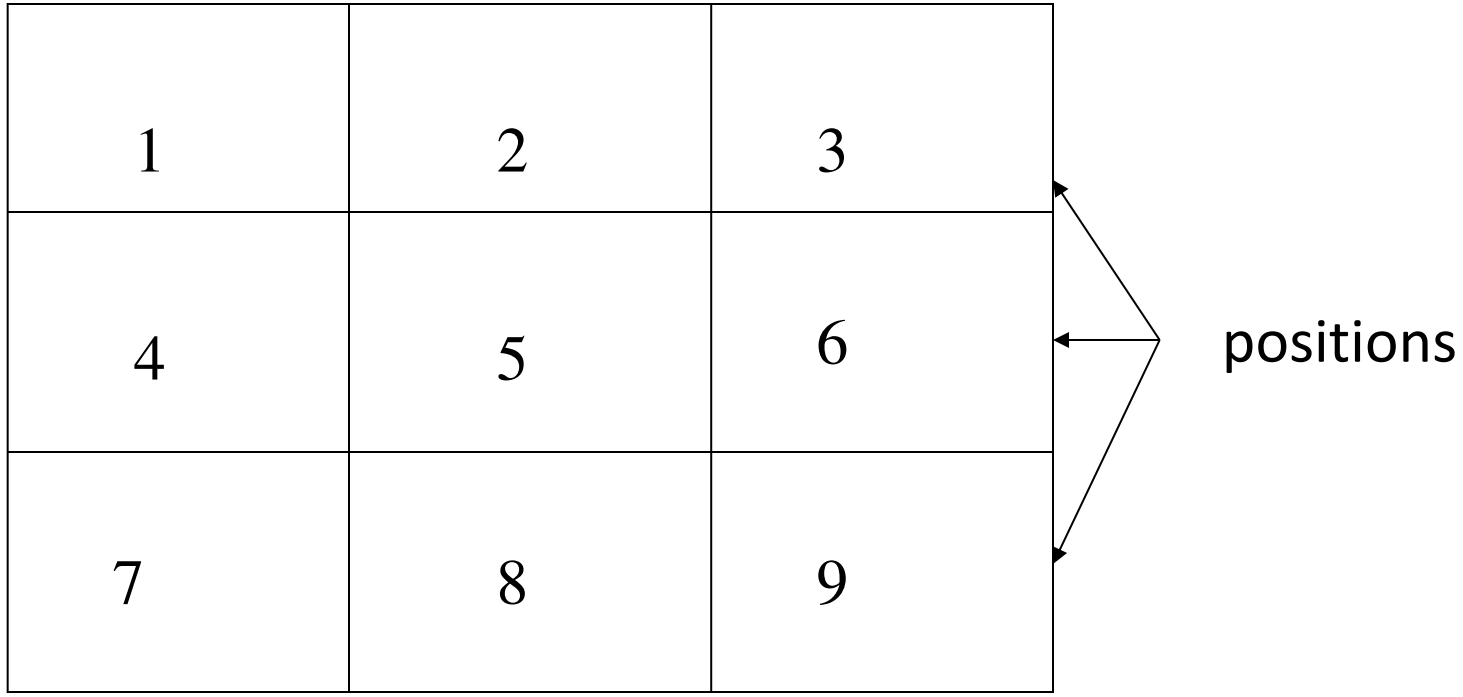
If an environment consists of a finite number of actions that can be deliberated in the environment to obtain the output, it is said to be a discrete environment.
The game of chess is discrete as it has only a finite number of moves. The number of moves might vary with every game, but still, it’s finite.
The environment in which the actions performed cannot be numbered ie. is not discrete, is said to be continuous.
Self-driving cars are an example of continuous environments as their actions are driving, parking, etc. which cannot be numbered.

Tic–Tac–Toe game playing

- Two players
  - human
  - computer.
- The objective is to write a computer program in such a way that computer wins most of the time.
- Three approaches are presented to play this game which increase in
  - Complexity
  - Use of generalization
  - Clarity of their knowledge
  - Extensibility of their approach
- These approaches will move towards being representations of what we will call AI techniques.

Tic Tac Toe Board- (or Noughts and crosses, Xs and Os)

It is two players, *X* and *O*, game who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three respective marks in a horizontal, vertical, or diagonal row wins the game.



Approach 1

- Data Structure
  - Consider a Board having nine elements vector.
  - Each element will contain
    - 0 for blank
    - 1 indicating X player move
    - 2 indicating O player move
  - Computer may play as X or O player.
  - First player who so ever is always plays X.

Move Table MT

- MT is a vector of 3^9 elements, each element of which is a nine element vector representing board position.
- Total of 3^9 (19683) elements in MT

Index	Current Board position	New Board position
0	000000000	000010000
1	000000001	020000001
2	000000002	000100002
3	000000010	002000010
:		
:		

Algorithm

- To make a move, do the following:
  - View the vector (board) as a ternary number and convert it to its corresponding decimal number.
  - Use the computed number as an index into the MT and access the vector stored there.
    - The selected vector represents the way the board will look after the move.
  - Set board equal to that vector.

Comments

- Very efficient in terms of time but has several disadvantages.
  - Lot of space to store the move table.
  - Lot of work to specify all the entries in move table.
  - Highly error prone as the data is voluminous.
  - Poor extensibility
    - 3D tic-tac-toe = 3^{27} board position to be stored.
  - Not intelligent at all.

Approach 2

- **Data Structure**
  - **Board**: A nine-element vector representing the board: B[1..9]
  - Following conventions are used

    2 - indicates blank

    3 - X

    5 - 0
  - **Turn**: An integer

    1 - First move

    9 - Last move

Procedures Used

- ***Make_2*** → Tries to make valid 2
  - Make_2 first tries to play in the center if free and returns 5 (square number).
  - If not possible, then it tries the various suitable non corner square and returns square number.
- ***Go(n)*** ← makes a move in square ‘n’ which is blank represented by 2.

Procedure - PossWin

- ***PossWin (P)*** → Returns
  - 0, if player P cannot win in its next move,
  - otherwise the number of square that constitutes a winning move for P.
- Rule
  - If PossWin (P) = 0 {P can not win} then find whether opponent can win. If so, then block it.

Strategy used by PosWin

- ***PosWin*** checks one at a time, for each rows /columns and diagonals as follows.
  - If 3 * 3 * 2 = 18 then player X can win
  - else if 5 * 5 * 2 = 50 then player O can win
- These procedures are used in the algorithm on the next slide.

Algorithm

- Assumptions
  - The first player always uses symbol X.
  - There are in all 8 moves in the worst case.
  - Computer is represented by C and Human is represented by H.
  - Convention used in algorithm on next slide
    - If C plays first (Computer plays X, Human plays O) - **Odd moves**
    - If H plays first (Human plays X, Computer plays O) - **Even moves**
    - For the sake of clarity, we use C and H.

Algo - Computer plays first – C plays odd moves

- **Move 1:** Go (5)
- **Move** 2: *H plays*
- **Move 3:** If B[9] is blank, then Go(9) else Go(3) ***{make 2}***
- **Move 4:** *H plays*
- **Move 5:** ***{By now computer has played 2 chances}***
  - If PossWin(C) then ***{won}*** Go(PossWin(C))
  - else ***{block H}*** if PossWin(H) then Go(PossWin(H)) else if B[7] is blank then Go(7) else Go(3)
- **Move 6***: H plays*
- **Moves 7 & 9 :**
  - If PossWin(C) then ***{won}*** Go(PossWin(C))
  - else ***{block H}*** if PossWin(H) then Go(PossWin(H)) else Go(Anywhere)
- **Move 8:** *H plays*

Algo - Human plays first – C plays even moves

- **Move 1:** *H plays*
- **Move** 2: If B[5] is blank, then Go(5) else Go(1)
- **Move 3:** *H plays*
- **Move 4**: ***{By now H has played 2 chances}***
  - If PossWin(H) then ***{block H}*** Go (PossWin(H))
  - else Go (Make_2)
- **Move 5:** *H plays*
- **Move** 6: ***{By now both have played 2 chances}***
  - If PossWin(C) then ***{won}*** Go(PossWin(C))
  - else ***{block H}*** if PossWin(H) then Go(PossWin(H)) else Go(Make_2)
- **Moves 7 & 9 :** *H plays*
- **Move** 8: ***{By now computer has played 3 chances}***
  - If PossWin(C) then ***{won}*** Go(PossWin(C))
  - else ***{block H}*** if PossWin(H) then Go(PossWin(H)) else Go(Anywhere)

Complete Algorithm – Odd moves or even moves for C playing first or second

- **Move 1:** go (5)
- **Move** 2: If B[5] is blank, then Go(5) else Go(1)
- **Move 3:** If B[9] is blank, then Go(9) else Go(3) ***{make 2}***
- **Move 4**: ***{By now human (playing X) has played 2 chances}*** If PossWin(X) then ***{block H}*** Go (PossWin(X)) else Go (Make_2)
- **Move 5:** ***{By now computer has played 2 chances}*** If PossWin(X) then ***{won}*** Go(PossWin(X)) else ***{block H}*** if PossWin(O) then Go(PossWin(O)) else if B[7] is blank then Go(7) else Go(3)
- **Move** 6: ***{By now both have played 2 chances}*** If PossWin(O) then ***{won}*** Go(PossWin(O)) else ***{block H}*** if PossWin(X) then Go(PossWin(X)) else Go(Make_2)
- **Moves 7 & 9 :** ***{By now human (playing O) has played 3 chances}*** If PossWin(X) then ***{won}*** Go(PossWin(X)) else ***{block H}*** if PossWin(O) then Go(PossWin(O)) else Go(Anywhere)
- **Move** 8: ***{By now computer has played 3 chances}*** If PossWin(O) then ***{won}*** Go(PossWin(O)) else ***{block H}*** if PossWin(X) then Go(PossWin(X)) else Go(Anywhere)

Comments

- Not as efficient as first one in terms of time.
- Several conditions are checked before each move.
- It is memory efficient.
- Easier to understand & complete strategy has been determined in advance
- Still can not generalize to 3-D.

Approach 3

- Same as approach 2 except for one change in the representation of the board.
  - Board is considered to be a magic square of size 3 X 3 with 9 blocks numbered by numbers indicated by magic square.
- This representation makes process of checking for a possible win more simple.

Board Layout – Magic Square

- Board Layout as magic square. Each row, column and diagonals add to 15.

Magic Square

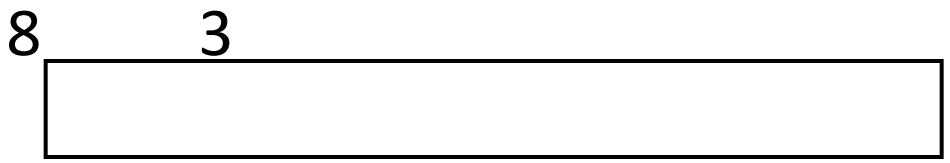
8	3 4	
1	5 9	
6	7 2	

Strategy for possible win for one player

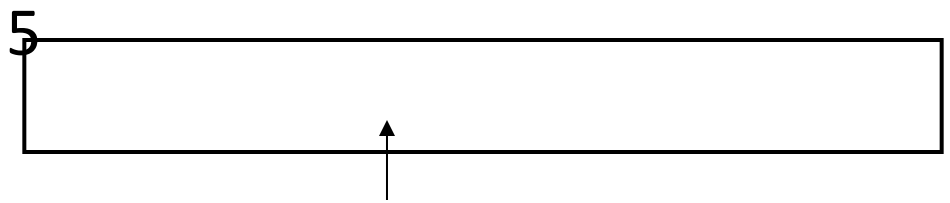
- Maintain the list of each player's blocks in which he has played.
  - **Consider each pair of blocks that player owns.**
  - Compute difference D between 15 and the sum of the two blocks.
  - If D < 0 or D > 9 then
    - these two blocks are not collinear and so can be ignored
    - otherwise if the block representing difference is blank (i.e., not in either list) then a move in that block will produce a win.

Working Example of algorithm

- Assume that the following lists are maintained up to 3rd move.
- Consider the magic block shown in slide 18.
  - First Player X (Human)



  - Second Player O (Computer)



Working – contd..

- Strategy is same as in approach 2
  - First check if computer can win.
    - If not then check if opponent can win.
    - If so, then block it and proceed further.
- Steps involved in the play are:
  - First chance, H plays in block numbered as 8
  - Next C plays in block numbered as 5
  - H plays in block numbered 3
  - Now there is a turn of computer.

Working – contd..

- Strategy by computer: Since H has played two turns and C has played only one turn, C checks if H can win or not.
  - Compute sum of blocks played by H
    - S = 8 + 3 = 11
    - Compute D = 15 – 11 = 4
    - Block 4 is a winning block for H.
    - So block this block and play in block numbered 4.
    - The list of C gets updated with block number 4 as follows:

      H 8 3 C 5 4

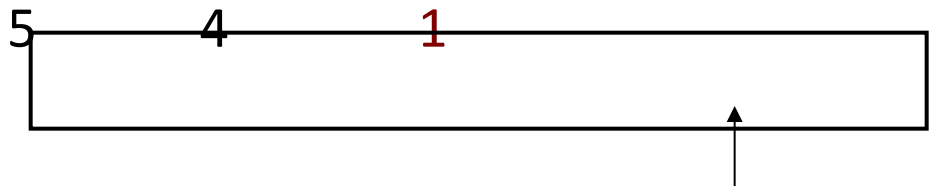
Contd..

- Assume that H plays in block numbered 6.
- Now it's a turn of C.
  - C checks, if C can win as follows:
    - Compute sum of blocks played by C
    - S = 5 + 4 = 9
    - Compute D = 15 – 9 = 6
    - Block 6 is not free, so C can not win at this turn.
  - Now check if H can win.
    - Compute sum of new pairs (8, 6) and (3, 6) from the list of H
    - S = 8 + 6 = 14
    - Compute D = 15 – 14 = 1
    - Block 1 is not used by either player, so C plays in block numbered as 1

Contd..

- The updated lists at 6th move looks as follows:
  - First Player H

  - Second Player C



- Assume that now H plays in 2.
- Using same strategy, C checks its pair (5, 1) and (4, 1) and finds bock numbered as 9 {15-6 = 9}.
- Block 9 is free, so C plays in 9 and win the game.

Comments

- This program will require more time than two others as
  - it has to search a tree representing all possible move sequences (Move sequence list is maintained separately) before making each move.
- This approach is extensible to handle
  - 3-dimensional tic-tac-toe.
  - games more complicated than tic-tac-toe.

