Problem Solving: State-Space Search and Control Strategies

Chapter 2(b)

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.

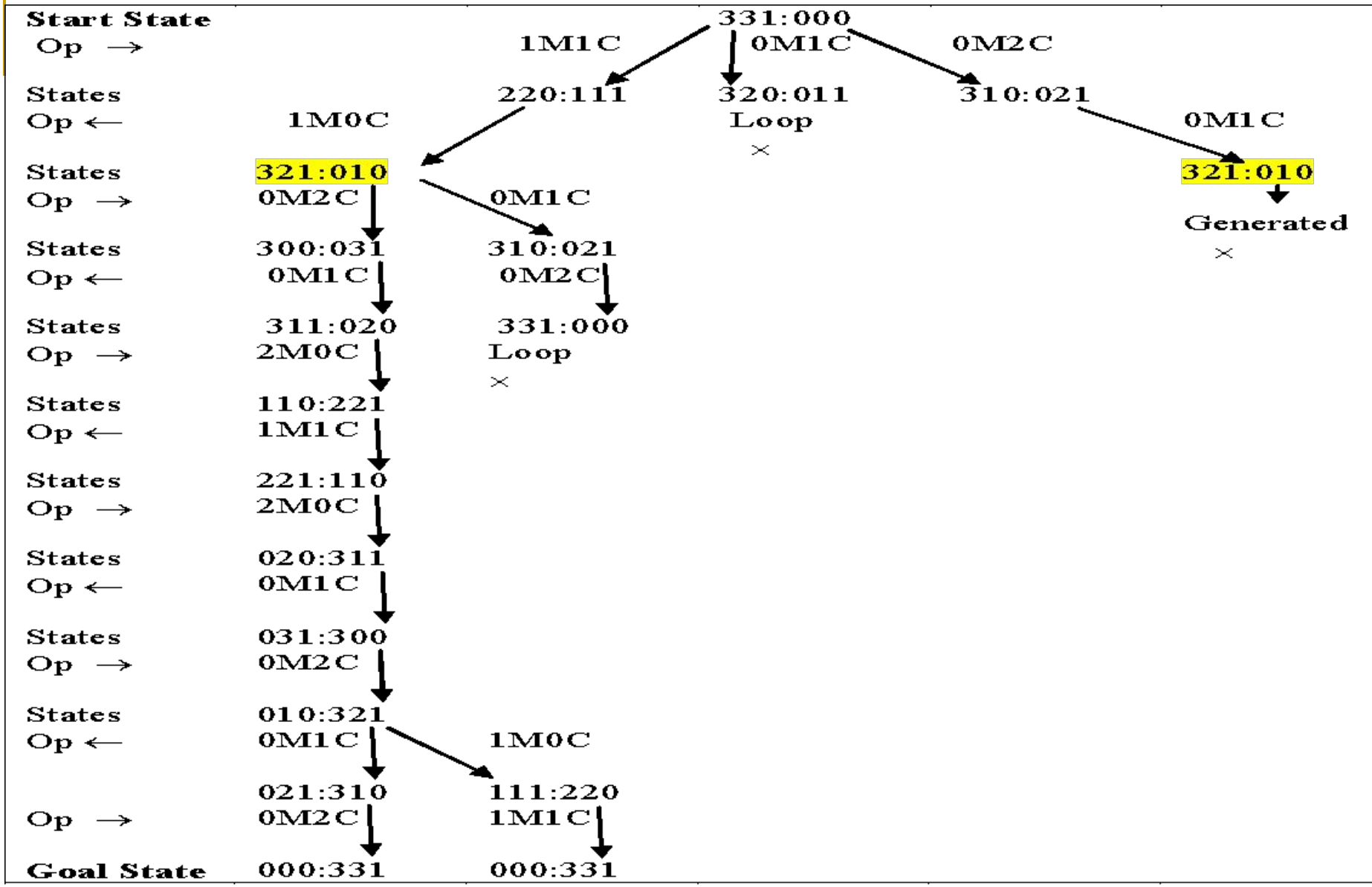
Missionaries and Cannibals

- 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 →" and if the boat is on right bank of the river then we write "Operator ←".

