# Problem Solving: State Space Search …

# Problem Solving

- General Problem Solving
  - Production System
  - Water Jug Problem
  - Missionaries and Cannibals Problem
- State-Space Search
- Control Strategies

# Problem Solving

- AI programs have a clean separation of
  - computational components of data,
  - operations & control.
- Search forms the core of many intelligent processes.
- It is useful to structure AI programs in a way that facilitates describing the search process.

# Production System - PS

- PS is a formation for structuring AI programs which facilitates describing search process.
- It consists of
  - Initial or start state of the problem
  - Final or goal state of the problem
  - It consists of one or more databases containing information appropriate for the particular task.
- The information in databases may be structured
  - using knowledge representation schemes.

# Production Rules

- PS contains set of production rules,
  - each consisting of a left side that determines the applicability of the rule and
  - a right side that describes the action to be performed if the rule is applied.
  - These rules operate on the databases.
  - Application of rules change the database.
- A control strategy that specifies the order in which the rules will be applied when several rules match at once.
- One of the examples of Production Systems is an Expert System.

# Advantages of PS

- In addition to its usefulness as a way to describe search, the production model has other advantages as a formalism in AI.
  - It is a good way to model the strong state driven nature of intelligent action.
  - As new inputs enter the database, the behavior of the system changes.
  - New rules can easily be added to account for new situations without disturbing the rest of the system, which is quite important in real-time environment.

# Example : Water Jug Problem

- **Problem statement**:
  - Given two jugs, a 5-gallon and 3-gallon having no measuring markers on them. There is a pump that can be used to fill the jugs with water. How can you get exactly 4 gallons of water into 5-gallon jug.
- **Solution**:
  - State for this problem can be described as the set of ordered pairs of integers (X, Y) such that
    - X represents the number of gallons of water in 5-gallon jug and
    - Y for 3-gallon jug.
  - Start state is (0,0)
  - Goal state is (4, N) for N<=3.

# Production Rules

- Following are the production rules for this problem.

R1: $(X, Y \mid X < 5) \rightarrow (5, Y)$
{Fill 5-gallon jug}

R2: $(X, Y \mid X > 0) \rightarrow (0, Y)$
{Empty 5-gallon jug}

R3: $(X, Y \mid Y < 3) \rightarrow (X, 3)$
{Fill 3-gallon jug}

R4: $(X, Y \mid Y > 0) \rightarrow (X, 0)$
{Empty 3-gallon jug}

R5: $(X, Y \mid X+Y <= 5 \wedge Y > 0) \rightarrow (X+Y, 0)$
{Empty 3- gallon into 5-gallon jug}

# Contd..

R6: $(X, Y \mid X+Y <= 3 \Lambda X > 0) \rightarrow (0, X+Y)$

{Empty 5- gallon into 3-gallon jug}

R7: $(X, Y \mid X+Y >= 5 \Lambda Y > 0) \rightarrow (5, Y-(5-X))$

{Pour water from 3-gallon jug into 5-gallon jug, untill 5-g jug is full}

R8: $(X, Y \mid X+Y >= 3 \Lambda X > 0) \rightarrow (X-(3-Y), 3)$

{Pour water from 5-gallon jug into 3-gallon jug , untill 3-g jug is full }

# Trace of steps involved in solving the water jug problem - First solution

| Rule Applied | 5-g jug | 3-g jug | Step No |
|---|---|---|---|
| Start State | 0 | 0 | |
| 1 | 5 | 0 | 1 |
| 8 | 2 | 3 | 2 |
| 4 | 2 | 0 | 3 |
| 6 | 0 | 2 | 4 |
| 1 | 5 | 2 | 5 |
| 8 | 4 | 3 | 6 |
| Goal State | 4 | - | |

# Trace of steps involved in solving the water jug problem - Second solution

| Rule Applied | 5-g jug | 3-g jug | Step No |
|---|---|---|---|
| Start State | 0 | 0 | |
| 3 | 0 | 3 | 1 |
| 5 | 3 | 0 | 2 |
| 3 | 3 | 3 | 3 |
| 7 | 5 | 1 | 4 |
| 2 | 0 | 1 | 5 |
| 5 | 1 | 0 | 6 |
| 3 | 1 | 3 | 7 |
| 5 | 4 | 0 | 8 |
| Goal State | 4 | - | |

# Missionaries and Cannibals

- *Problem Statement:* Three missionaries and three cannibals want to cross a river. There is a boat on their side of the river that can be used by either one or two persons.
  - How should they use this boat to cross the river in such a way that cannibals never outnumber missionaries on either side of the river? If the cannibals ever outnumber the missionaries (on either bank) then the missionaries will be eaten. How can they all cross over without anyone being eaten?

# Contd…

- PS for this problem can be described as the set of ordered pairs of left and right bank of the river as (L, R) where each bank is represented as a list [nM, mC, B]
  - n is the number of missionaries M, m is the number of cannibals C, and B represents boat.
- Start state: ( [3M, 3C, 1B], [0M, 0C, 0B] ),
  - 1B means that boat is present and 0B means it is not there on the bank of river.
- Goal state: ( [0M, 0C, 0B], [3M, 3C, 1B] )

# Contd…

- Any state: $([n_1 M, m_1 C, 1B], [n_2 M, m_2 C, 0B])$ , with constraints/conditions as $n_1 (\neq 0) \geq m_1$; $n_2 (\neq 0) \geq m_2$; $n_1 + n_2 = 3$, $m_1 + m_2 = 3$
  - By no means, this representation is unique.
  - In fact one may have number of different representations for the same problem.
  - The table on the next slide consists of production rules based on the chosen representation.

**Set of Production Rules**
Applied keeping constrains in mind

| RN | Left side of rule | → | Right side of rule |
|---|---|---|---|
| *Rules for boat going from left bank to right bank of the river* | | | |
| L1 | ([$n_1$M, $m_1$C, 1B], [$n_2$M, $m_2$C, 0B]) | → | ([($n_1$-2)M, $m_1$C, 0B], [($n_2$+2)M, $m_2$C, 1B]) |
| L2 | ([$n_1$M, $m_1$C, 1B], [$n_2$M, $m_2$C, 0B]) | → | ([($n_1$-1)M,($m_1$-1)C,0B],[($n_2$+1)M,($m_2$+1)C, 1B]) |
| L3 | ([$n_1$M, $m_1$C, 1B], [$n_2$M, $m_2$C, 0B]) | → | ([$n_1$M, ($m_1$-2)C, 0B], [$n_2$M, ($m_2$+2)C, 1B]) |
| L4 | ([$n_1$M, $m_1$C, 1B], [$n_2$M, $m_2$C, 0B]) | → | ([($n_1$-1)M, $m_1$C,0B],[($n_2$+1)M, $m_2$C, 1B]) |
| L5 | ([$n_1$M, $m_1$C, 1B], [$n_2$M, $m_2$C, 0B]) | → | ([$n_1$M, ($m_1$-1)C, 0B], [$n_2$M, ($m_2$+1)C, 1B]) |
| *Rules for boat coming from right bank to left bank of the river* | | | |
| R1 | ([$n_1$M, $m_1$C, 0B], [$n_2$M, $m_2$C, 1B]) | → | ([($n_1$+2)M, $m_1$C, 1B], [($n_2$-2)M, $m_2$C, 0B]) |
| R2 | ([$n_1$M, $m_1$C, 0B], [$n_2$M, $m_2$C, 1B]) | → | ([($n_1$+1)M,($m_1$+1)C,1B],[($n_2$-1)M,($m_2$-1)C, 0B]) |
| R3 | ([$n_1$M, $m_1$C, 0B], [$n_2$M, $m_2$C, 1B]) | → | ([$n_1$M, ($m_1$+2)C, 1B], [$n_2$M, ($m_2$-2)C, 0B]) |
| R4 | ([$n_1$M, $m_1$C, 0B], [$n_2$M, $m_2$C, 1B]) | → | ([($n_1$+1)M, $m_1$C,1B],[($n_2$-1)M, $m_2$C, 0B]) |
| R5 | ([$n_1$M, $m_1$C, 0B], [$n_2$M, $m_2$C, 1B]) | → | ([$n_1$M, ($m_1$+1)C, 1B], [$n_2$M, ($m_2$-1)C, 0B]) |