3D Tic Tac Toe (Magic cube)

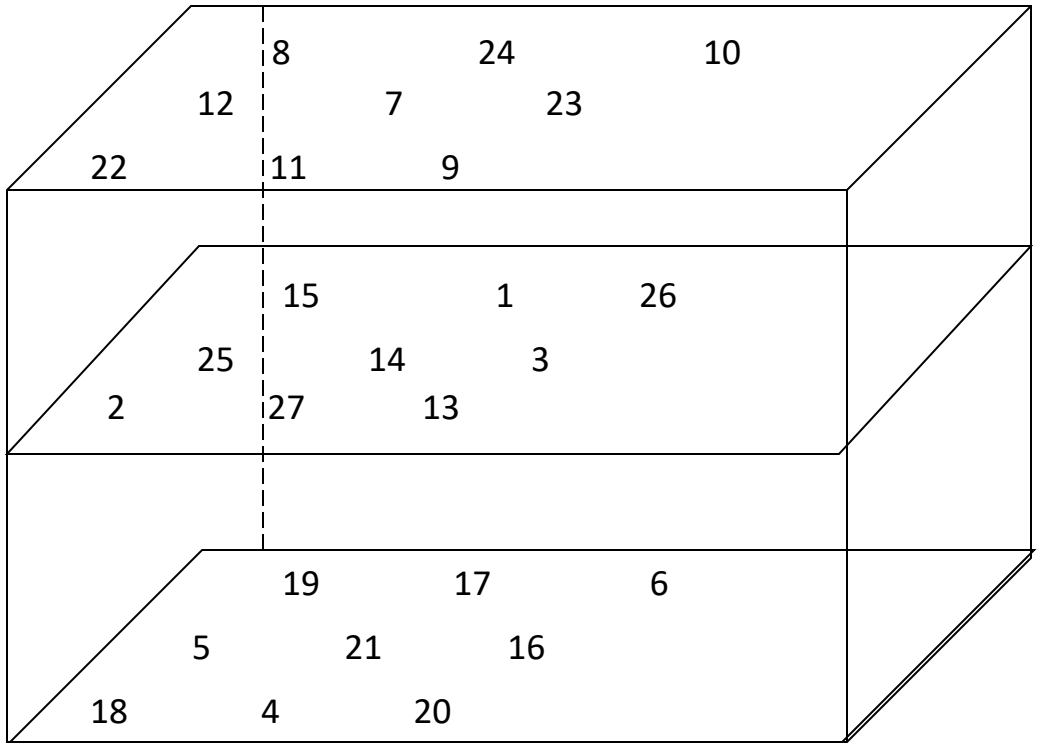
- All lines parallel to the faces of a cube, and all 4 triagonals (Not Diagonals) sum correctly to 42 defined by

$$S = m(m^3 + 1)/2 \text{ , where } m=3$$

- No planar diagonals of outer su rfaces sum to 42
  - there are probably no magic squares in the cube.

8	24	10	15	1	26	19	17	6
12	7	23	25	14	3	5	21	16
22	11	9	2	27	13	18	4	20

8	24	10	15	1	26	19	17	6
12	7	23	25	14	3	5	21	16
22	11	9	2	27	13	18	4	20



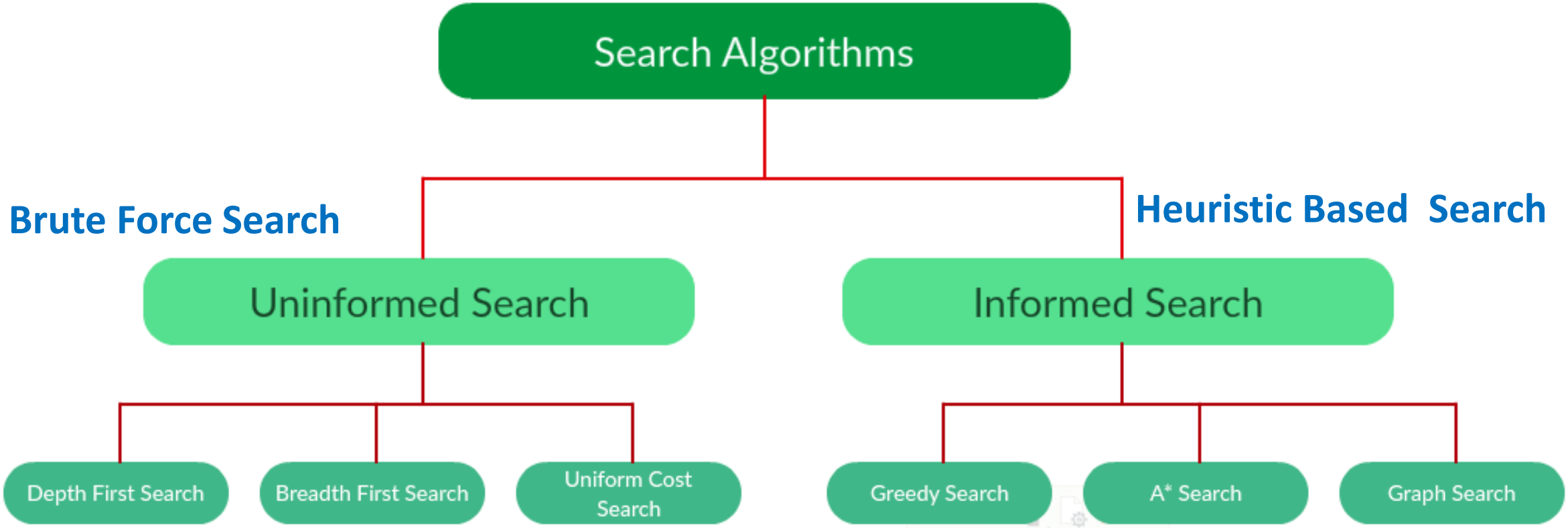
- Magic Cube has 6 outer and 3 inner and 2 diagonal surfaces
- Outer 6 surfaces are not magic squares as diagonals are not added to 42.
- Inner 5 surfaces are magic square.

AI – Prof. R.G.Mehta

- Brief introduction to AI
- Goal-based agents
- Representing states and operators
- Example problems
- **State-space search algorithm(Graph searching)**

Introduction

- General mechanism of searching and present it in terms of searching for paths in directed graphs.
  - First define the underlying search space
  - Apply a search algorithm to that search space.
- A (directed) graph consists of a set of nodes and a set of directed arcs between nodes.
  - The core idea is to find a path along these arcs from a start node to a goal node.
- there may be more than one way to represent a problem as a graph.
- **state-space searching as graph search**
  - Nodes represent states
  - arcs represent actions

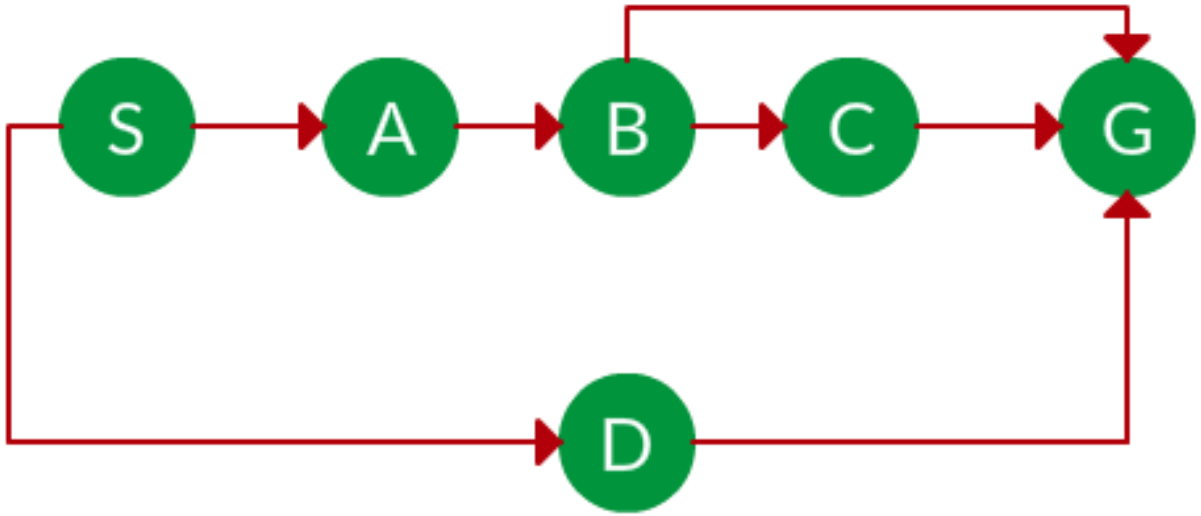

Search Algorithms
Brute Force Search
Heuristic Based Search
Uninformed Search
Informed Search
Depth First Search
Breadth First Search
Uniform Cost Search
Greedy Search
A* Search
Graph Search

Other approaches

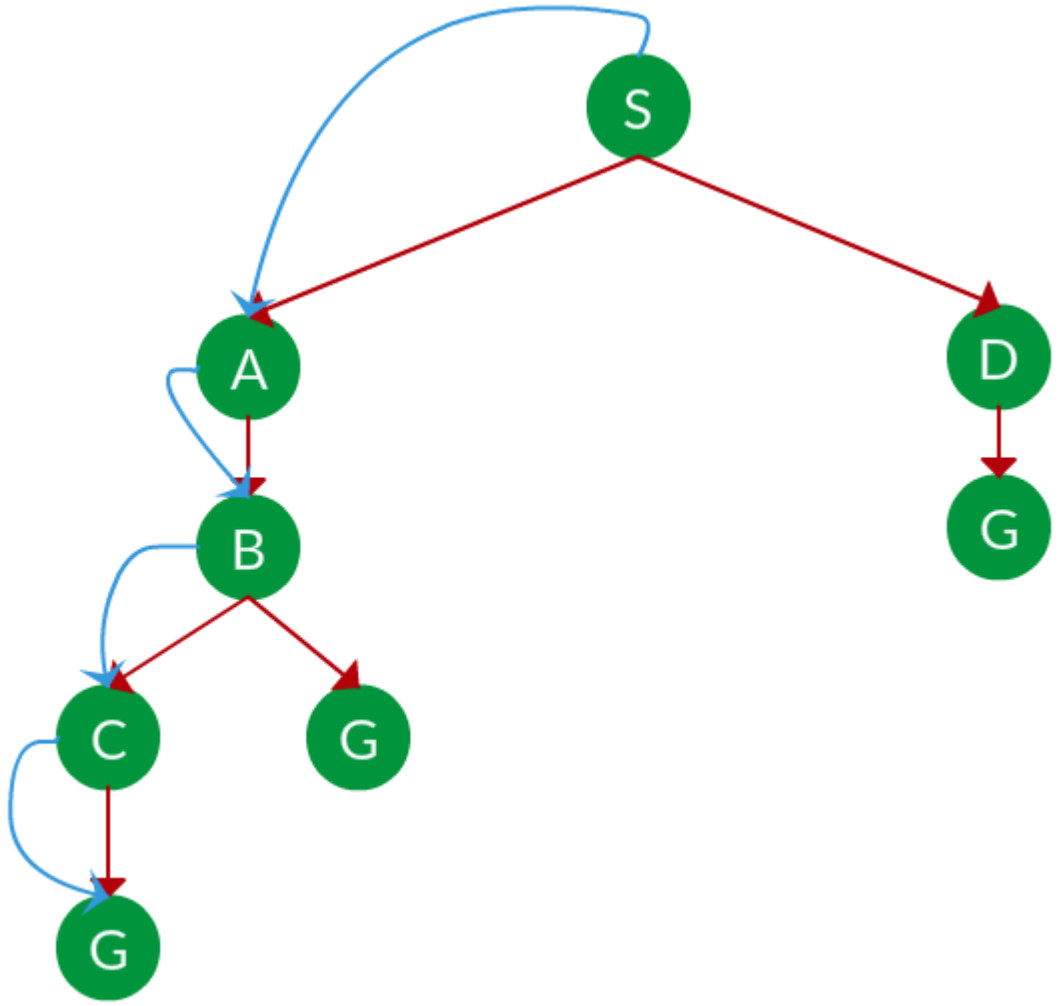
- Brute-Force Approach
- Heuristic Search
- **Two-Player-Games Search**
- **Interleaving Search**

- Each of these algorithms will have:
  - A problem **graph,** containing the start node S and the goal node G.
  - A **strategy,** describing the manner in which the graph will be traversed to get to G.
  - A **fringe,** which is a data structure used to store all the possible states (nodes) that you can go from the current states.
  - A **tree,** that results while traversing to the goal node.
  - A solution **plan,** which the sequence of nodes from S to G.

Search tree for Graph Search



- Source : S
- Goal : G



Brute-Force Approach

- The most general search algorithms
- No domain specific knowledge.
- Require
  - a state description,
  - a set of legal operators,
  - initial state,
  - goal state.
- Also called **uninformed search and blind search.**
- proceed in a systematic way by exploring nodes in some predetermined order or simply by selecting nodes at random.
- return
  - only a solution value when a goal is found

  or
  - record and return the solution path.

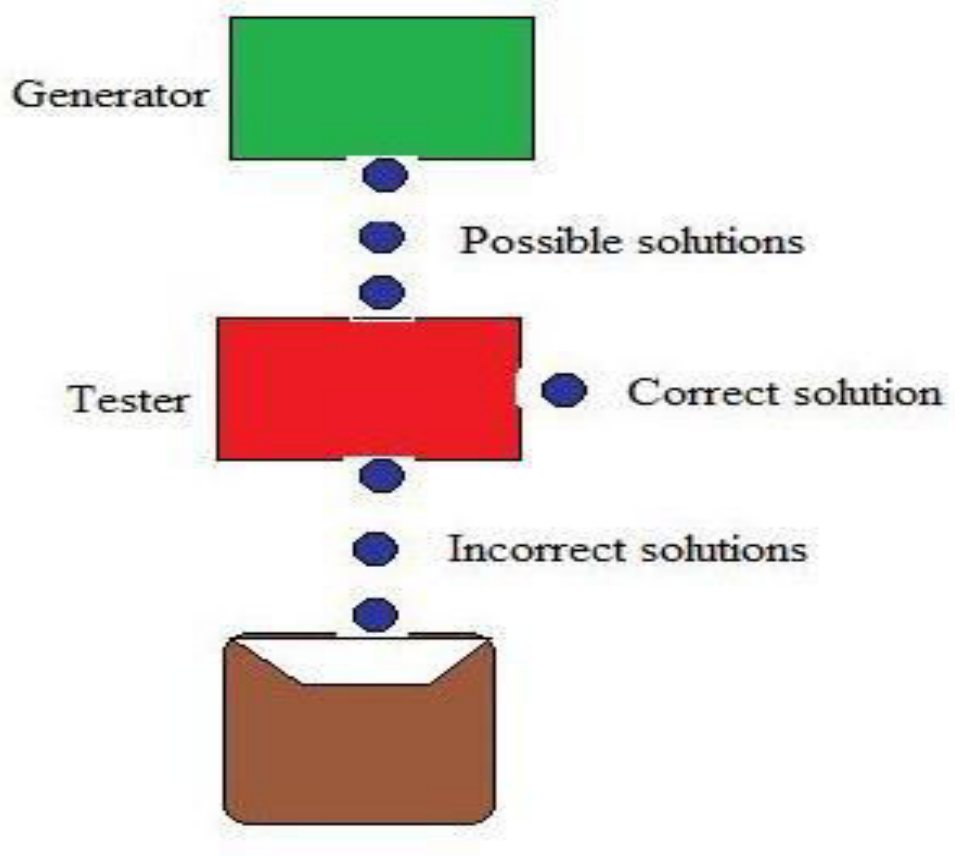
Brute force Techniques

- **Generate-And-Test**
- Breadth-First Search
- Uniform-Cost Search
- Depth-First Search
- Depth-First Iterative-Deepening Search
- Bidirectional Search

Generate-And-Test Algorithm

- very simple algorithm
- guarantees to find a solution if
  - done systematically and
  - there exists a solution.
- **Algorithm:**
  1. Generate a possible solution.
  2. Test to see if this is the expected solution.
  3. If the solution has been found quit else go to step 1.
- Potential solutions that need to be generated vary depending on the kinds of problems.
  - For some problems the possible solutions may be particular points in the problem space
  - For some problems, paths from the start state.