- 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

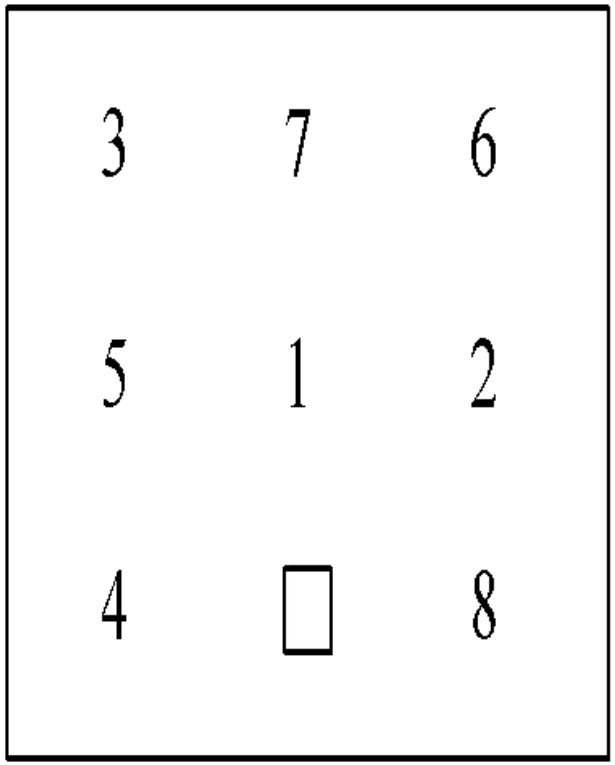
Two Possible solution paths	
Solution Path 1	*Solution Path 2*
1M1C →	1M1C →
1M0C ←	1M0C ←
0M2C →	0M2C →
0M1C ←	0M1C ←
2M0C →	2M0C →
1M1C ←	1M1C ←
2M0C →	2M0C →
0M1C ←	0M1C ←
0M2C →	0M2C →
0M1C ←	1M0C ←
0M2C →	1M1C →

The 8-Puzzle

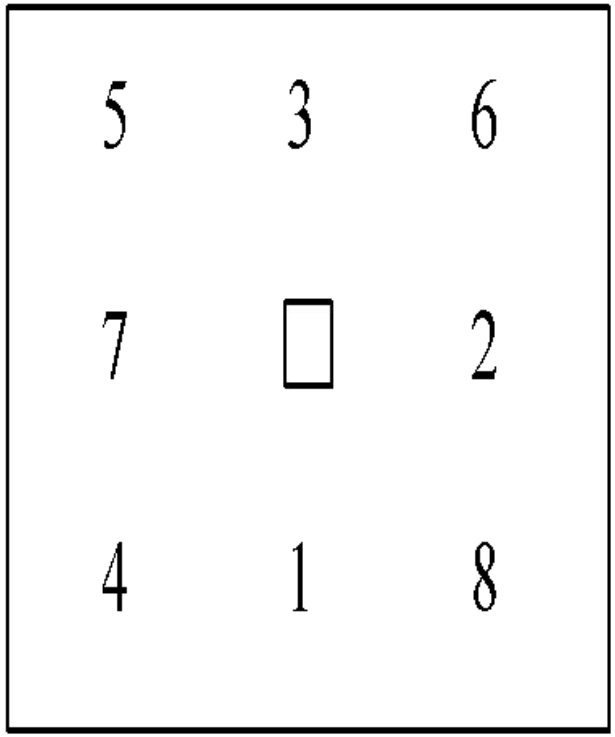
Problem Statement:

- 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

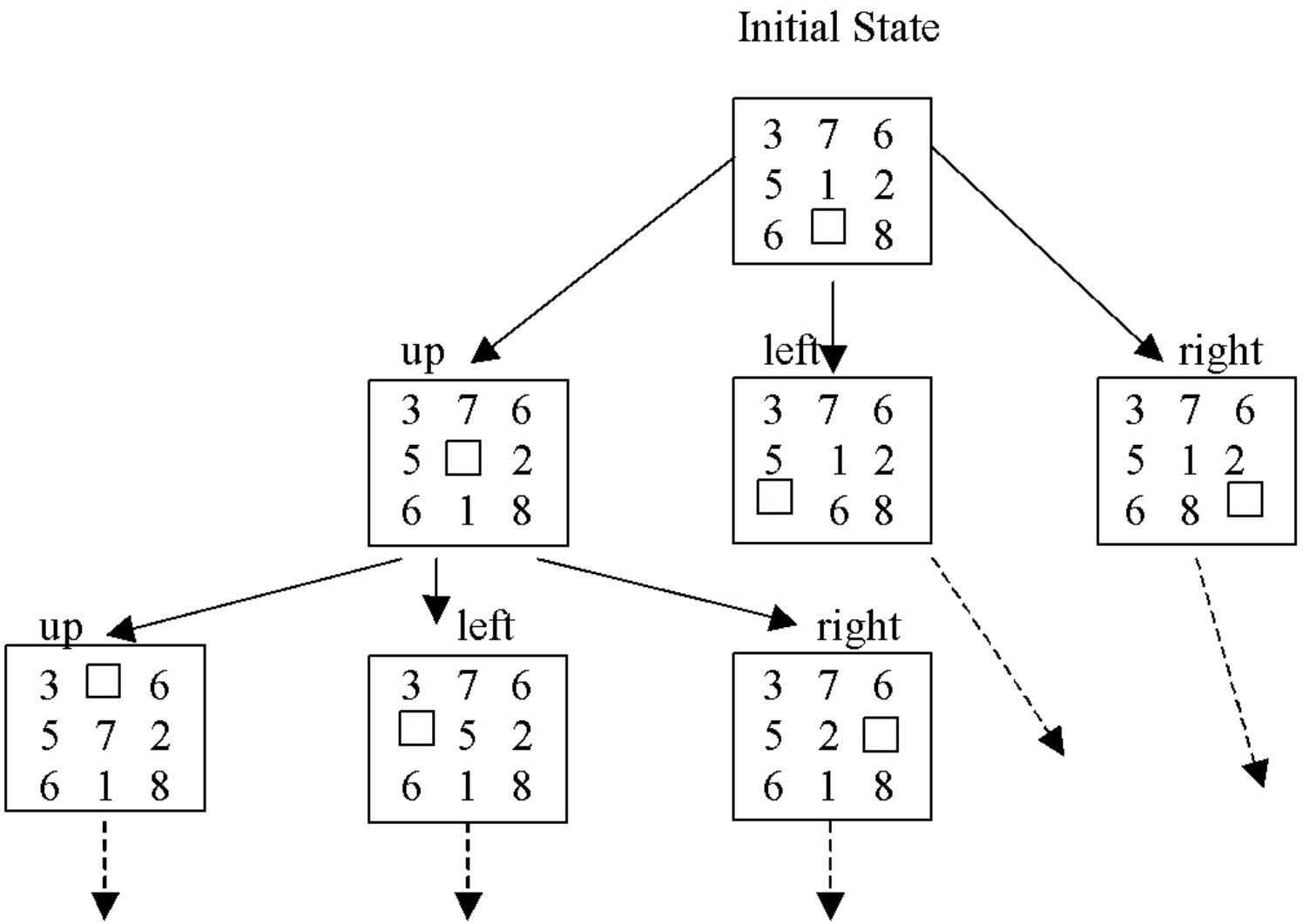


Goal state



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 SUCCS;
    - Remove from successors those states, if any, that are in the CLOSED list;
    - Append SUCCS 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, SUCCS

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 SUCCS; create each action record by associating with each state its parent.

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

- Insert SUCCS 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)

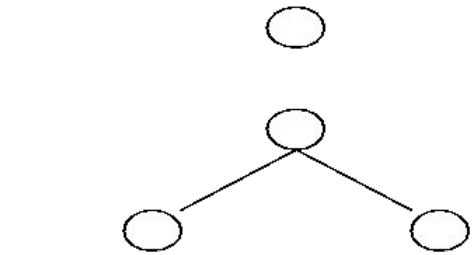
  {
  - perform a depth first search from start to depth d.
  - if goal state is obtained then **Found = true** else discard the nodes generated in the search of depth d
  - d = d + 1