## One of the possible paths

| | | |
|---|---|---|
| **Start →** | **([3M, 3C, 1B], [0M, 0C, 0B]]** | |
| L2: | ([2M, 2C, 0B], [1M, 1C, 1B]) | **1M,1C →** |
| R4: | ([3M, 2C, 1B], [0M, 1C, 0B]) | **1M ←** |
| L3: | ([3M, 0C, 0B], [0M, 3C, 1B]) | **2C →** |
| R4: | ([3M, 1C, 1B], [0M, 2C, 0B]) | **1C ←** |
| L1: | ([1M, 1C, 0B], [2M, 2C, 1B]) | **2M →** |
| R2: | ([2M, 2C, 1B], [1M, 1C, 0B]) | **1M,1C ←** |
| L1: | ([0M, 2C, 0B], [3M, 1C, 1B]) | **2M →** |
| R5: | ([0M, 3C, 1B], [3M, 0C, 0B]) | **1C ←** |
| L3: | ([0M, 1C, 0B], [3M, 2C, 1B]) | **2C →** |
| R5: | ([0M, 2C, 1B], [3M, 1C, 0B]) | **1C ←** |
| L3: | **([0M, 0C, 0B], [3M, 3C, 1B])** | **2C → Goal state** |

# State Space Search

- State space is another method of problem representation that facilitates easy search similar to PS.
- In this method also problem is viewed as finding a path from start state to goal state.
- A solution path is a path through the graph from a node in a set S to a node in set G.
- Set S contains start states of the problem.
- A set G contains goal states of the problem.
- The aim of search algorithm is to determine a solution path in the graph.

# Contd..

- A state space consists of four components.
  - Set of nodes (states) in the graph/tree. Each node represents the state in problem solving process.
  - Set of arcs connecting nodes. Each arc corresponds to operator that is a step in a problem solving process.
  - Set S containing start states of the problem.
  - Set G containing goal states of the problem.

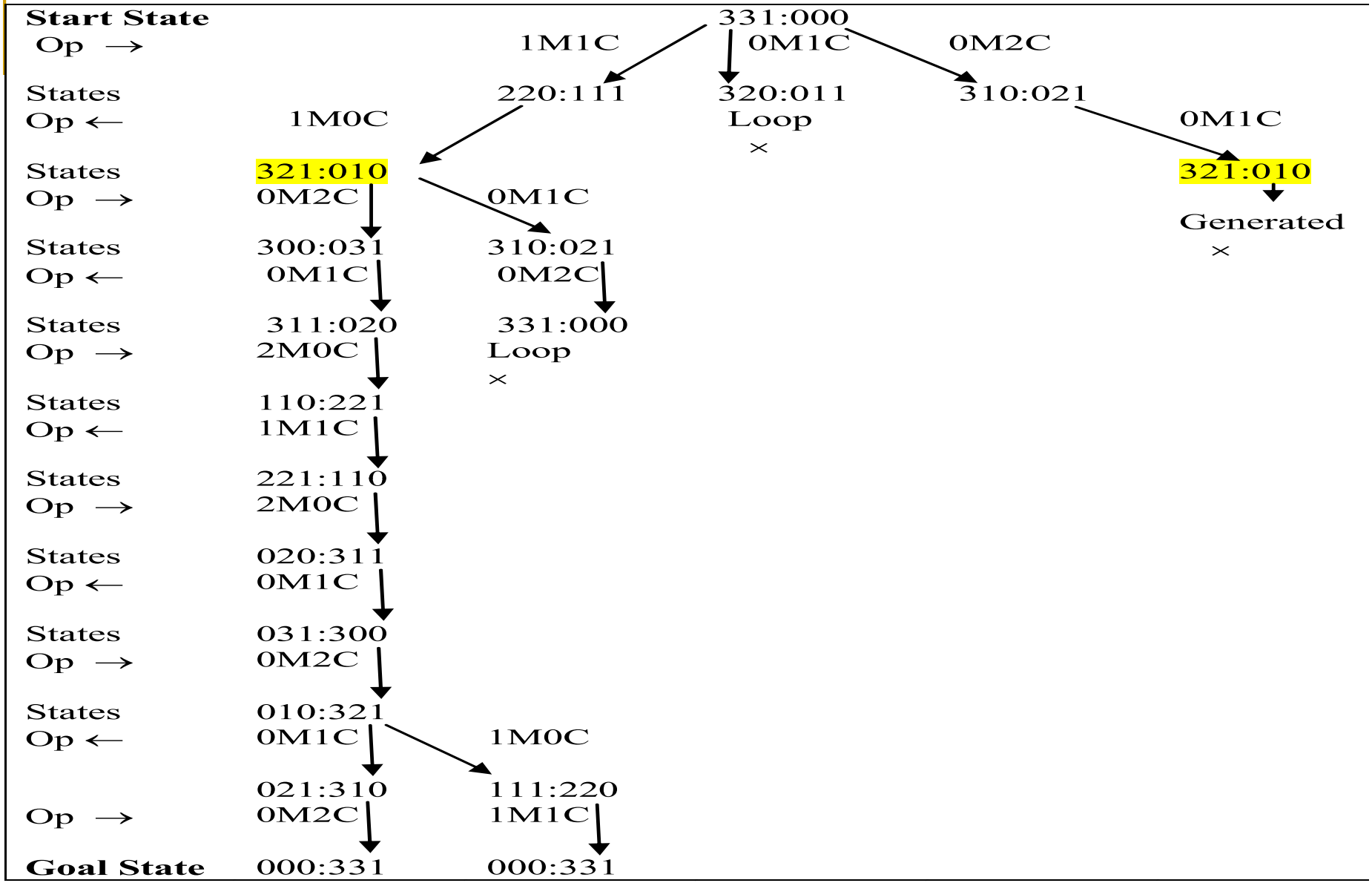
# Missionaries and Cannibals

- The possible operators applied in this problem are {2M0C, 1M1C, 0M2C, 1M0C, 0M1C}.
- Here M is missionary and C is cannibal.
  - Digit before these characters means number of missionaries and cannibals possible at any point in time.
- These operators can be applied in both the situations i.e., if boat is on left bank then we write "Operator $\rightarrow$" and if the boat is on right bank of the river then we write "Operator $\leftarrow$".

- For the sake of simplicity, let us represent state (L:R),where L= $n_1m_1$1 and R = $n_2m_2$0.
- Here boat with 1 or 0 representing presence of absence of the boat.
  - Start state: (331:000)
  - Goal state: (000:331)

# Illegal states, operators

- Invalid state such as (121:210) would lead to one missionary and two cannibals on the left bank which is not a possible state.
- Given a valid state, say, (221:110), the operator 0M1C or 0M2C would be illegal.
- Looping situations are to be avoided.