Generate-And-Test Algorithm (cont....)



Generate-And-Test Algorithm (cont....)

- Requires complete solutions be generated for testing (like Depth First Search)
- 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.
  - Also known as British Museum algorithm: finding an object in the British Museum by wandering randomly.

Systematic Generate-And-Test

- Two extreme approaches for G&T
  - generating complete solutions
  - generating random solutions
- The intermediate approach
  - The search proceeds systematically but some paths that unlikely to lead the solution are not considered.
  - This evaluation is performed by a **heuristic function.**
  - E.g. Depth-first search tree with backtracking

Note : 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.

Generate-And-Test And Planning

- Exhaustive generate-and-test is very useful for simple problems.
- For complex problems even heuristic generate-and-test is not very effective technique.
- Combined with other techniques
  - E.g. An AI **program DENDRAL used plan-Generate-and-test** technique.
    - First the planning process uses constraint-satisfaction techniques and creates lists of recommended and contraindicated sub-structures.
    - Then the G&T procedure uses the lists generated and required to explore only a limited set of structures.
    - Major weakness : It often produces inaccurate solutions as there is no feedback from the world.
    - If it is used to produce only pieces of solutions then lack of detailed accuracy becomes unimportant.

Brute force Techniques

- **Generate-And-Test**
- **Breadth-First Search**
- Uniform-Cost Search
- Depth-First Search
- Depth-First Iterative-Deepening Search
- Bidirectional Search

Breadth First Search (BFS)

- Searches breadth-wise in the problem space
- like traversing a tree
  - where each node is a state which may a be a potential candidate for solution.
- expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found.
- implemented by maintaining a queue of nodes
  - Initially the queue contains just the root.
  - In each iteration, node at the head of the queue is removed and then expanded.
  - The generated child nodes are then added to the tail of the queue.

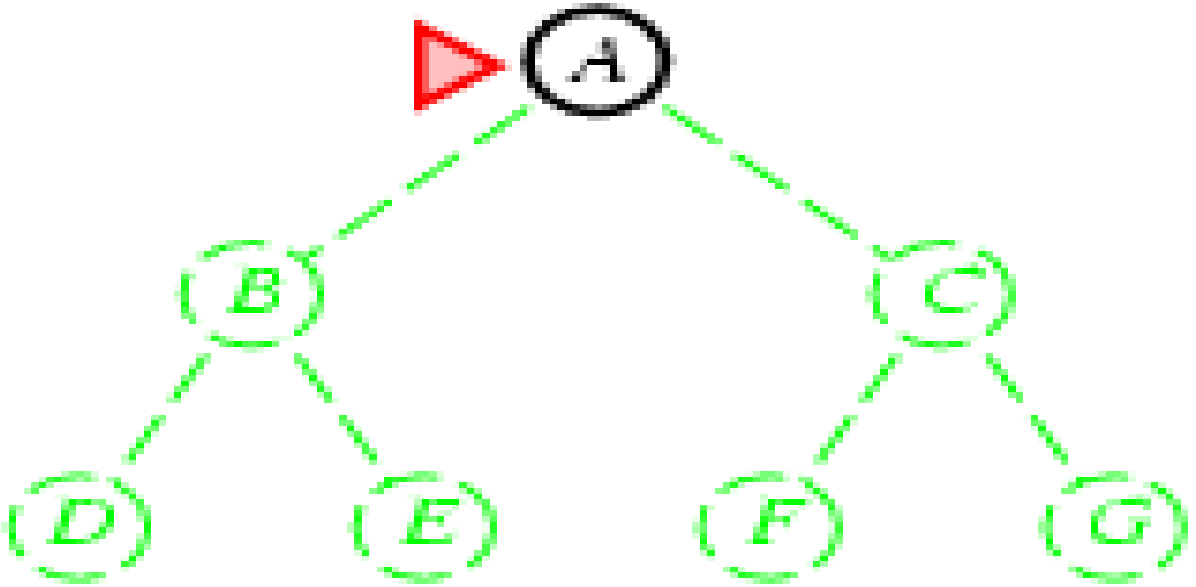
Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.
2. Loop until the goal state is found or NODE-LIST is empty.
   a. Remove the first element, say E, from the NODE-LIST. If NODE-LIST was empty then quit.
   b. For each way that each rule can match the state described in E do:
3. Apply the rule to generate a new state.
4. If the new state is the goal state, quit and return this state.
5. Otherwise add this state to the end of NODE-LIST

Breadth-First Search

- Expand shallowest unexpanded node
- Fringe: nodes waiting in a queue to be explored
- Implementation:
  - *fringe* is a first-in-first-out (FIFO) queue, i.e., new successors go at end of the queue.

Is A a goal state?

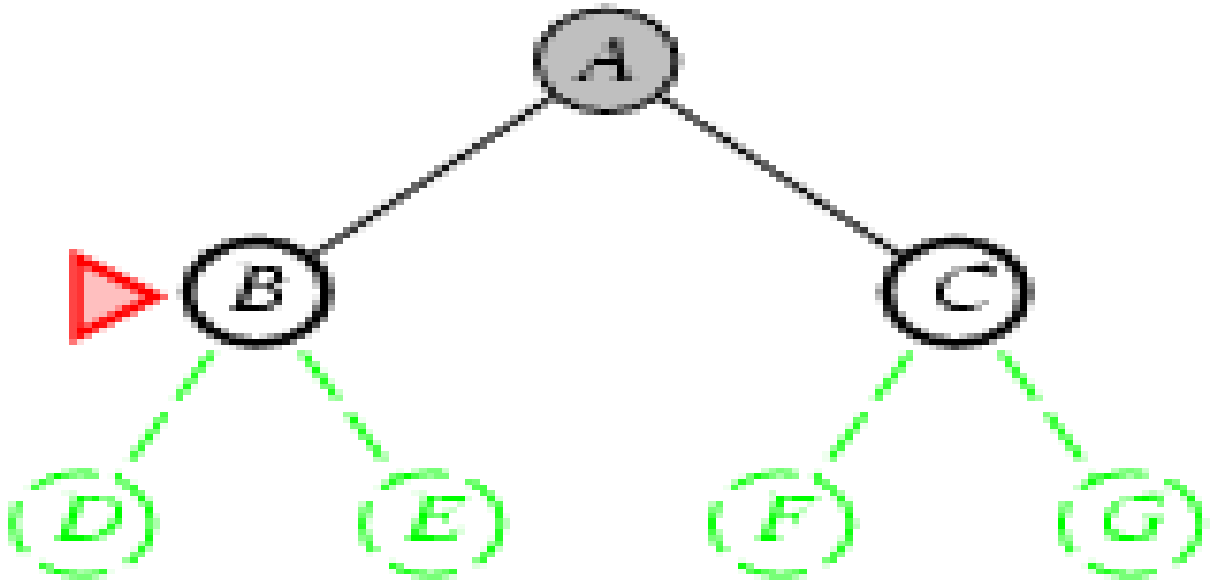


Breadth-First Search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at endc

Expand:
fringe = [B,C]

Is B a goal state?



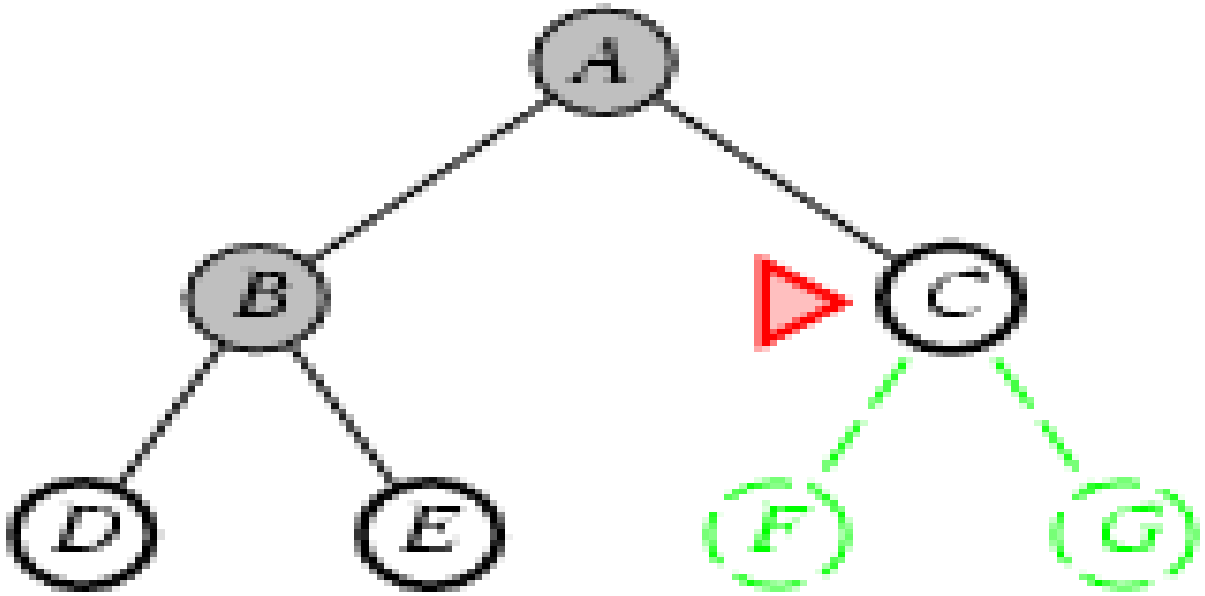
17

Breadth-first search

- Expand shallowest unexpanded node
-
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

Expand:
fringe=[C,D,E]

Is C a goal state?

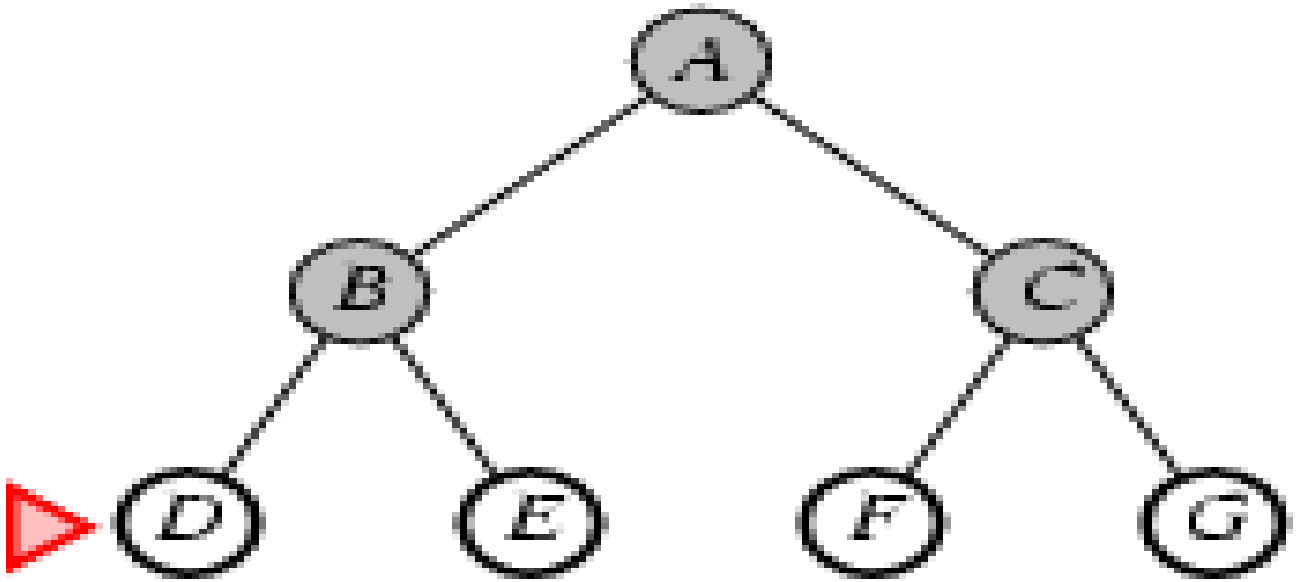


Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end

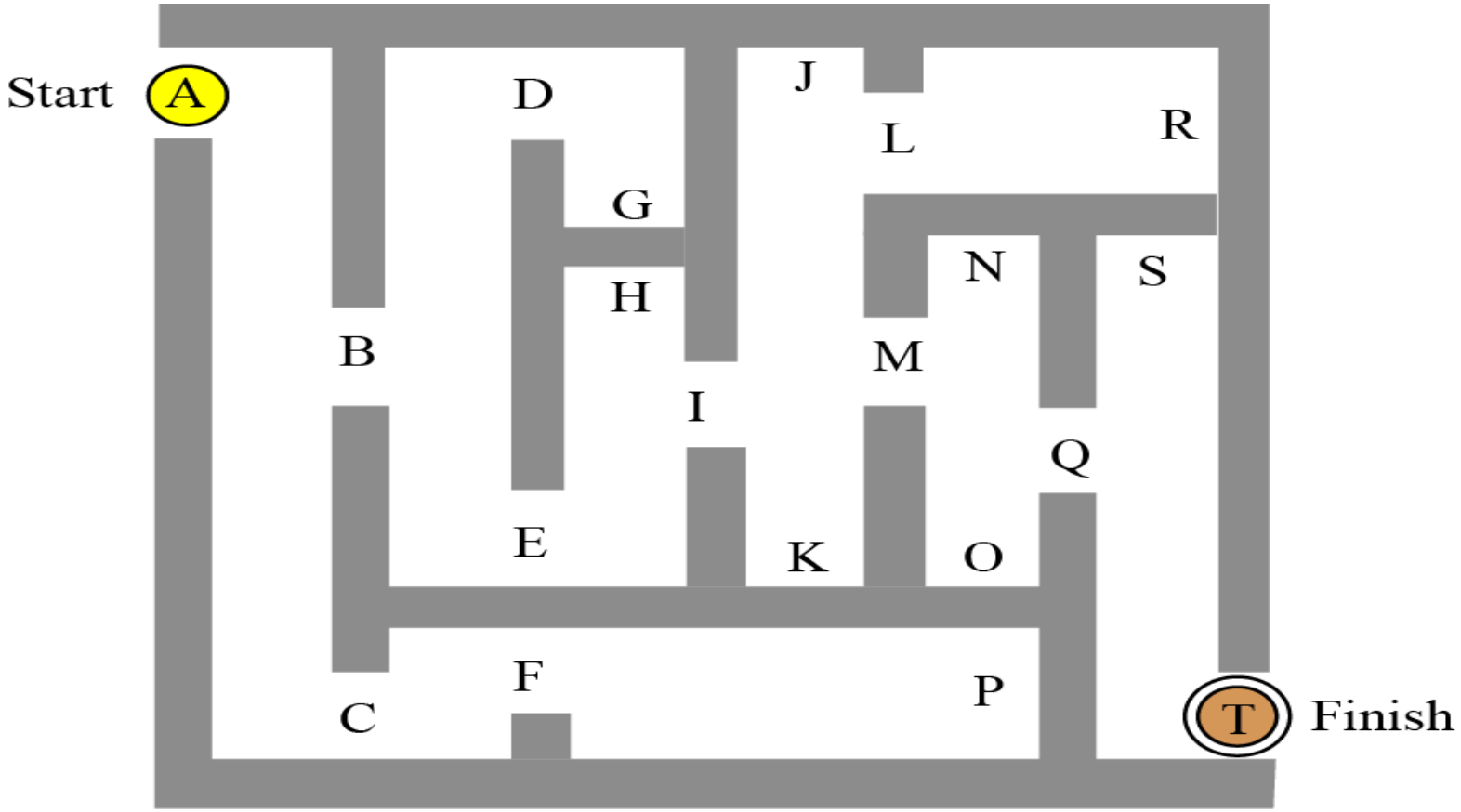
Expand:
fringe=[D,E,F,G]

Is D a goal state?