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

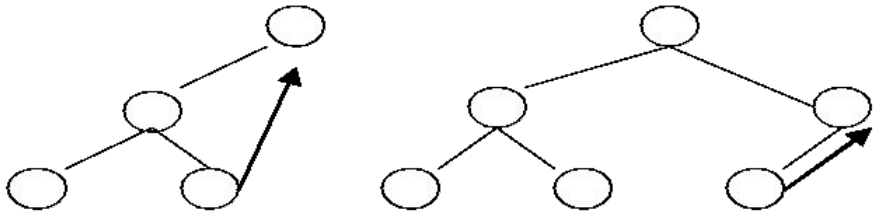
Working of DFID

Initial state

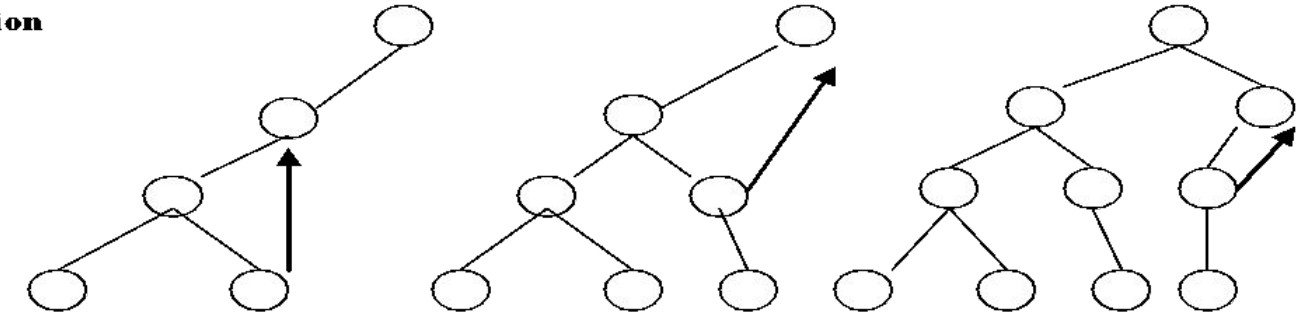
Ist iteration



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 with branching factor b.

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

    - 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

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

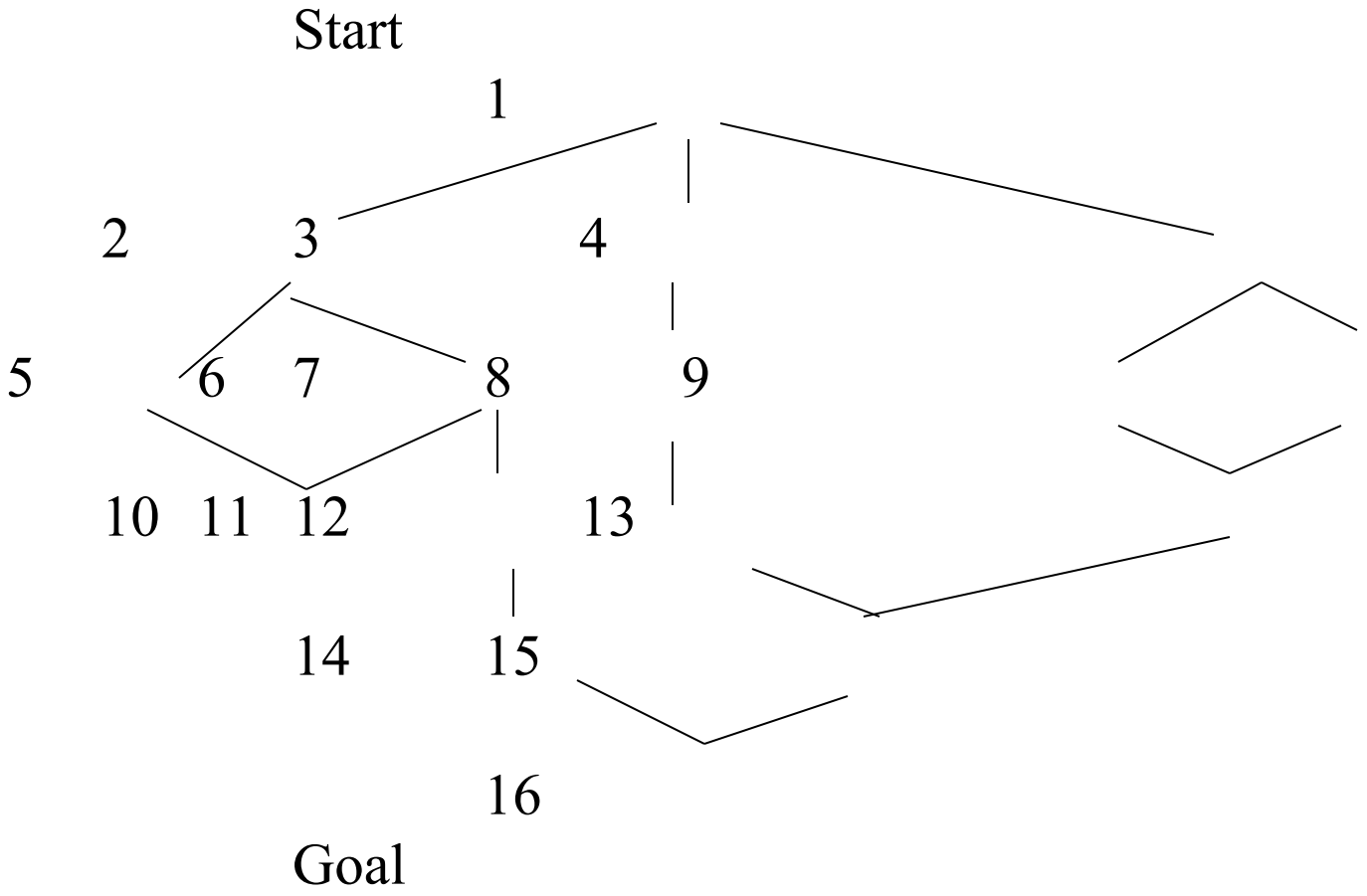
  - T converges to $b^d (1-x)^{-2}$ for $|x| < 1$. Since $(1-x)^{-2}$ is a constant and independent of d, if $b > 1$, $T \propto O(b^d)$ Space complexity
- Space complexity
  - Space required for storing nodes at depth d is O(d)