Start State
331:000
Op →
1M1C
0M1C
0M2C
States
220:111
320:011
310:021
Op ←
1M0C
Loop
×
0M1C
States
321:010
321:010
Op →
0M2C
0M1C
Generated
States
300:031
310:021
×
Op ←
0M1C
0M2C
States
311:020
331:000
Op →
2M0C
Loop
×
States
110:221
Op ←
1M1C
States
221:110
Op →
2M0C
States
020:311
Op ←
0M1C
States
031:300
Op →
0M2C
States
010:321
Op ←
0M1C
1M0C
021:310
111:220
Op →
0M2C
1M1C
Goal State
000:331
000:331

| Two Possible solution paths | |
|---|---|
| *Solution Path 1* | *Solution Path 2* |
| 1M1C → | 1M1C → |
| 1M0C ← | 1M0C ← |
| 0M2C → | 0M2C → |
| 0M1C ← | 0M1C ← |
| 2M0C → | 2M0C → |
| 1M1C ← | 1M1C ← |
| 2M0C → | 2M0C → |
| 0M1C ← | 0M1C ← |
| 0M2C → | 0M2C → |
| 0M1C ← | 1M0C ← |
| 0M2C → | 1M1C → |

# Important Points

- For each problem
  - there is an initial description of the problem.
  - final description of the problem.
  - more than one ways of solving the problem.
  - a path between various solution paths based on some criteria of goodness or on some heuristic function is chosen.
  - there are set of rules that describe the actions called production rules.
    - Left side of the rules is current state and right side describes new state that results from applying the rule.

- Summary: In order to provide a formal description of a problem, it is necessary to do the following things:
  - Define a state space that contains all the possible configurations of the relevant objects.
  - Specify one or more states within that space that describe possible situations from which the problem solving process may start. These states are called **initial states.**
  - Specify one or more states that would be acceptable as solutions to the problem called **goal states.**
  - Specify a set of rules that describe the actions. Order of application of the rules is called control strategy.
  - Control strategy should cause motion towards a solution.

# Problem Characteristics

- In real life, there are three types of problems:
  - Ignorable,
  - Recoverable and
  - Irrecoverable.

# Example - Ignorable

- **(Ignorable)**: **In theorem proving -**
  (solution steps can be ignored)
  - Suppose we have proved some lemma in order to prove a theorem and eventually realized that lemma is no help at all, then ignore it and prove another lemma.
  - Can be solved by using simple **control strategy**?

# Example - Recoverable

- **8 puzzle game** - (solution steps can be undone)
  - Objective of 8 puzzle game is to rearrange a given initial configuration of eight numbered tiles on 3 X 3 board (one place is empty) into a given final configuration (goal state).
    - Rearrangement is done by sliding one of the tiles into empty square.
  - Steps can be undone if they are not leading to solution.
  - Solved by backtracking, so control strategy must be implemented using a **push down stack**.

# Example - Irrecoverable

- **Chess** (solution steps cannot be undone)
  - A stupid move cannot be undone.
  - Can be solved by **planning process.**

# Contd..

- What is the Role of knowledge?
  - In Chess game, knowledge is important to constrain the search
  - Newspapers scanning to decide some facts, a lot of knowledge is required even to be able to recognize a solution.
- Is the knowledge Base consistent?
  - Should not have contradiction

# Contd…

- Is a good solution Absolute or Relative ?
  - In water jug problem there are two ways to solve a problem.
    - If we follow one path successfully to the solution, there is no reason to go back and see if some other path might also lead to a solution.
    - Here a solution is **absolute**.
- In travelling salesman problem, our goal is to find the shortest route. Unless all routes are known, the shortest is difficult to know.
  - This is a best-path problem whereas water jug is any-path problem.

# Contd…

- Any path problem can often be solved in reasonable amount of time using heuristics that suggest good paths to explore.
- Best path problems are in general computationally harder than any-path.

# Control Strategies

- Control Strategy decides which rule to apply next during the process of searching for a solution to a problem.
- Requirements for a good Control Strategy
  - **It should cause motion**

    In water jug problem, if we apply a simple control strategy of starting each time from the top of rule list and choose the first applicable one, then we will never move towards solution.

  - **It should explore the solution space in a systematic manner**

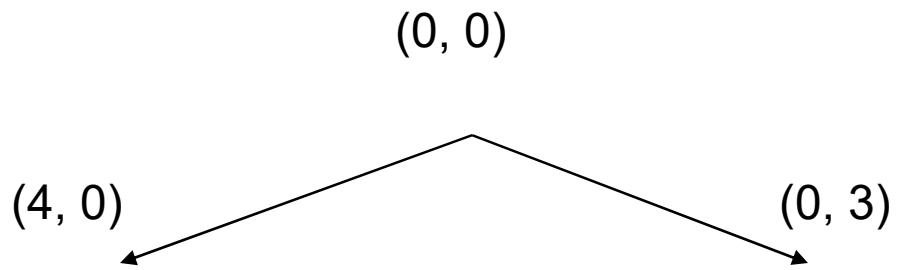
    If we choose another control strategy, say, choose a rule randomly from the applicable rules then definitely it causes motion and eventually will lead to a solution. But one may arrive to same state several times. This is because control strategy is not systematic.

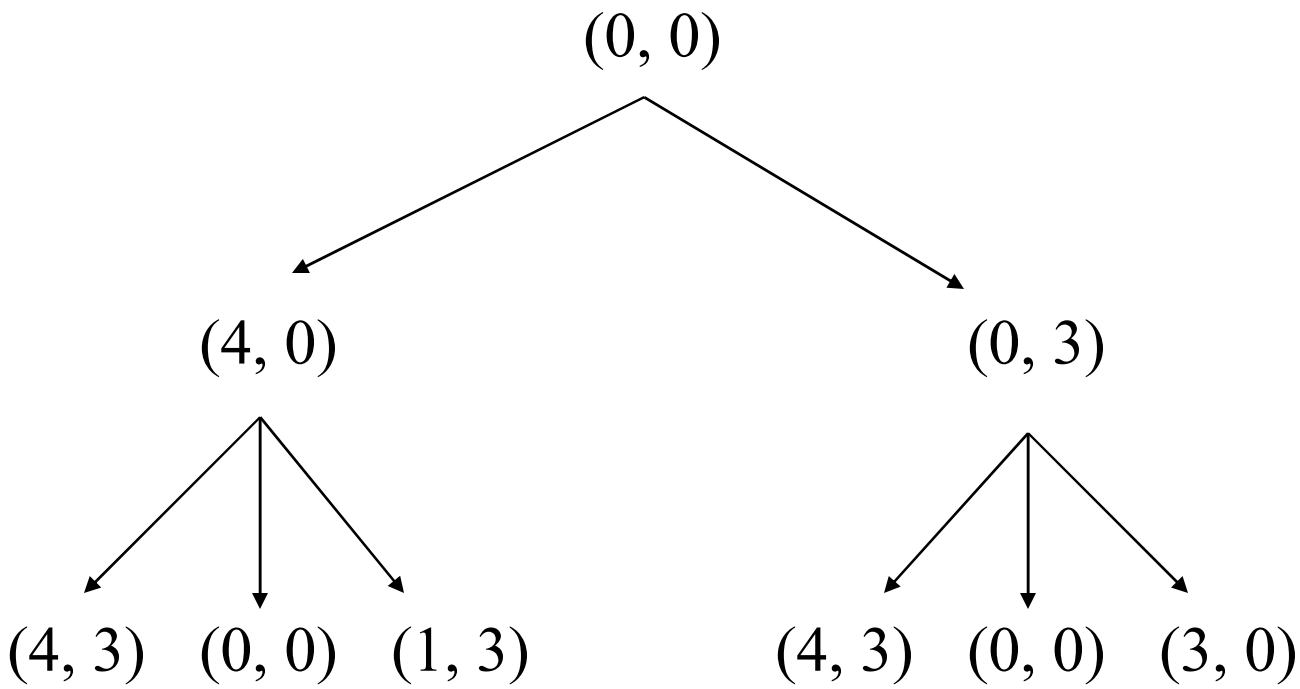
# Systematic Control Strategies (Blind searches)

- Blind searches are exhaustive in nature.
- These are uninformed searches.
- If the problem is simple then
  - any control strategy that causes motion and is systematic will lead to an answer.
- But in order to solve some real world problems,
  - we must use a control strategy that is efficient.
- Let us discuss these strategies using water jug problem.
- These may be applied to any search problem.