Example : The Maze Problem

- find our way through the maze in Figure
  - A : Starting Point
  - T : finishing point



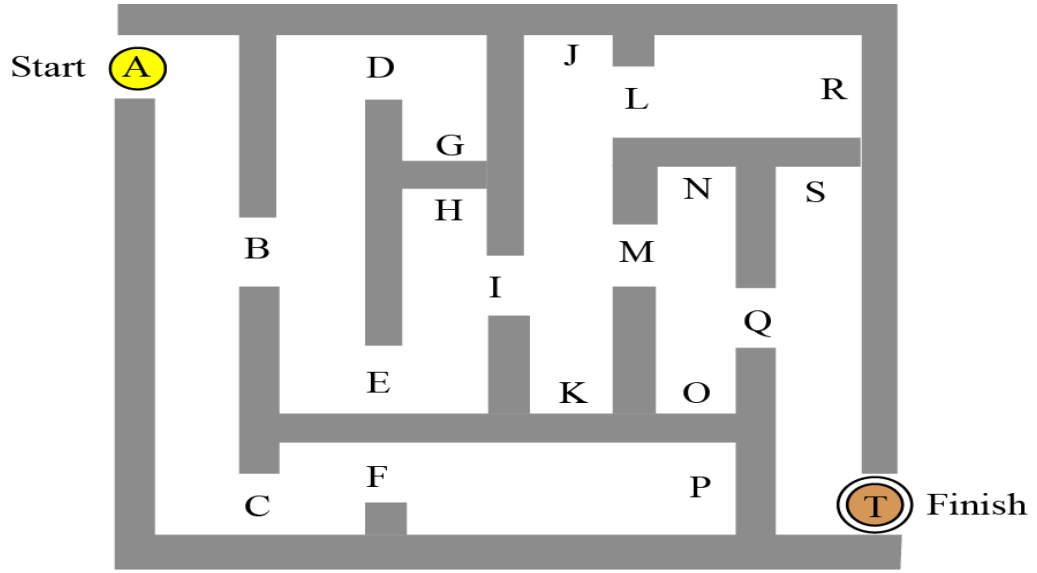
Example : The Maze Problem

The state space is

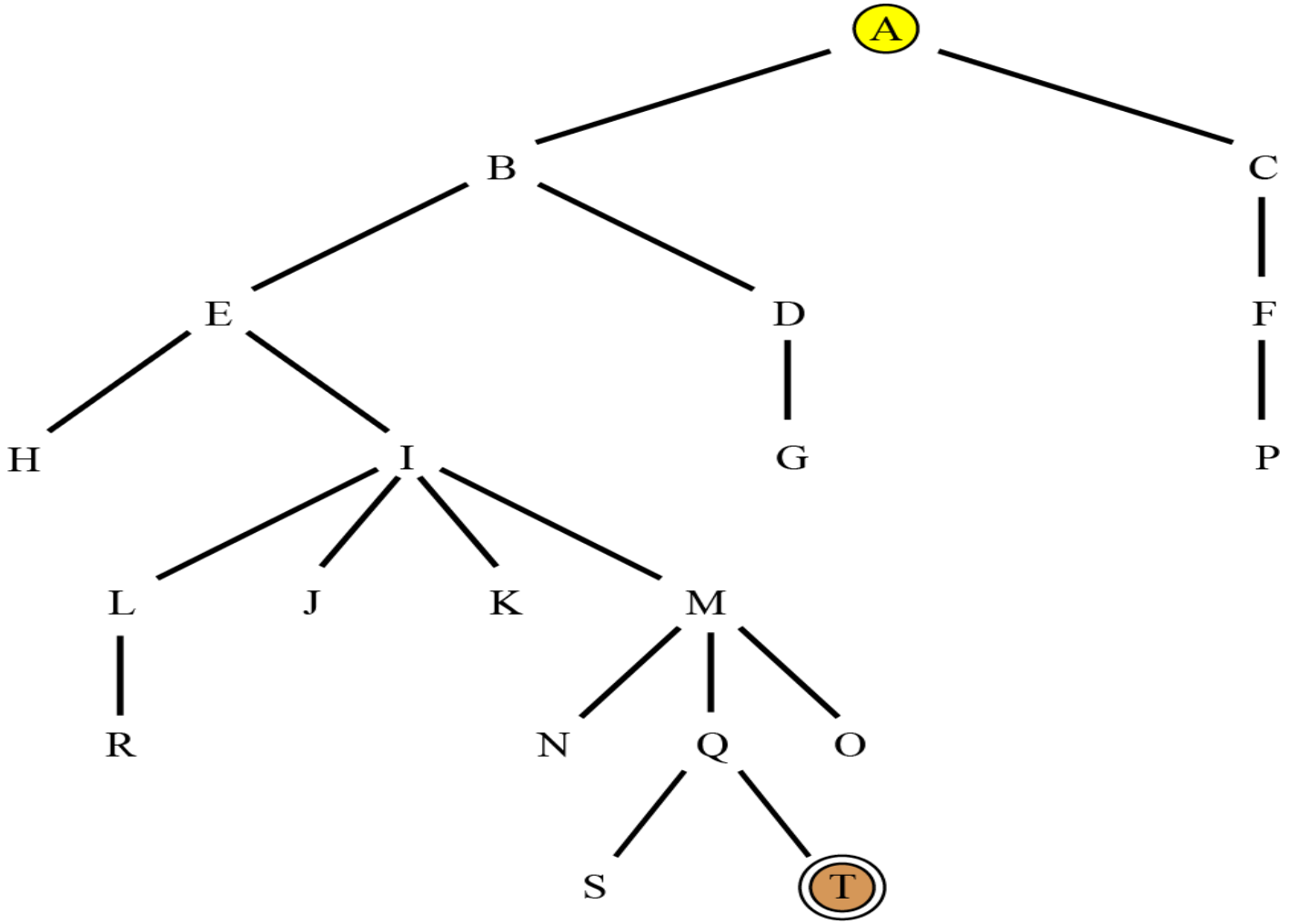
- Initial state :A
- Goal state : T

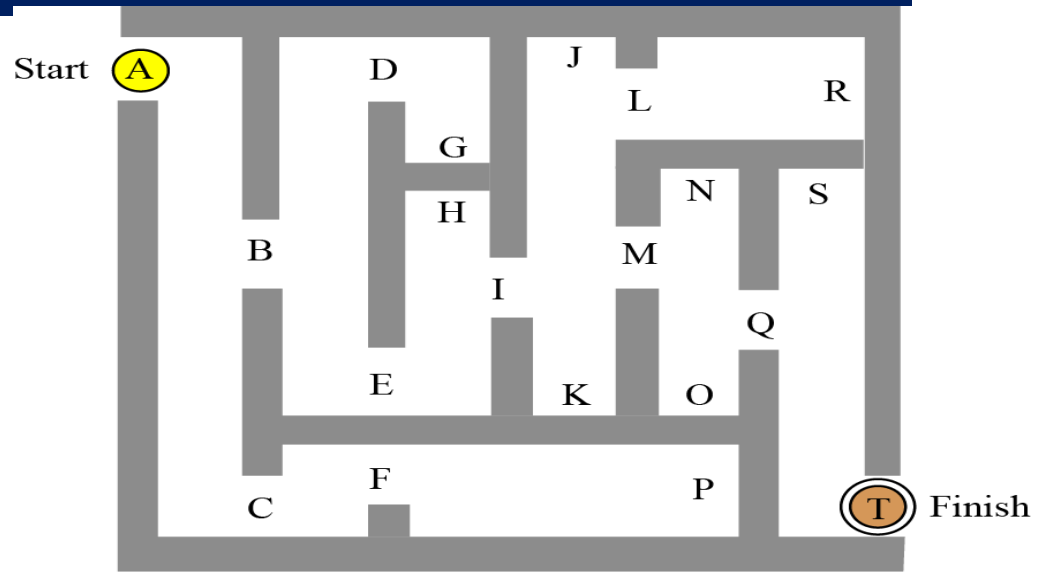
Set of neighboring nodes

e.g. (A,B), (A,C) , (B,D), (B,E) ……

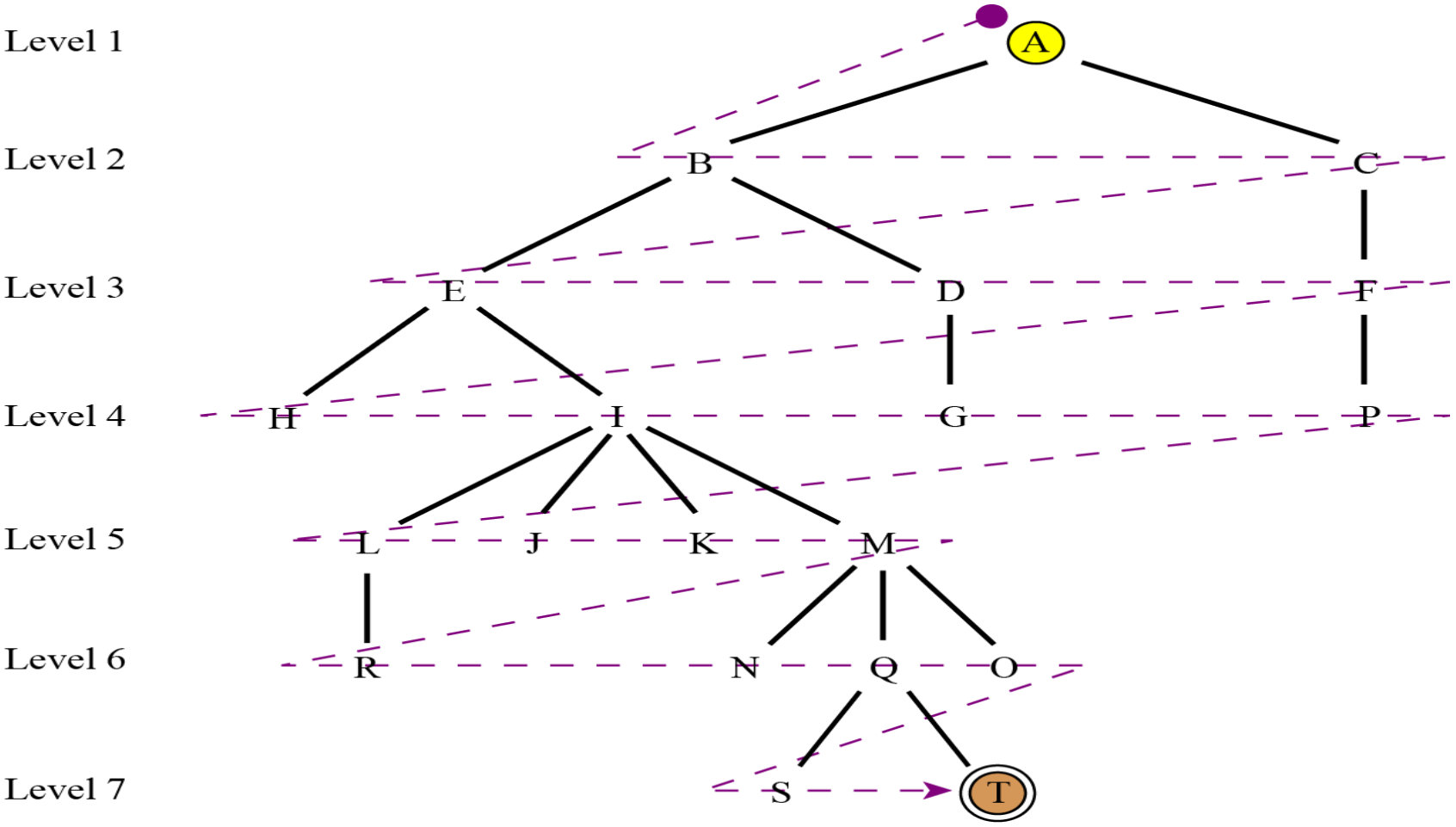


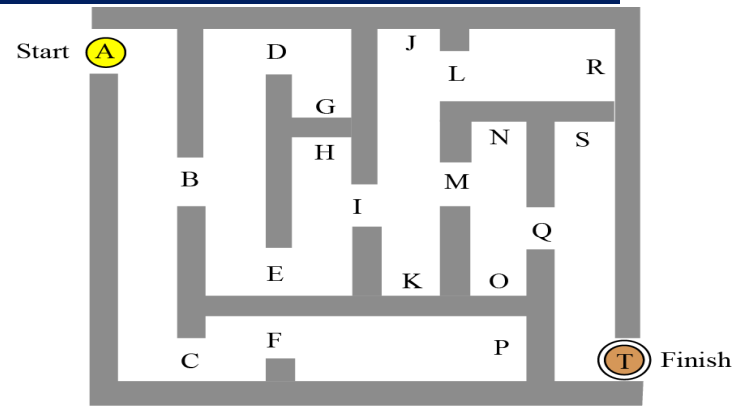
Maze as a graph search Problem



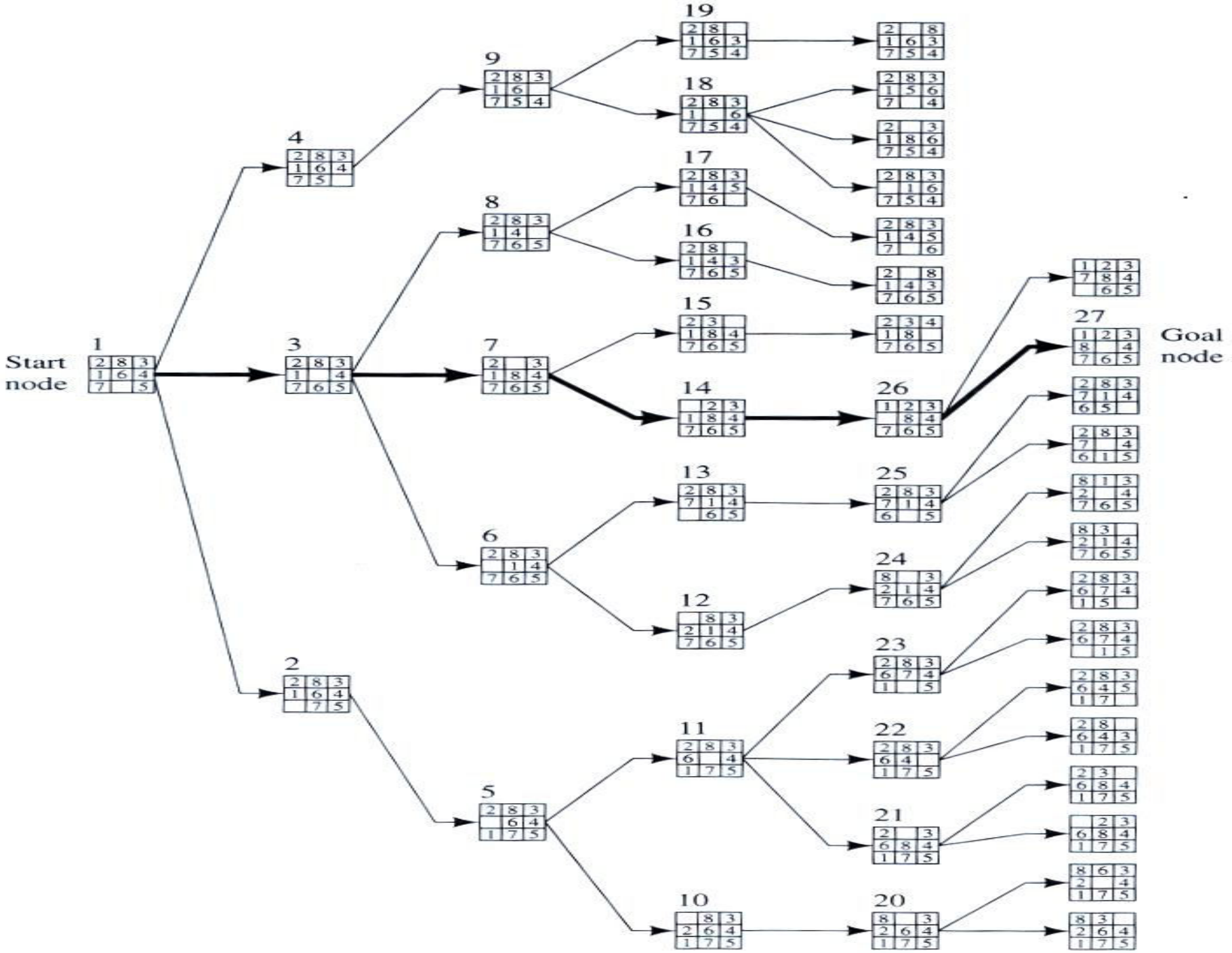


Breadth-first search of the tree





Example BFS



Properties of breadth-first search

- Complete? Yes it always reaches goal (if *b* is finite)
- *d= the depth of the shallowest solution*
- *b = number of nodes in level i*
- Time? $1+b+b^2+b^3+\ldots+b^d + (b^{d+1}-b)) = O(b^{d+1})$

  (this is the number of nodes we generate)
- Space? $O(b^{d+1})$ (keeps every node in memory, either in fringe or on a path to fringe).
- Fringe Space ? O(b X d)
- Optimal? Yes (if we guarantee that deeper solutions are less optimal, e.g. step-cost=1).

- Space is the bigger problem (more than time)

Properties of breadth-first search (Self study)

- Since it never generates a node in the tree until all the nodes at shallower levels have been generated
  - breadth-first search always finds a shortest path to a goal.
- Since each node can be generated in constant time,
  - the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d.
- Since the number of nodes at level d is b^d, the total number of nodes generated in the worst case is $b + b^2 + b^3 + \ldots + b^d$
  - i.e. $O(b^d)$, the asymptotic time complexity of breadth first search.

Properties of breadth-first search