	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

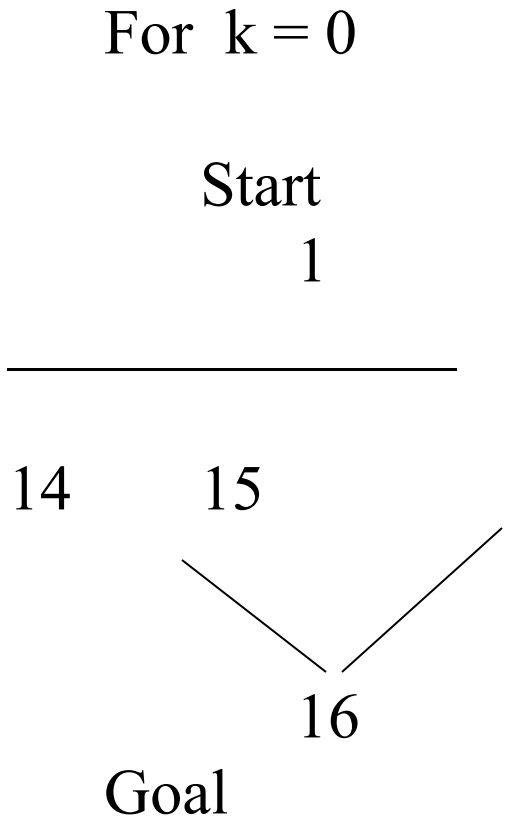
Bi-Directional Search

- 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.

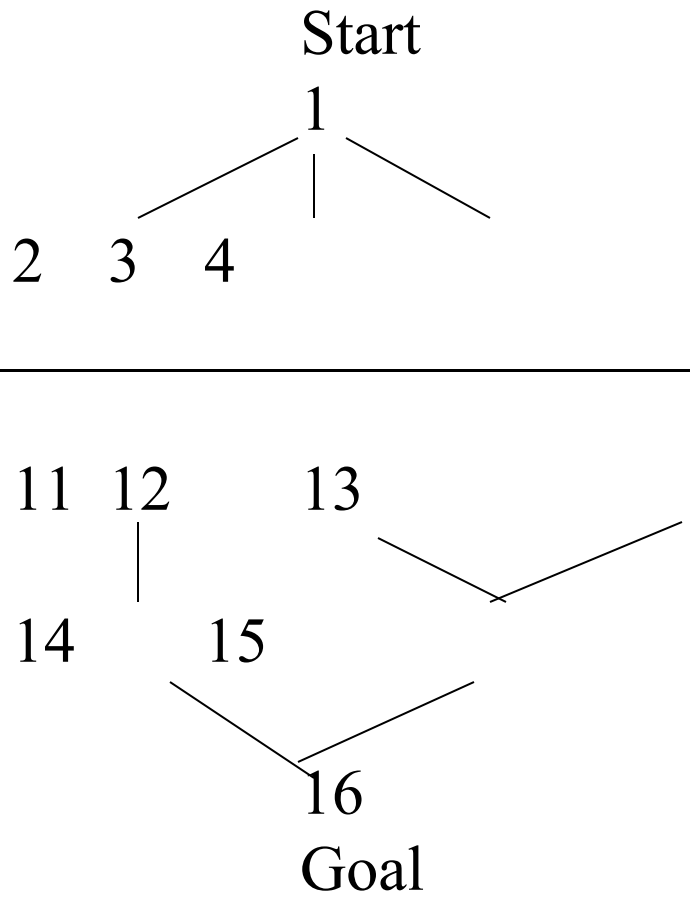
Graph:



For k = 0



For k = 1



For k = 2