# Breadth First Search – BFS : Water Jug Problem

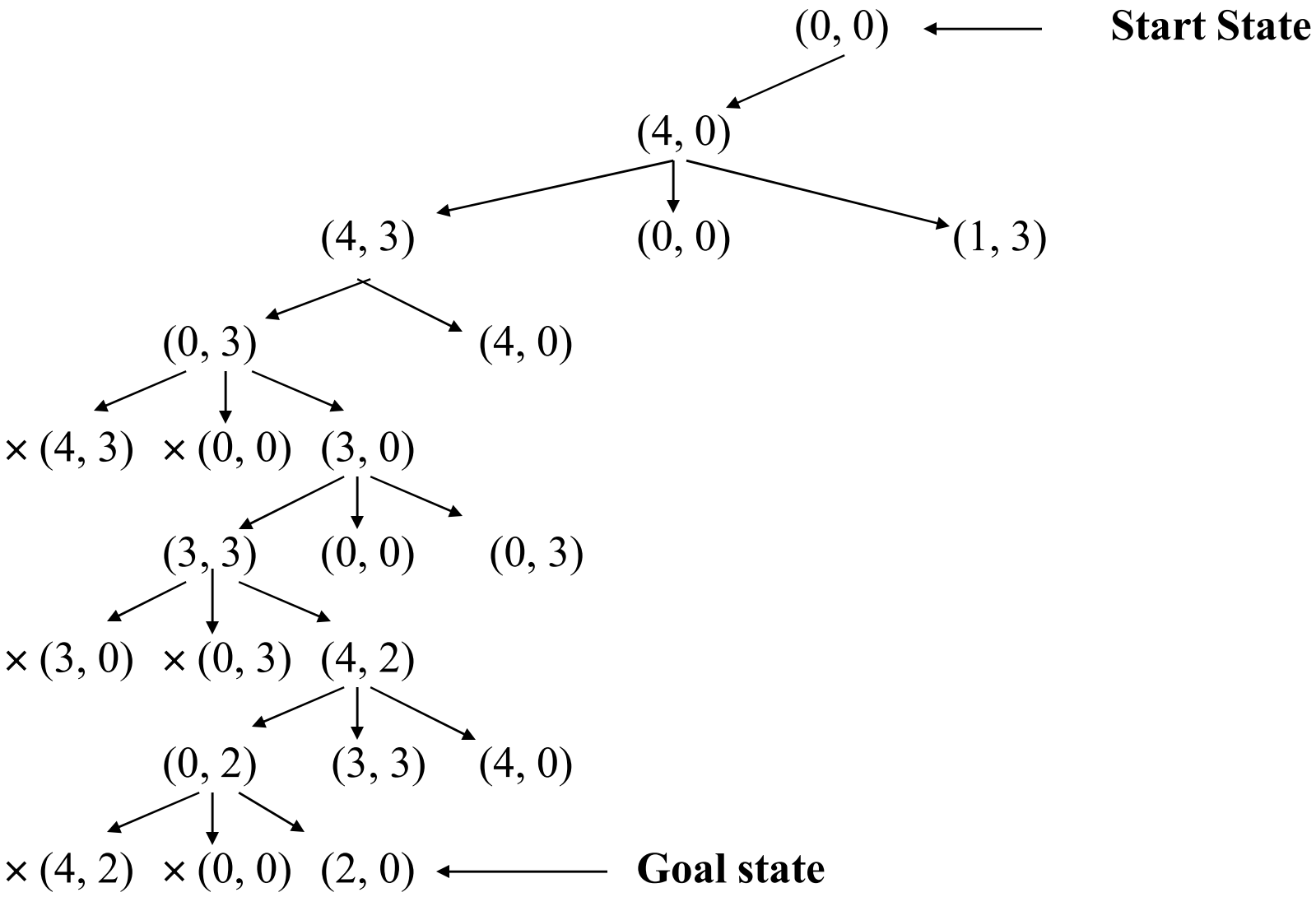
- BFS
  - Construct a tree with the initial state of the problem as its root.
  - Generate all the offspring of the root by applying each of the applicable rules to the initial state.
  - For each leaf node, generate all its successors by applying all the rules that are appropriate.
  - Repeat this process till we find a solution, if it exists.




(0, 0)
(4, 0)
(0, 3)
(4, 3)
(0, 0)
(1, 3)
(4, 3)
(0, 0)
(3, 0)

# Depth First Search - DFS

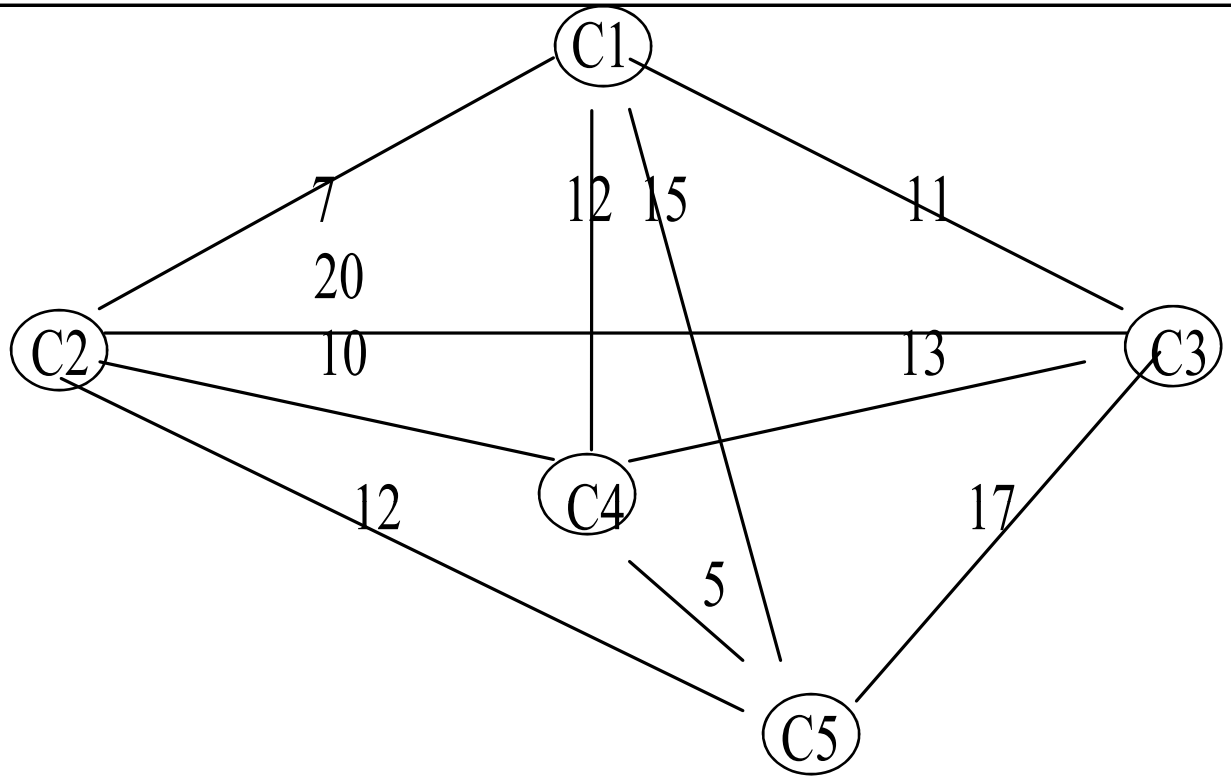
- Here we pursue a single branch of the tree until it yields a solution or some pre-specified depth has reached.
- If solution is not found then
  - go back to immediate previous node and
  - explore other branches in DF fashion.
- Let us see the tree formation for water jug problem using DFS


(0, 0)
Start State
(4, 0)
(4, 3)
(0, 0)
(1, 3)
(0, 3)
(4, 0)
× (4, 3)
× (0, 0)
(3, 0)
(3, 3)
(0, 0)
(0, 3)
× (3, 0)
× (0, 3)
(4, 2)
(0, 2)
(3, 3)
(4, 0)
× (4, 2)
× (0, 0)
(2, 0)
Goal state

# Traveling Salesman Problem

- Consider 5 cities.
  - A salesman is supposed to visit each of 5 cities.
  - All cities are pair wise connected by roads.
  - There is one start city.
  - The problem is to find the shortest route for the salesman who has to
    - visit each city only once and
    - returns to back to start city.