- **Advantages of BFS**
- It will never get trapped exploring the useless path forever.
  - If there is a solution, BFS will definitely find it out.
  - If there is more than one solution then BFS can find the minimal one that requires less number of steps.
- **Disadvantages of BFS**
- The main drawback is its memory requirement.
  - Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is $O(b^d)$.
  - As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes.
  - If the solution is farther away from the root, breath first search will consume lot of time.

Brute force Techniques

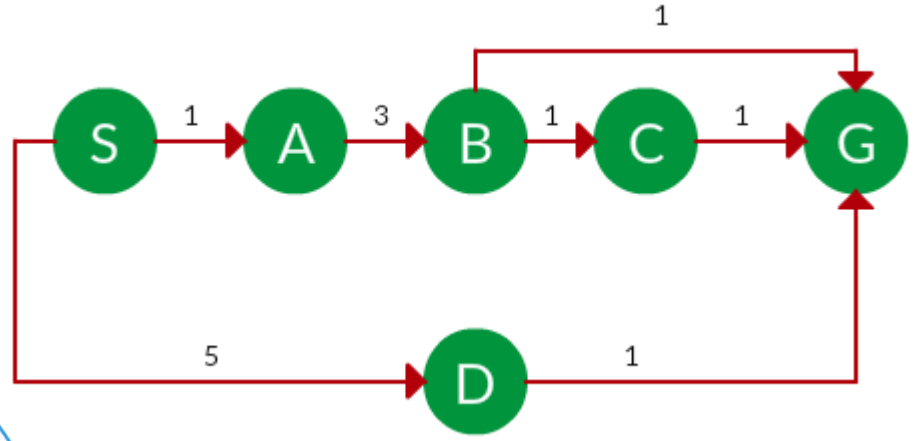
- **Generate-And-Test**
- **Breadth-First Search**
- **Uniform-Cost Search**
- Depth-First Search
- Depth-First Iterative-Deepening Search
- Bidirectional Search

Uniform-Cost Search Algorithm

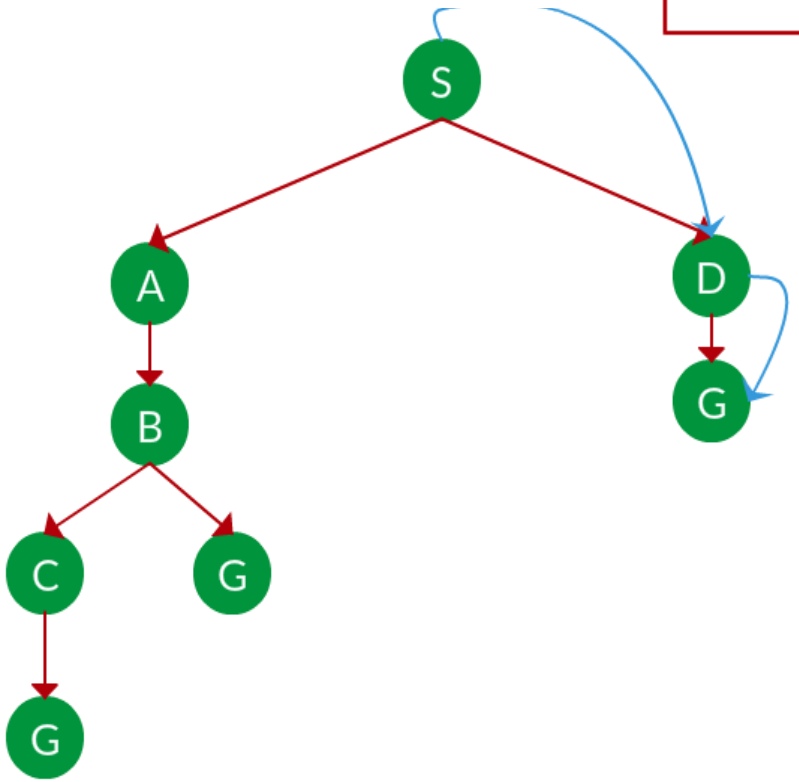
- If all the edges in the search graph do not have the same cost then BFS is generalizes to uniform-cost search
  - traversing via different edges might not have the same cost.
  - goal is to find a path where the cumulative sum of costs is the least.
- **Cost of a node** is defined as:
  - cost(node) = cumulative cost of all nodes from root
  - cost(root) = 0
- **Fringe :** The nodes are stored in **a priority queue.**
- also known as **Dijkstra's single-source shortest algorithm**.

UCS Example

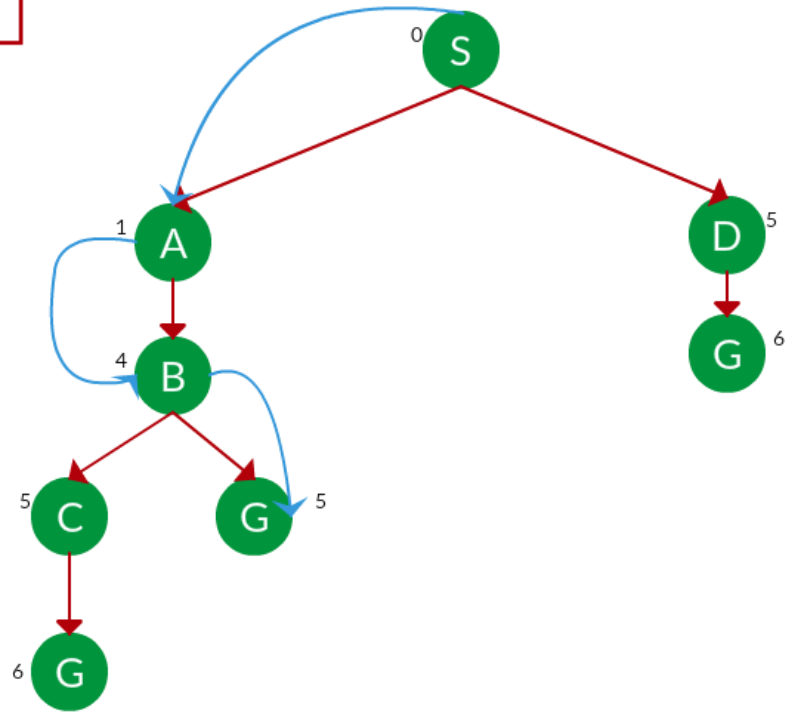
Path: S → G



BFS Path: S →D→ G



Path: S → A → B → G



UCS Example

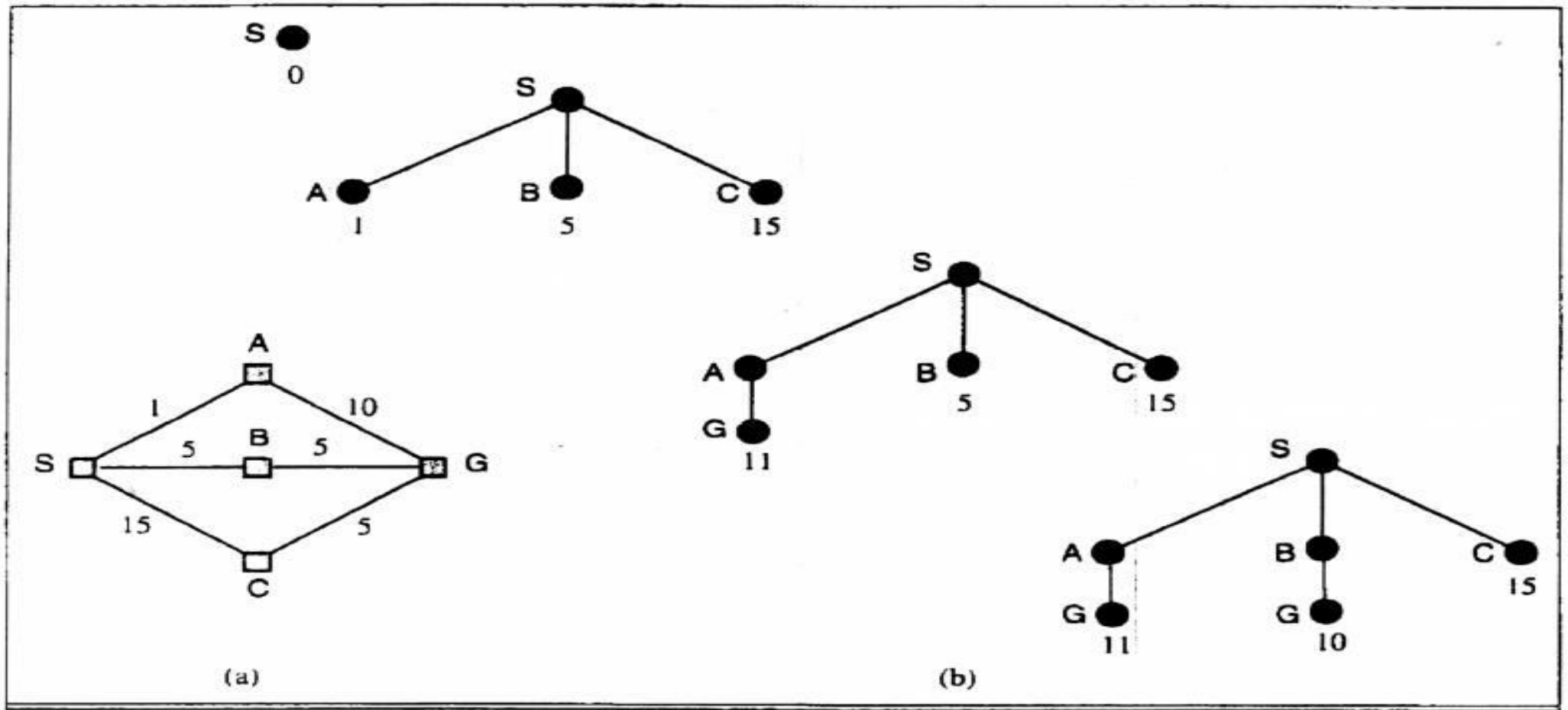


Figure 3.13 A route-finding problem. (a) The state space, showing the cost for each operator. (b) Progression of the search. Each node is labelled with $g(n)$. At the next step, the goal node with $g = 10$ will be selected.

UCS Complexity Analysis

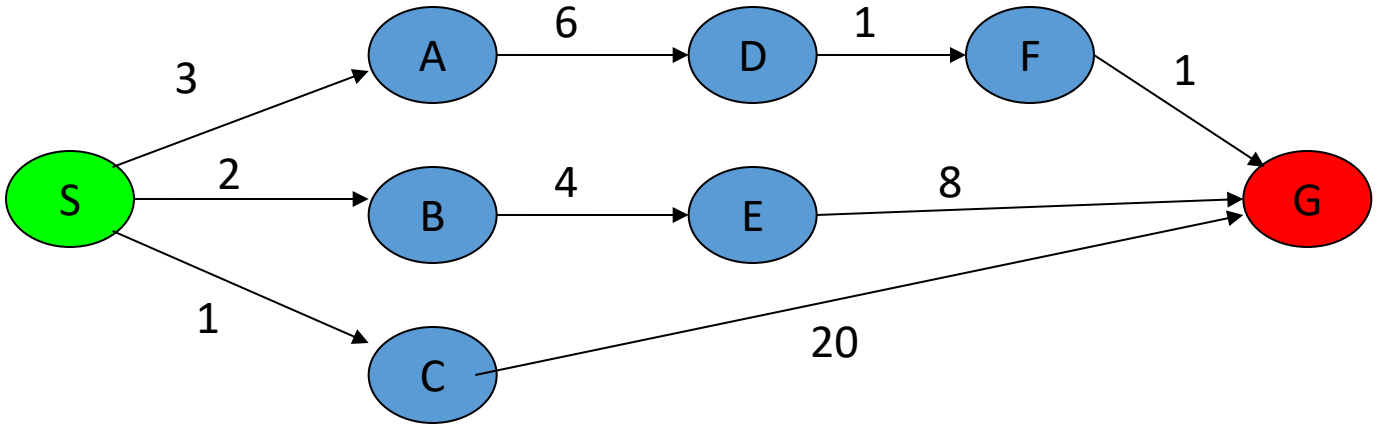
- UCS expands nodes in order of their cost from the root.
  - At each step, the next step n to be expanded is one whose cost g(n) is lowest where g(n) is the sum of the edge costs from the root to node n.
- implementation: *fringe* = queue ordered by path cost
  - Equivalent to breadth-first if all step costs all equal.
- Complete? Yes, if step cost ≥ ε (otherwise it can get stuck in infinite loops)
- Time? No. of nodes with *path cost* ≤ cost of optimal solution.
  - The worst case time complexity of UCS : $O(b^c/m)$,
  - where c is the cost of an optimal solution and m is the minimum edge cost.
- Space? No. of nodes on paths with path cost ≤ cost of optimal solution
  - Same memory limitation as BFS.
- Optimal? Yes, for any step cost ≥ ε

Uniform-cost search (Self study)

- Proof Completeness:
  - Given
    - that every step will cost more than 0,
    - and assuming a finite branching factor,
  - finite number of expansions required before the total path cost is equal to the path cost of the goal state
  - Hence, we will reach it.
- Proof of optimality given completeness:
  - Assume UCS is not optimal.
    - Then there must be a goal state with path cost smaller than the goal state which was found (invoking completeness).
  - However, **this is impossible** because UCS would have expanded that node first by definition Contradiction.

Pros and Cons of UCS

- **Advantages:**
  - UCS is complete only if states are finite and there should be no loop with zero weight.
  - UCS is optimal only if there is no negative cost.
- **Disadvantages:**
  - Explores options in every “direction”.
  - No information on goal location.



straight-line distances

h(S-G)=10
h(A-G)=7
h(D-G)=1
h(F-G)=1
h(B-G)=10
h(E-G)=8
h(C-G)=20

The graph above shows the step-costs for different paths going from the start (S) to the goal (G). On the right you find the straight-line distances.

Use uniform cost search to find the optimal path to the goal.

Exercise for at home

Brute force Techniques

- **Generate-And-Test**
- **Breadth-First Search**
- **Uniform-Cost Search**
- **Depth-First Search**
- Depth-First Iterative-Deepening Search
- Bidirectional Search

Depth First Search Algorithm

- Depth First Search (DFS) searches deeper into the problem space.
  - BFS search always generates successor of the deepest unexpanded node (FIFO-QUEUE).
  - DFS uses last-in first-out stack for keeping the unexpanded nodes (stack).

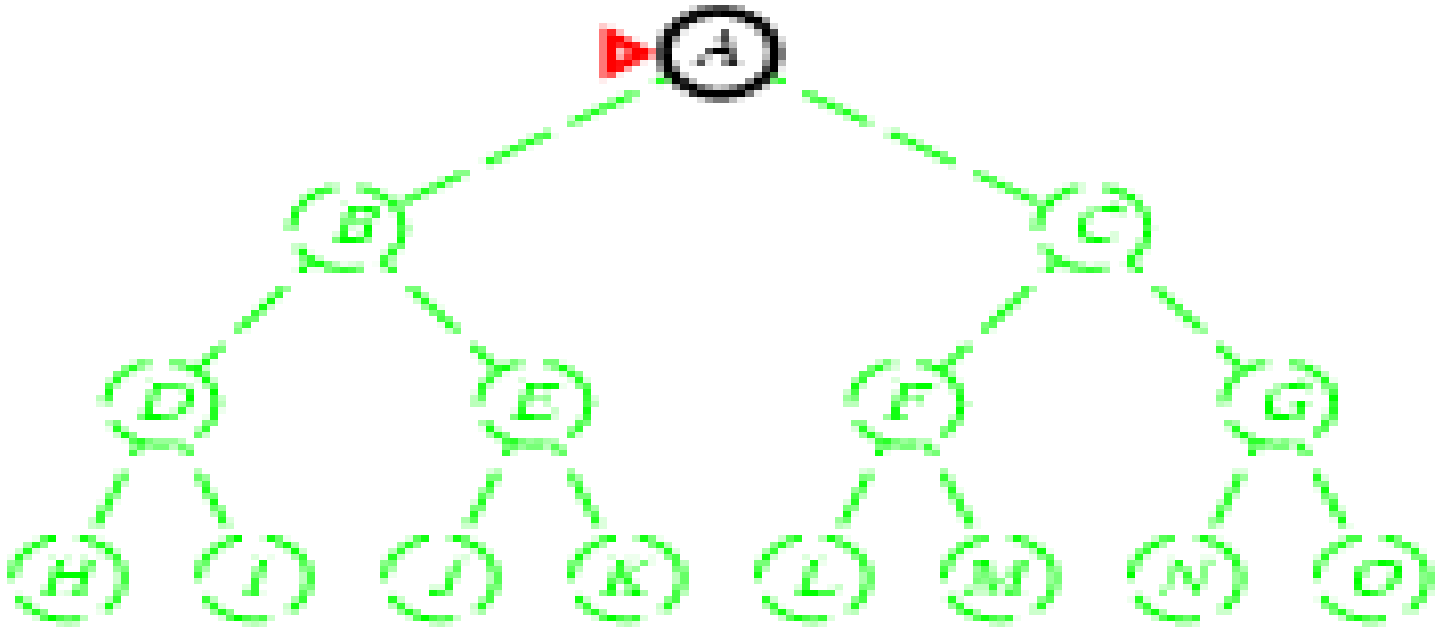
Algorithm: Depth First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, loop until success or failure is signaled.
   a. Generate a state, say E, and let it be the successor of the initial state. If there is no successor, 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

- Expand *deepest* unexpanded node
- Implementation:
  - *fringe* = Last In First Out (LIFO) queue, i.e., put successors at front

Is A a goal state?

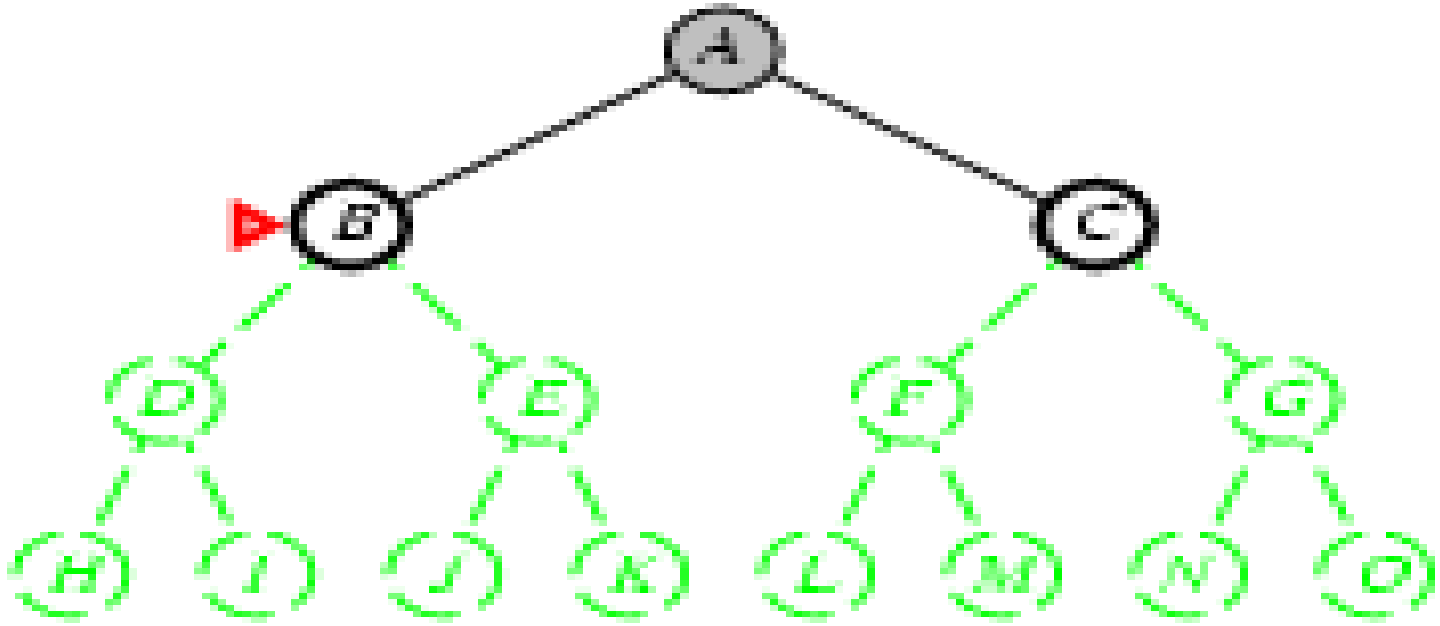


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[B,C]

Is B a goal state?

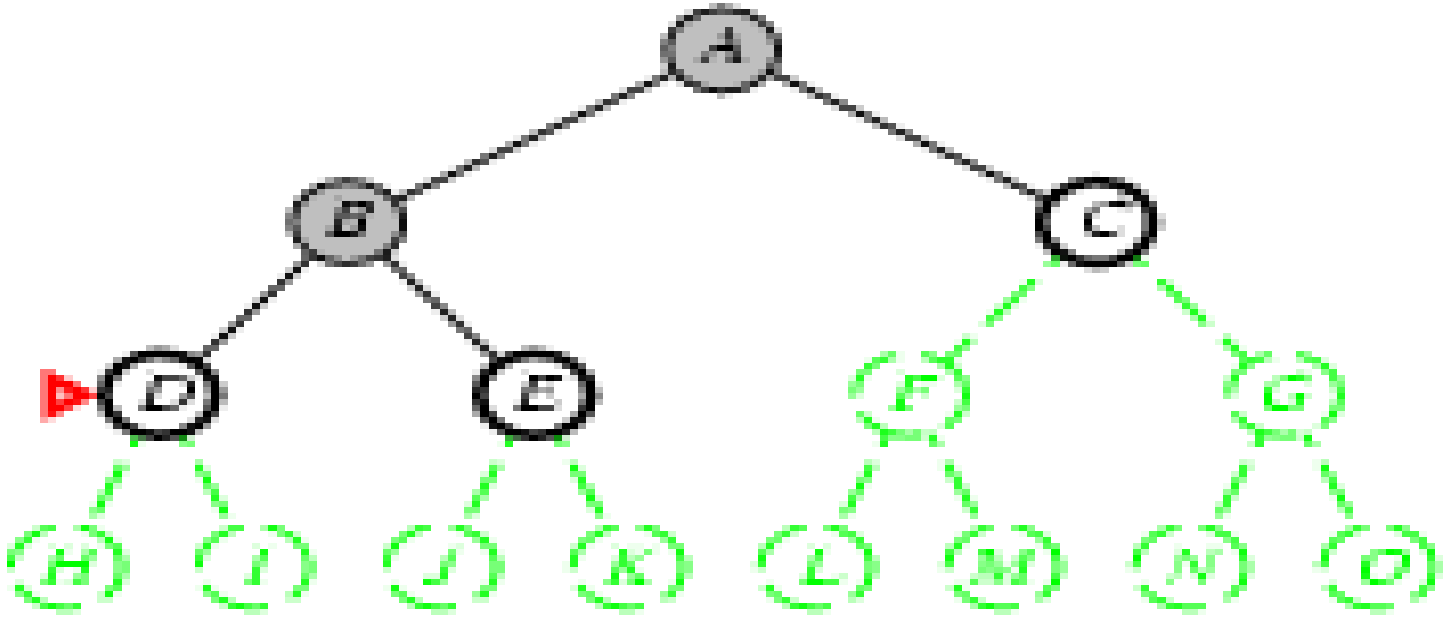


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[D,E,C]

Is D = goal state?

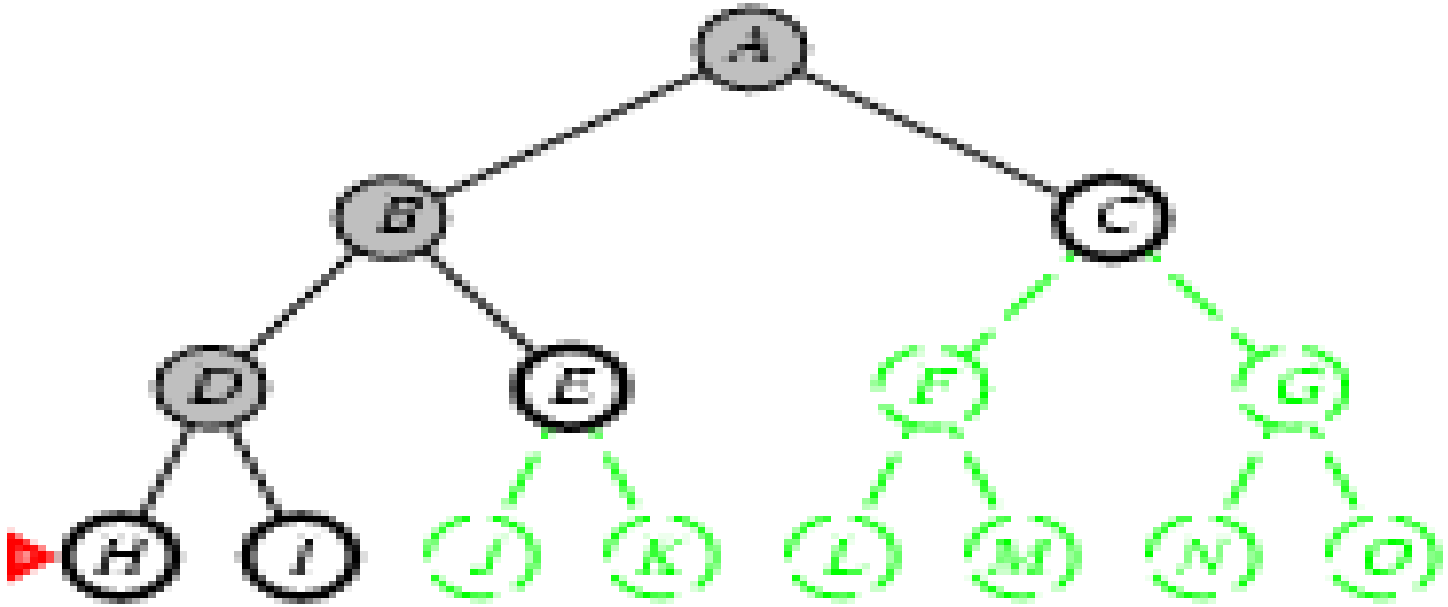


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[H,I,E,C]

Is H = goal state?