# Traveling Salesman Problem (Example) – Start city is C1




D(C1,C2) = 7; D(C1,C3) = 11; D(C1,C4) = 12; D(C1,C5) = 15; D(C2,C3) =20;

D(C2,C4) = 10; D(C2,C5) =12; D(C3,C4) =13; D(C3,C5) = 17; D(C4,C5) =5;

# Contd..

- A simple motion causing and systematic control structure could, in principle solve this problem.
- Explore the search tree of all possible paths and return the shortest path.
- This will require 4! paths to be examined.
- If number of cities grow, say 25 cities, then the time required to wait a salesman to get the information about the shortest path is of 0(24!) which is not a practical situation.

# Contd..

- This phenomenon is called **combinatorial explosion**.
- We can improve the above strategy as follows:
  - Begin generating complete paths, keeping track of the shortest path found so far.
  - Give up exploring any path as soon as its partial length becomes greater than the shortest path found so far.
  - This algorithm is efficient than the first one, still requires exponential time $\propto$ some number raised to N (number of cities).

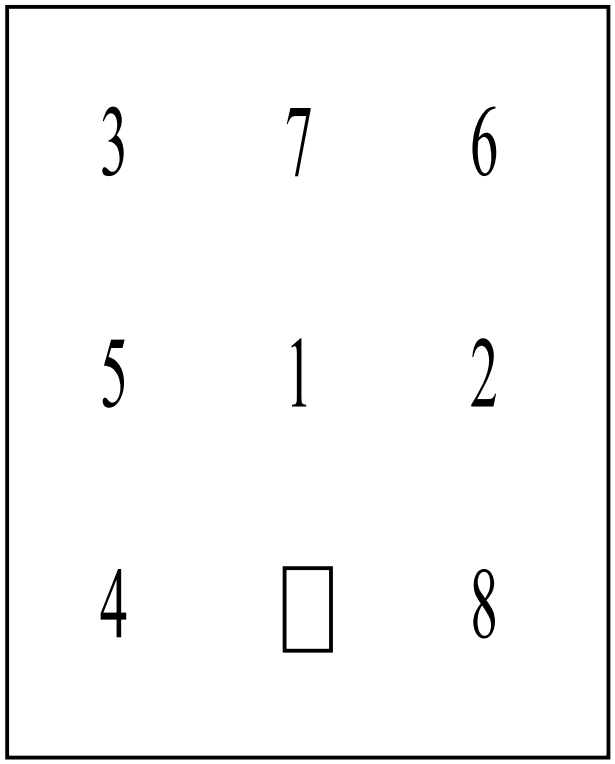
| Paths explored. Assume C1 to be the start city | | Distance |
|---|---|---|
| **1.** C1 → C2 → C3 → C4 → C5 → C1<br>7 20 13 5 15<br>27 40 45 60 | current best path | 60 √ × |
| **2.** C1 → C2 → C3 → C5 → C4 → C1<br>7 20 17 5 12<br>27 44 49 61 | | 61 × |
| **3.** C1 → C2 → C4 → C3 → C5 → C1<br>7 10 13 17 15<br>17 40 57 72 | | 72 × |
| **4.** C1 → C2 → C4 → C5 → C3 → C1<br>7 10 5 17 11<br>17 22 39 50 | current best path, cross path at S.No 1. | 50 √ × |
| **5.** C1 → C2 → C5 → C3 → C4 → C1<br>7 12 17 13 12<br>19 36 49 61 | | 61 × |
| **6.** C1 → C2 → C5 → C4 → C3 → C1<br>7 12 5 13 11<br>19 24 37 48 | current best path, cross path at S.No. 4. | 48 √ |
| **7.** C1 → C3 → C2 → C4 → C5<br>7 20 10 5<br>37 47 52 | (not to be expanded further) | 52 × |
| **8.** C1 → C3 → C2 → C5 → C4<br>11 20 12 5<br>37 49 54 | (not to be expanded further) | 54 × |
| **9.** C1 → C3 → C4 → C2 → C5 → C1<br>11 13 10 12 15<br>24 34 46 61 | | 61 × |
| **10.** C1 → C3 → C4 → C5 → C2 → C1<br>11 13 5 12 7<br>24 29 41 48 | same as current best path at S. No. 6. | 48 √ |
| **11.** C1 → C3 → C5 → C2<br>11 17 12<br>38 50 | (not to be expanded further) | 50 × |
| **12.** C1 → C3 → C5 → C4 → C2<br>11 17 5 10<br>38 43 53 | (not to be expanded further) | 53 × |
| **13.** C1 → C4 → C2 → C3 → C5<br>12 10 20 17<br>22 42 55 | (not to be expanded further) | 59 × |
| Continue like this | | |

# The 8-Puzzle

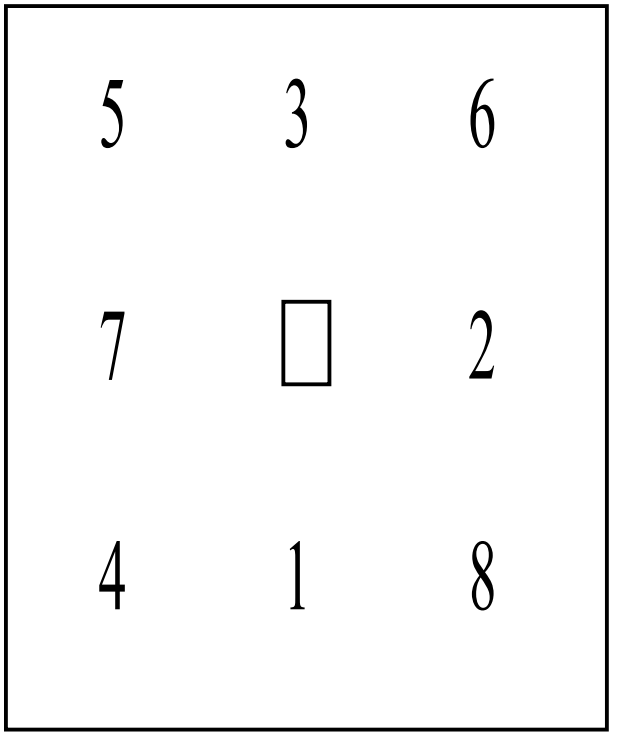
*Problem Statement*:

- The eight puzzle problem consists of a 3 x 3 grid with 8 consecutively numbered tiles arranged on it.
- Any tile adjacent to the space can be moved on it.
- Solving this problem involves arranging tiles in the goal state from the start state.