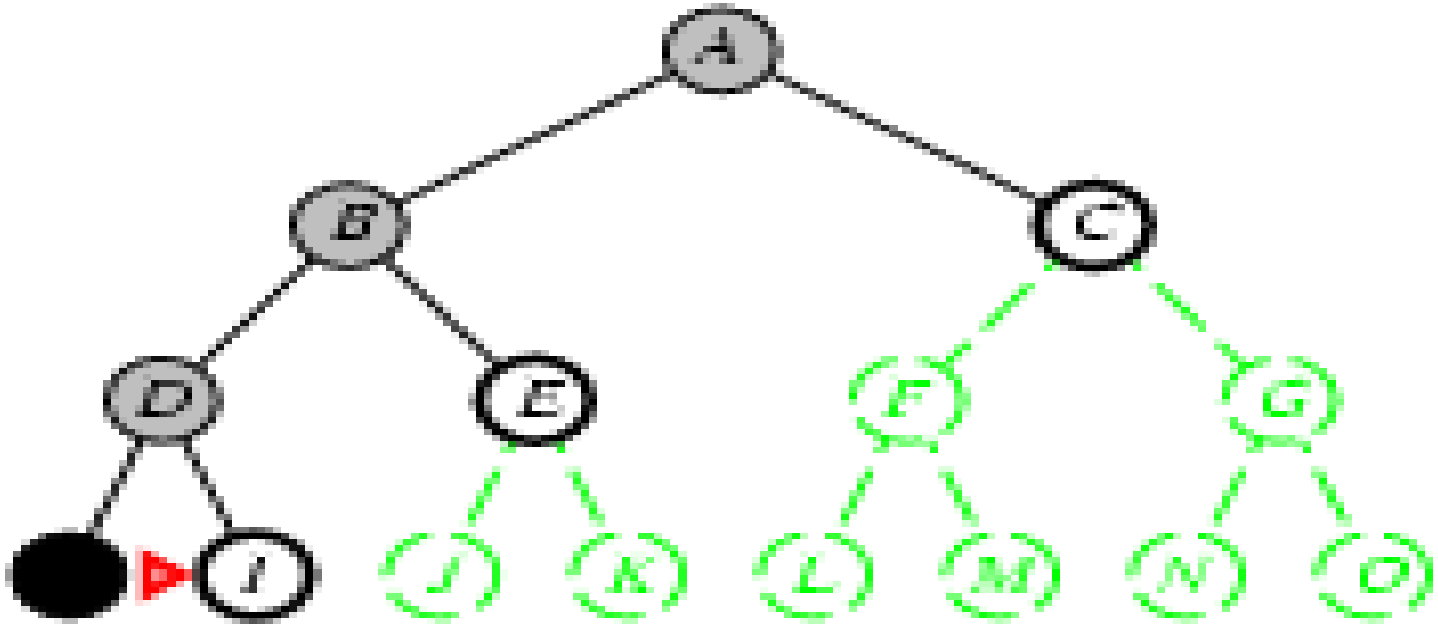


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[I,E,C]

Is I = goal state?

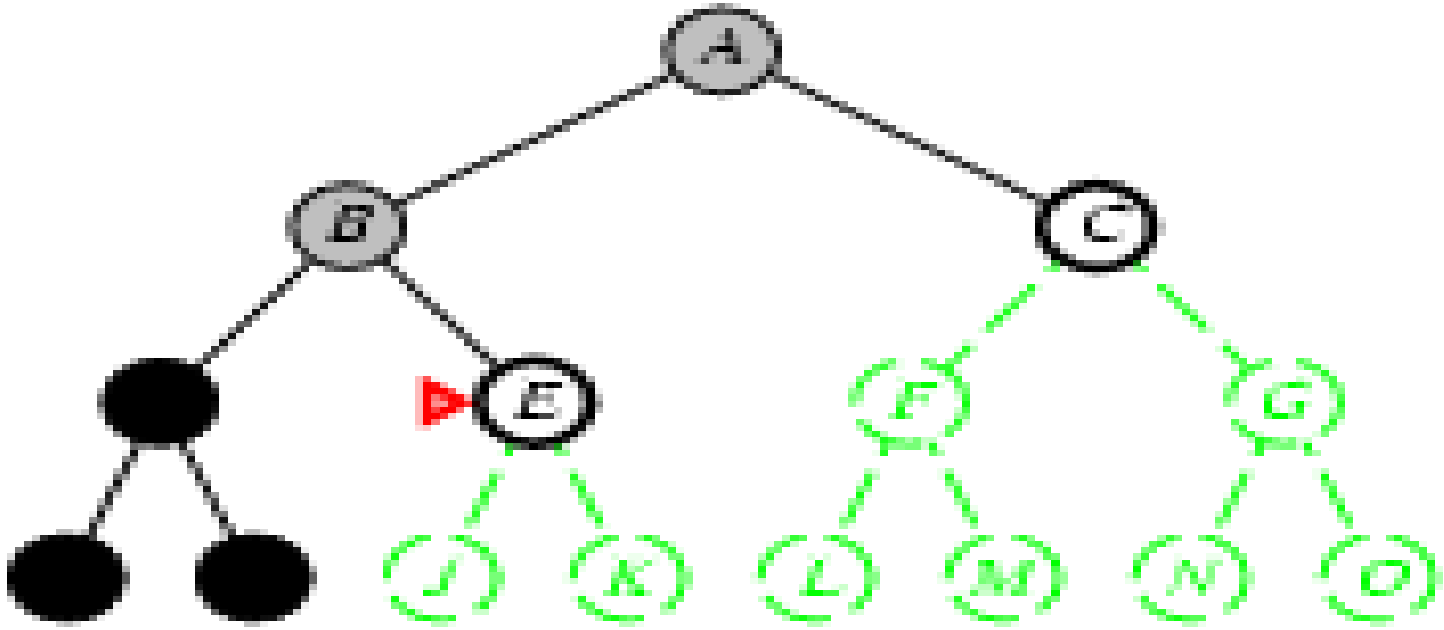


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[E,C]

Is E = goal state?

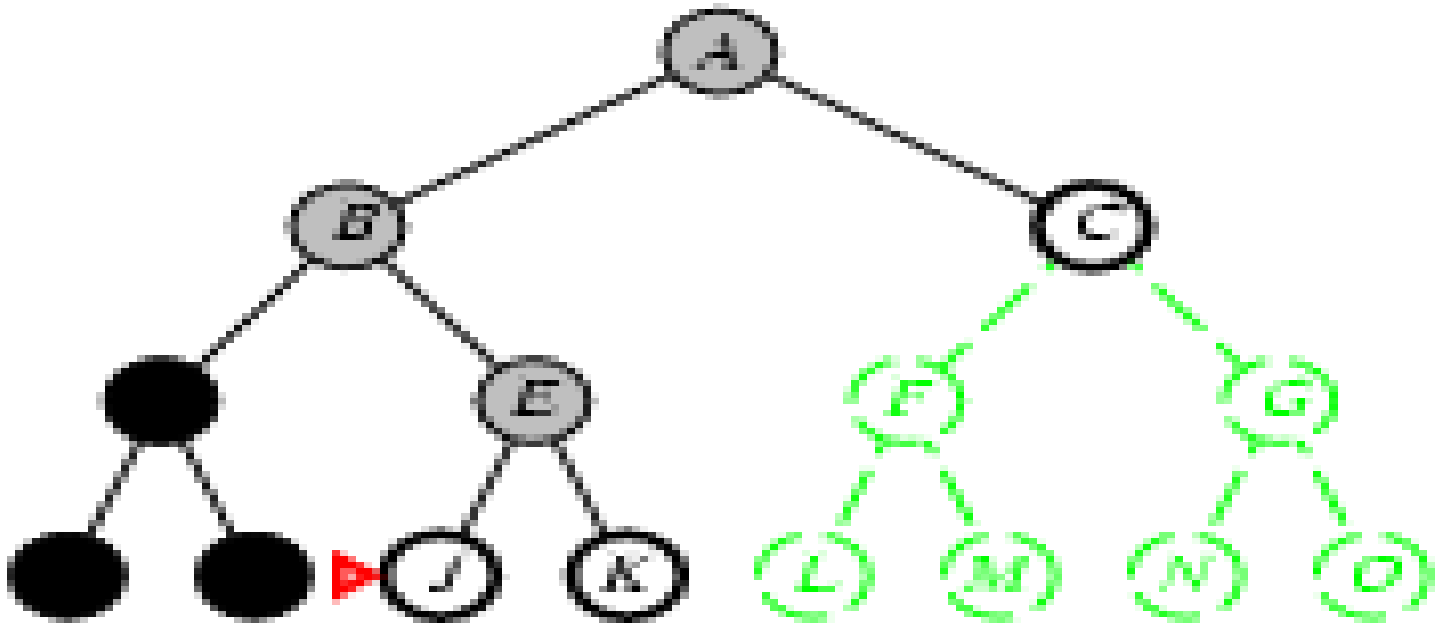


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[J,K,C]

Is J = goal state?

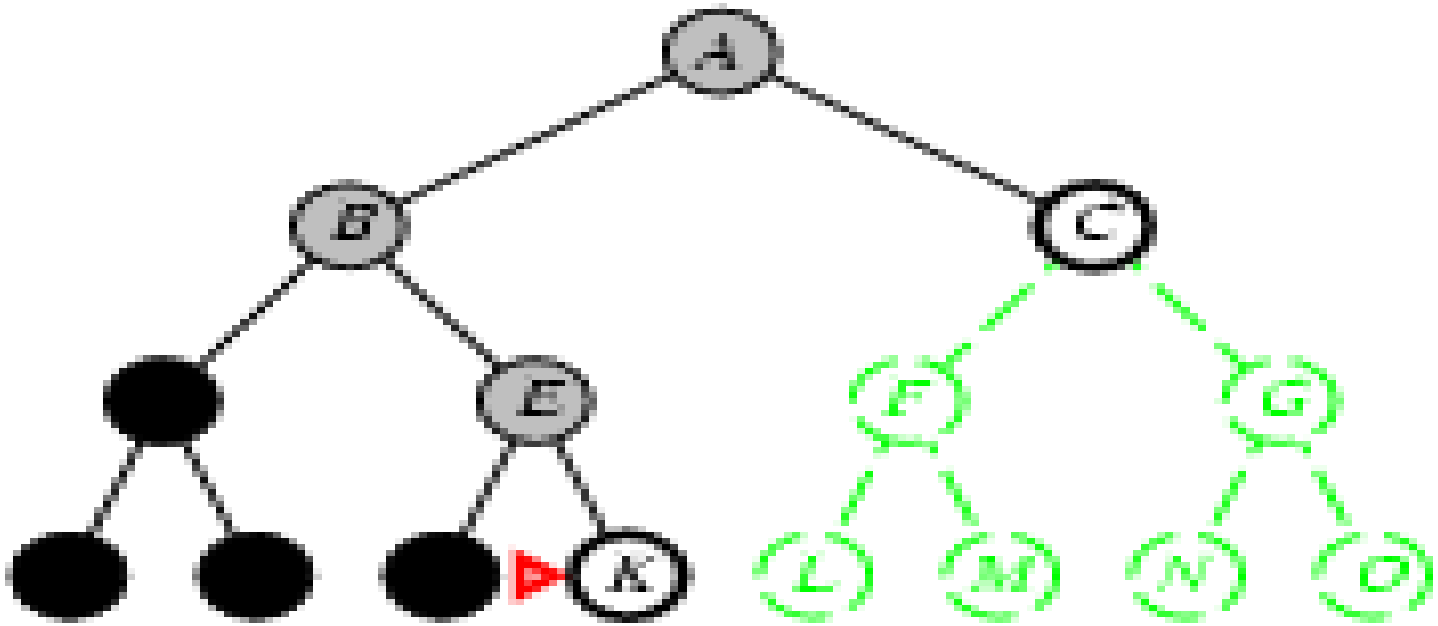


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[K,C]

Is K = goal state?

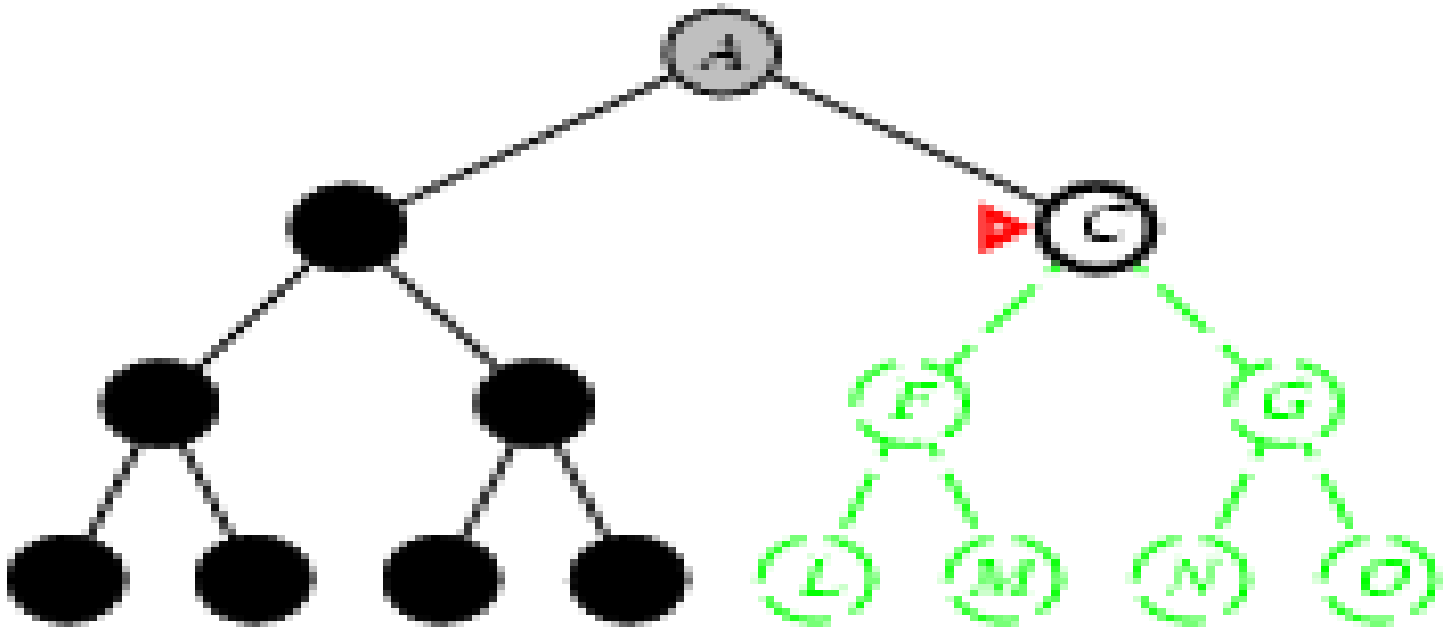


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[C]

Is C = goal state?

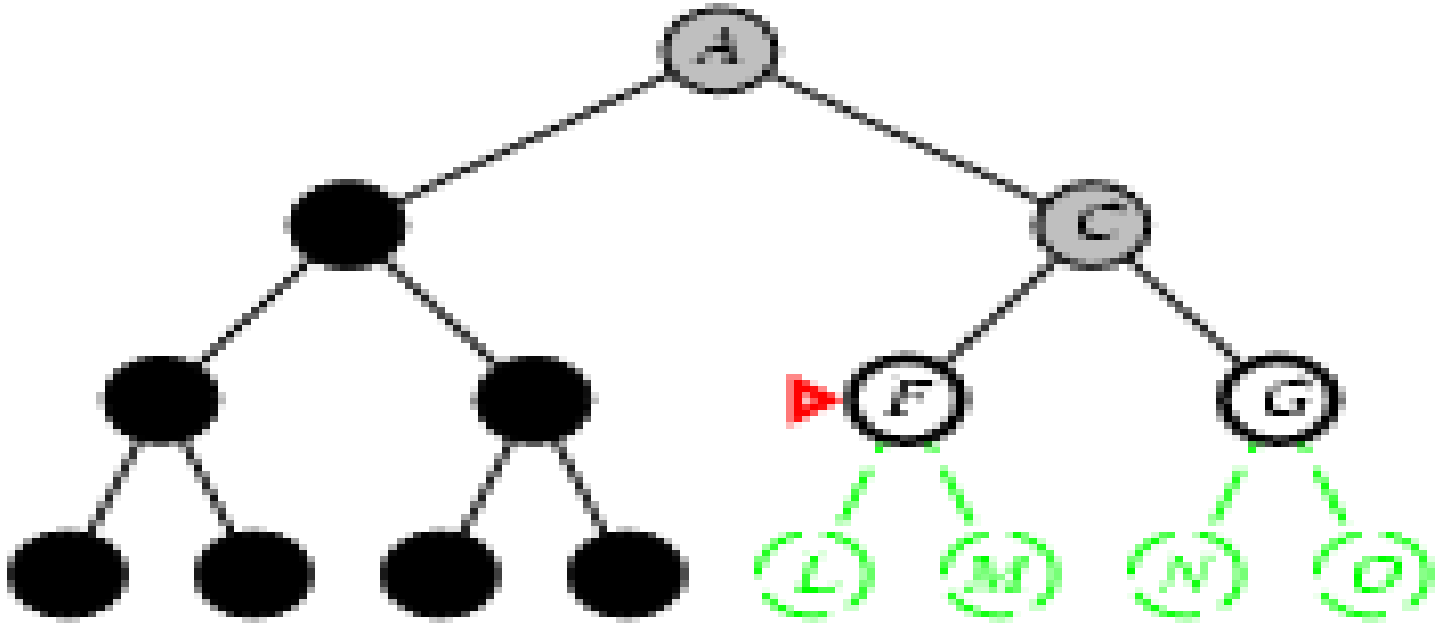


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[F,G]

Is F = goal state?

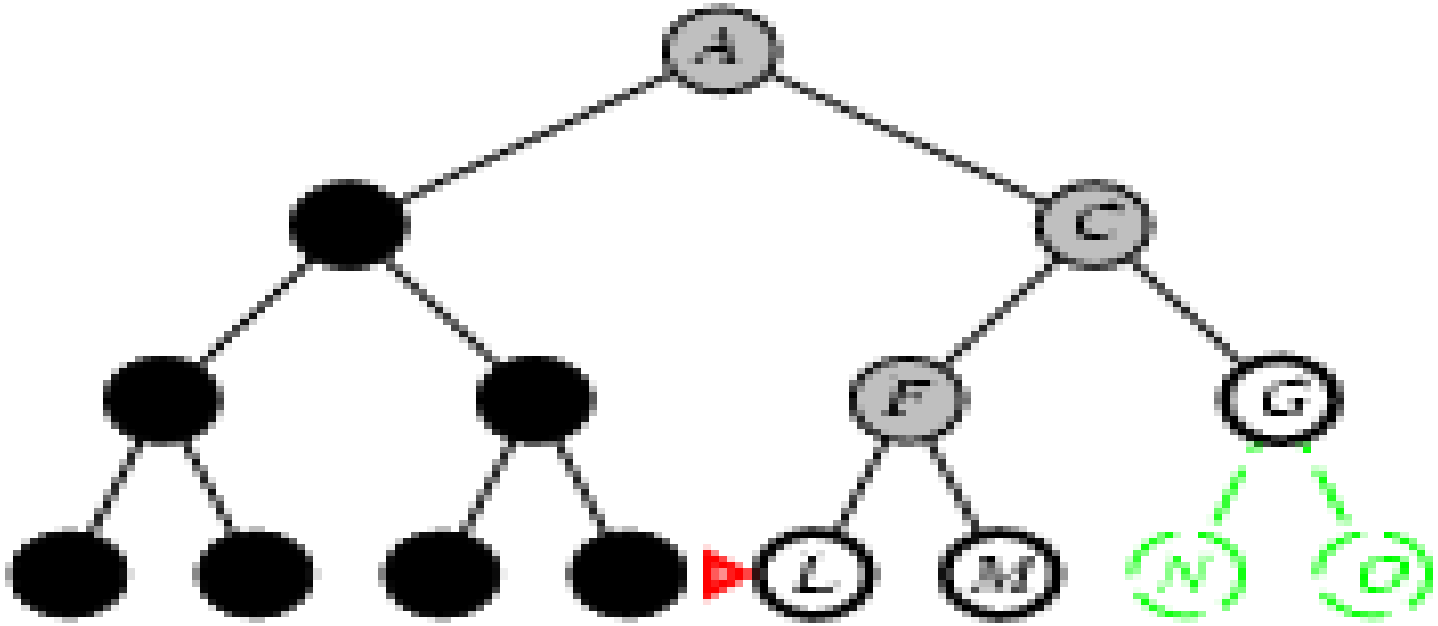


Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[L,M,G]

Is L = goal state?

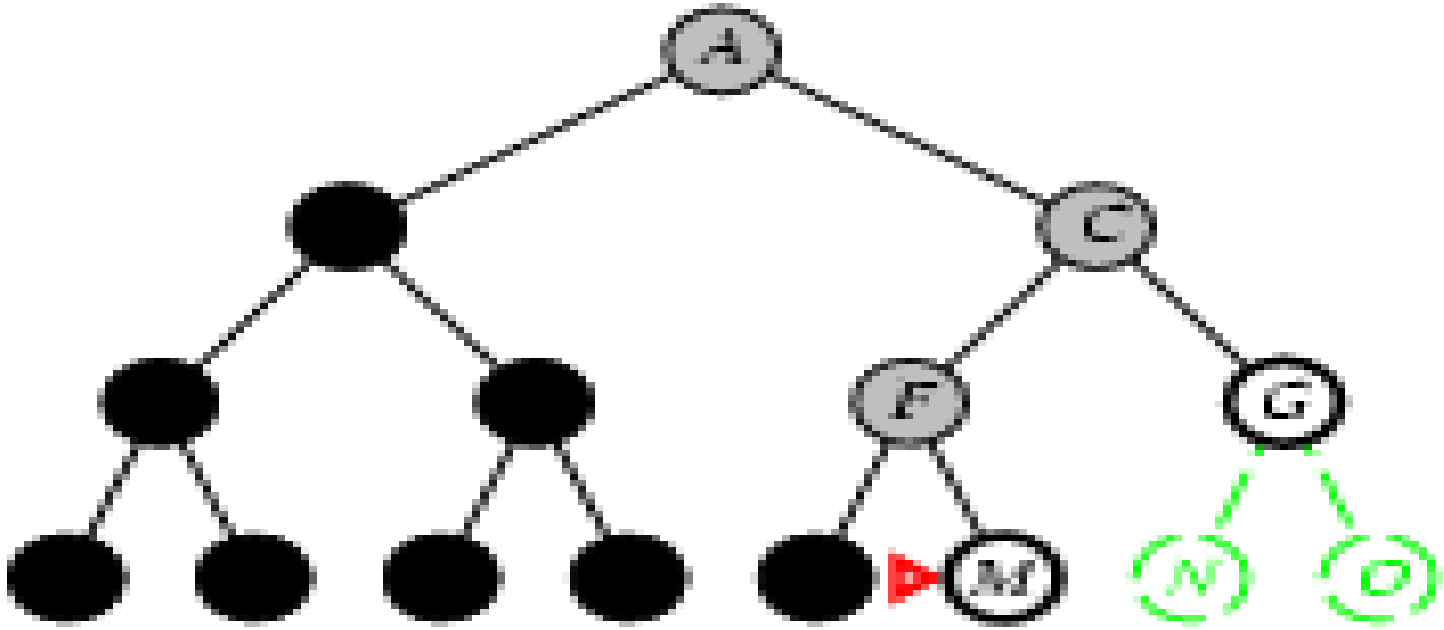


Depth-first search

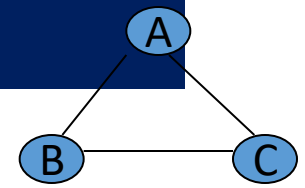
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue, i.e., put successors at front

queue=[M,G]

Is M = goal state?



Properties of depth-first search



- Complete? No: fails in infinite-depth spaces (loop or Cycle)
  - Can modify to avoid repeated states along path
- *d= the depth of the shallowest solution*
- *bi = number of nodes in level I*
- Time? : $T(b) = 1+b+b^2+b^3+\ldots+b^d + (b^{d+1}-b)) = O(b^{d+1})$
  - $O(b^m)$ with m=maximum depth
- terrible if *m* is much larger than *d*
  - but if solutions are dense, may be much faster than breadth-first
- Space? *O(bXm),* i.e., linear space! (we only need to remember a single path + expanded unexplored nodes) (figure in next slide)
- Optimal? No (It may find a non-optimal goal first)

Pros and Cons of DFS

Advantages of Depth-First Search

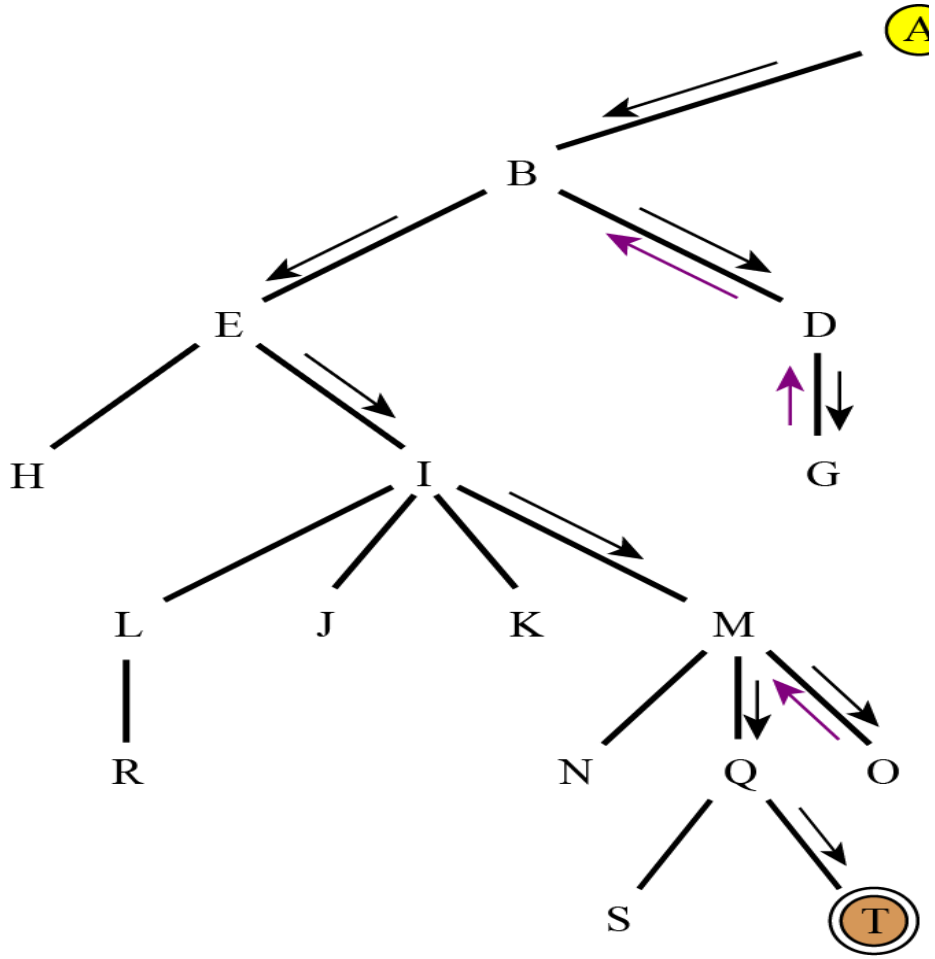
- memory requirement is linear with respect to the search graph.
  - Only needs memory to store a stack of nodes on the path from the root to the current node.
  - The time complexity is O(b^d)
    - As it generates the same set of nodes as BFS, but simply in a different order.
    - Thus practically depth-first search is time-limited rather than space-limited.
- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

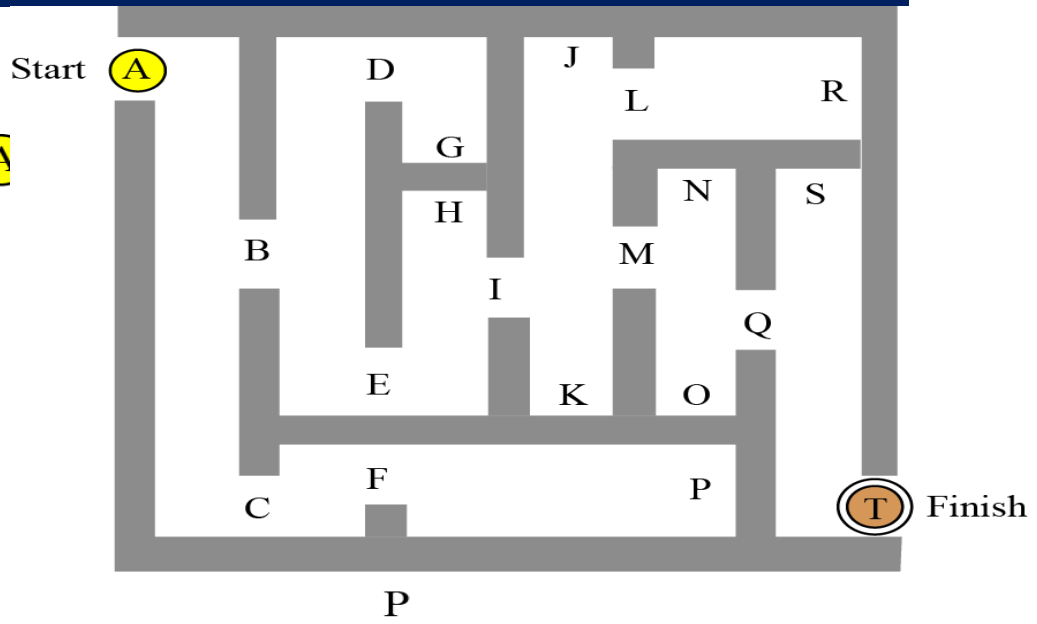
Pros and Cons of DFS (cont…)

Disadvantages of Depth-First Search

- There is a possibility that it may go down the left-most path forever.
  - Even a finite graph can generate an infinite tree.
    - One solution to this problem is to impose a cutoff depth on the search.
    - Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem.
    - If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
- It is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution
  - if more than one solution exists.

Depth-first search of the tree





Brute force Techniques

- **Generate-And-Test**
- **Breadth-First Search**
- **Uniform-Cost Search**
- **Depth-First Search**
- **Depth-First Iterative-Deepening Search**
- Bidirectional Search