Start state

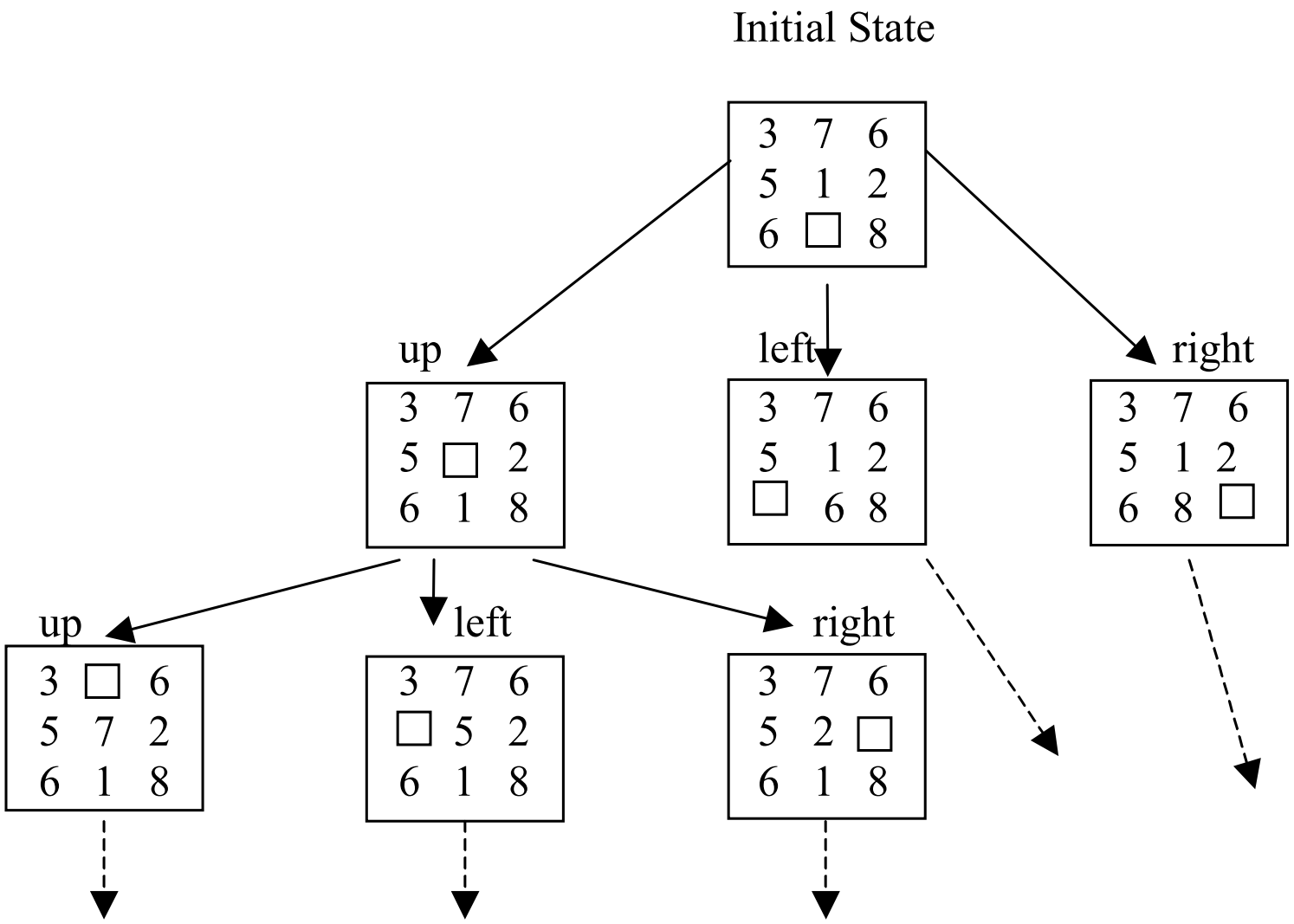



Goal state



# Solution by State Space method

- The start state could be represented as: [ [3,7,2], [5,1, 2], [4,0,6] ]
- The goal state could be represented as: [ [5,3,6] [7,0,2], [4,1,8] ]
- The operators can be thought of moving {up, down, left, right}, the direction in which blank space effectively moves.


Initial State
3 7 6
5 1 2
6 □ 8
up
3 7 6
5 □ 2
6 1 8
left
3 7 6
5 1 2
□ 6 8
right
3 7 6
5 1 2
6 8 □
up
3 □ 6
5 7 2
6 1 8
left
3 7 6
□ 5 2
6 1 8
right
3 7 6
5 2 □
6 1 8

# Searching for a Solution

- Problem can be solved by searching for a solution.
- Transform initial state of a problem into some final goal state.
- Problem can have more than one intermediate states between start and goal states.
- All possible states of the problem taken together are said to form
  - a **state space** or
  - problem state and
  - search is called **state space search**.

# Contd…

- Search is basically a procedure to discover a path through a problem space from initial state to a goal state.
- There are two directions in which such a search could proceed.
  - Data driven search, **forward**, from the start state
  - Goal driven search, **backward**, from the goal state

# Forward Reasoning (Chaining):

- It is a control strategy that starts with known facts and works towards a conclusion.
- For example in 8 puzzle problem, we start from initial state to goal state.
- In this case we begin building a tree of move sequences with initial state as the root of the tree.
- Generate the next level of the tree by finding all rules whose left sides match with root and use their right side to create the new state.
- Continue until a configuration that matches the goal state is generated.
- Language OPS5 uses forward reasoning rules. Rules are expressed in the form of “if-then rule”.
- Find out those sub-goals which could generate the given goal.

# Backward Reasoning (Chaining)

- It is a goal directed control strategy that begins with the final goal.
- Continue to work backward, generating more sub goals that must also be satisfied in order to satisfy main goal.
- Prolog (Programming in Logic) uses this strategy.

# General observations

- If there are large number of explicit goal states and one initial state,
  - then it would not be efficient to try to solve this in backward direction as we don't know which goal state is closest to the initial state. So it is better to reason forward.
- If problem has a single initial state and a single goal state, it makes no difference whether the problem is solved in the forward or the backward direction.
  - The computational effort is the same. In both these cases, same state space is searched but in different order.
- Move from the smaller set of states to the larger set of states.
- Proceed in the direction with the lower branching factor (the average number of nodes that can be reached directly from single node).

# Contd…

- In mathematics, suppose we have to prove a theorem.
  - There are initial states as small set of axioms.
  - From these set of axioms, we can prove large number of theorems.
- On the other hand, the large number of theorems must go back to the small set of axioms.
  - So branching factor is significantly greater going forward from axioms to theorem than going from theorems to axioms.

# General Purpose Search Strategies

- **Breadth First Search (BFS)**
  - It expands all the states one step away from the initial state, then expands all states two steps from initial state, then three steps etc., until a goal state is reached.
  - It expands all nodes at a given depth before expanding any nodes at a greater depth.
  - All nodes at the same level are searched before going to the next level down.
  - For implementation, two lists called OPEN and CLOSED are maintained.
    - The OPEN list contains those states that are to be expanded and CLOSED list keeps track of states already expanded.
    - Here OPEN list is used as a **queue**.

## Algorithm (BFS)

**Input**: Two states in the state space START and GOAL
**Local Variables**: OPEN, CLOSED, STATE-X, SUCCS
**Output:** Yes or No
**Method:**

- Initially OPEN list contains a START node and CLOSED list is empty; Found = false;
- While (OPEN ≠ empty and Found = false)
  Do {
  - Remove the first state from OPEN and call it STATE-X;
  - Put STATE-X in the front of CLOSED list;
  - If STATE-X = GOAL then **Found = true** else
    {- perform EXPAND operation on STATE-X, producing a list of SUCCESSORS;
    - Remove from successors those states, if any, that are in the CLOSED list;
    - Append SUCCESSORS at the end of the OPEN list/*queue*/
    } } /* end while */
- If Found = true then return **Yes** else return **No** and Stop

# Depth-First Search

- In depth-first search we go as far down as possible into the search tree / graph before backing up and trying alternatives.
  - It works by always generating a descendent of the most recently expanded node until some depth cut off is reached
  - then backtracks to next most recently expanded node and generates one of its descendants.
- So only path of nodes from the initial node to the current node is stored in order to execute the algorithm.
- For implementation, two lists called OPEN and CLOSED with the same conventions explained earlier are maintained.
  - Here OPEN list is used as a **stack.**
  - If we discover that first element of OPEN is the Goal state, then search terminates successfully else move it to closed list and stack its successor in open list.

## Algorithms (DFS)

**Input**: Two states in the state space, START and GOAL
**LOCAL Variables**: OPEN, CLOSED, RECORD-X, SUCCESSORS
**Output:** A path sequence if one exists, otherwise return No
**Method:**

- Form a stack consisting of (START, nil) and call it OPEN list.

Initially set CLOSED list as empty; Found = false;

- While (OPEN ≠ empty and Found = false) DO

{

• Remove the first state from OPEN and call it RECORD-X;
• Put RECORD-X in the front of CLOSED list;
• If the state variable of RECORD-X= GOAL,
then **Found = true**

Else

{ - Perform EXPAND operation on STATE-X, a state variable of RECORD-X, producing a list of action records called SUCCESSORS; create each action record by associating with each state its parent.

- Remove from SUCCESSORS any record whose state variables are in the record already in the CLOSED list.

- Insert SUCCESSORS in the front of the OPEN list /* Stack */

}

}/* end while */

- If Found = true then return the plan used /* find it by tracing through the pointers on the CLOSED list */ else return **No**
- Stop

# Comparisons

- DFS
  - is effective when there are few sub trees in the search tree that have only one connection point to the rest of the states.
  - can be dangerous when the path closer to the START and farther from the GOAL has been chosen.
  - Is best when the GOAL exists in the lower left portion of the search tree.
  - Is effective when the search tree has a low branching factor.
- BFS
  - can work even in trees that are infinitely deep.
  - requires a lot of memory as number of nodes in level of the tree increases exponentially.
  - is superior when the GOAL exists in the upper right portion of a search tree.

# Depth First Iterative Deepening (DFID)

- DFID is an iterative method that expands all nodes at a given depth before expanding any nodes at greater depth.
- For a given depth d, DFID performs a DFS and never searches deeper than depth d and d is increased by 1 in next iteration if solution is not found.
- Advantages:
  - It takes advantages of both the strategies (BFS & DFS) and suffers neither the drawbacks of BFS nor of DFS on trees
  - It is guaranteed to find a shortest - length (path) solution from initial state to goal state (same as BFS).
  - Since it is performing a DFS and never searches deeper than depth d. the space it uses is O(d) (same as DFS).
- Disadvantages:
  - DFID performs wasted computation prior to reaching the goal depth but time complexity remains same as that of BFS and DFS

# Algorithm (DFID)

- Initialize d = 1 /* depth of search tree */ , Found = false
- While (Found = false)

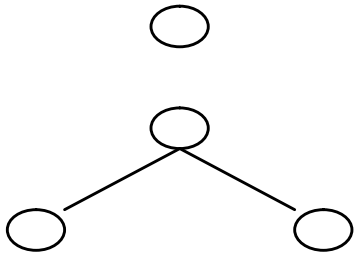
  {
  - perform a depth first search from start to depth d.
  - if goal state is obtained then **Found = true** else discard the nodes generated in the search of depth d
  - d = d + 1

  }/* end while */
- Report the solution
- Stop

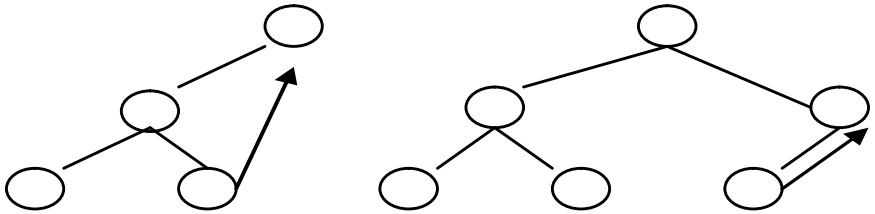
# Working of DFID

**Initial state**

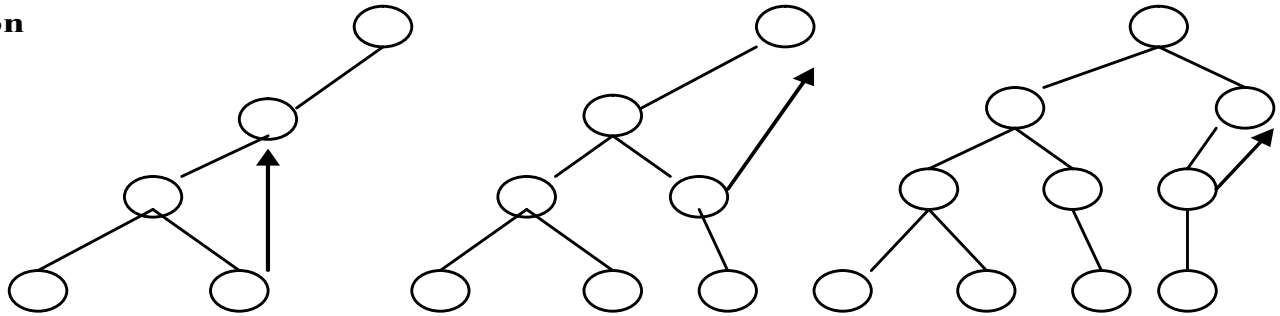
**Ist iteration**



**2nd iteration**



**3rd iteration**



**Continue this way**

# Analysis of Search methods

- Effectiveness of a search strategy in problem solving can be measured in terms of:
  - **Completeness:** Does it guarantees a solution when there is one?
  - **Time Complexity:** How long does it take to find a solution?
  - **Space Complexity:** How much space does it needs?
  - **Optimality:** Does it find the highest quality solution when there are several different solutions for the problem?

# Performance of BFS

- Time complexity
  - In worst case BFS must generate all nodes up to depth d

    $1 + b + b^2 + b^3 + \ldots + b^d = O(b^d)$

    - Note on average, half of the nodes at depth d must be examined.
  - So average case time complexity is $O(b^d)$
- Space complexity
  - Space required for storing nodes at depth d is $O(b^d)$

# Performance of DFS

- Time complexity
  - In worst case time complexity is $O(b^d)$
- Space complexity
  - If the depth cut off is d the space requirement is O(d)
- DFS requires an arbitrary cut off depth.
  - If branches are not cut off and duplicates are not checked for, the algorithm may not terminate.

# Performance of DFID

- Time complexity
  - nodes at depth d are generated once during the final iteration of the search
  - nodes at depth d-1 are generated twice
  - nodes at depth d-2 are generated thrice and so on.
  - Thus the total number of nodes generated in DFID to depth d are

$$T = b^d + 2b^{d-1} + 3b^{d-2} + \ldots. db$$
$$= b^d [1 + 2^{b-1} + 3^{b-2} + \ldots.db^{1-d}]$$
$$= b^d [1 + 2x + 3x^2 + 4x^3 + \ldots. + dx^{d-1}] \quad \{ \text{if } x = b^{-1} \}$$

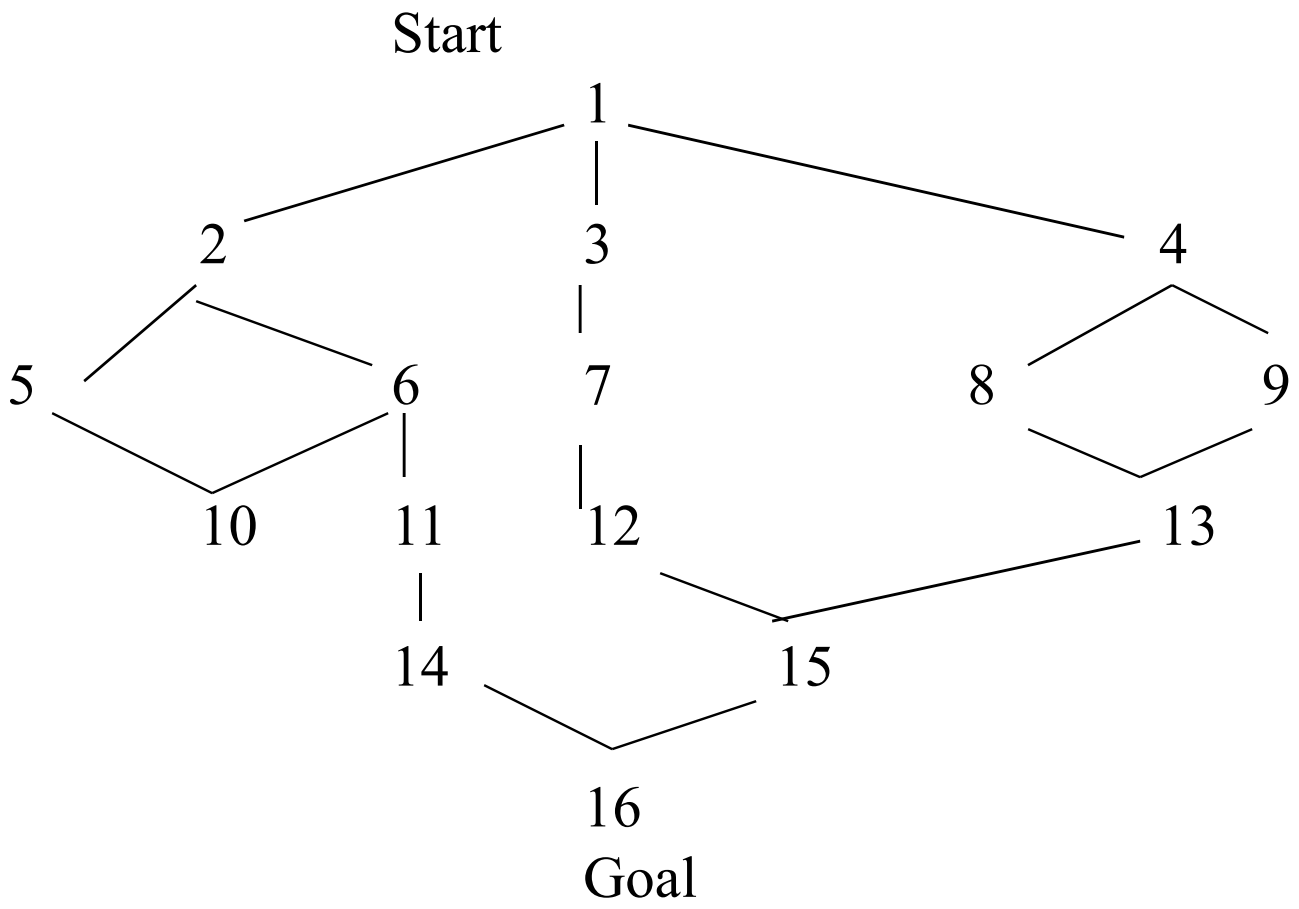
  - T converges to $b^d (1-x)^{-2}$ for $| x | < 1$. Since $(1-x)^{-2}$ is a constant and independent of d, if $b > 1$, $T \propto O(b^d)$ Space complexity
- Space complexity
  - Space required for storing nodes at depth d is O(d)

| | Time | Space | Optimality | Completeness |
|---|---|---|---|---|
| BFS | $O(b^d)$ | $O(b^d)$ | Yes | Yes |
| DFS | $O(b^d)$ | $O(d)$ | ---- | ---- |
| DFID | $O(b^d)$ | $O(d)$ | Yes | Yes |

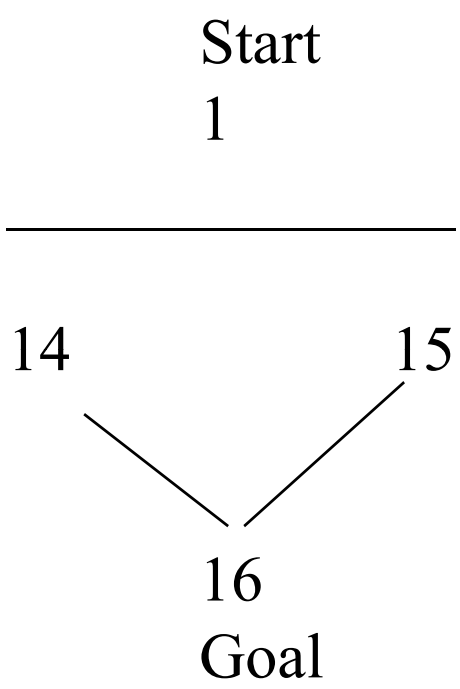
# Bi-Directional Search

- For those problems having a single goal state and single start state, bi-directional search can be used.
- It starts searching forward from initial state and backward from the goal state simultaneously starting the states generated until a common state is found on both search frontiers.
- DFID can be applied to bi-directional search for k = 1, 2, .. as follows :
  - kth iteration consists of a DFS from one direction to depth k storing all states at depth k, and
  - DFS from other direction : one to depth k and other to depth k+1 not storing states but simply matching against the stored states from forward direction.
    - The search to depth k+1 is necessary to find odd-length solutions.

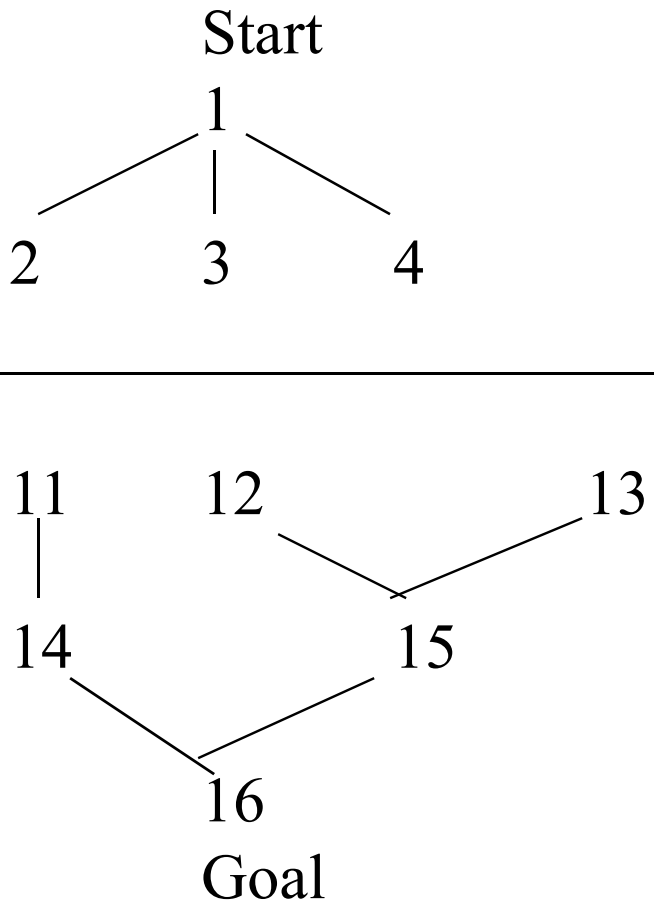
**Graph:**



For k = 0



For k = 1



For k = 2