Problem Solving in AI

The Eight Puzzle Problem

- The Eight Puzzle Problem has 3x3 grid with 8 randomly numbered tiles arrangement with one empty cell. At any time an adjacent cell can move to empty cell creating a new empty cell. Solving the puzzle involves arranging the cells in such a way so as to achieve the goal state.

START STATE

3	7	6
5	1	2
4		8

GOAL STATE

5	3	6
7		2
4	1	8

Eight Puzzle Problem Representation

- Start State: {[3,7,6], [5,1,2], [4,0,8]}
- Goal State: {[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.

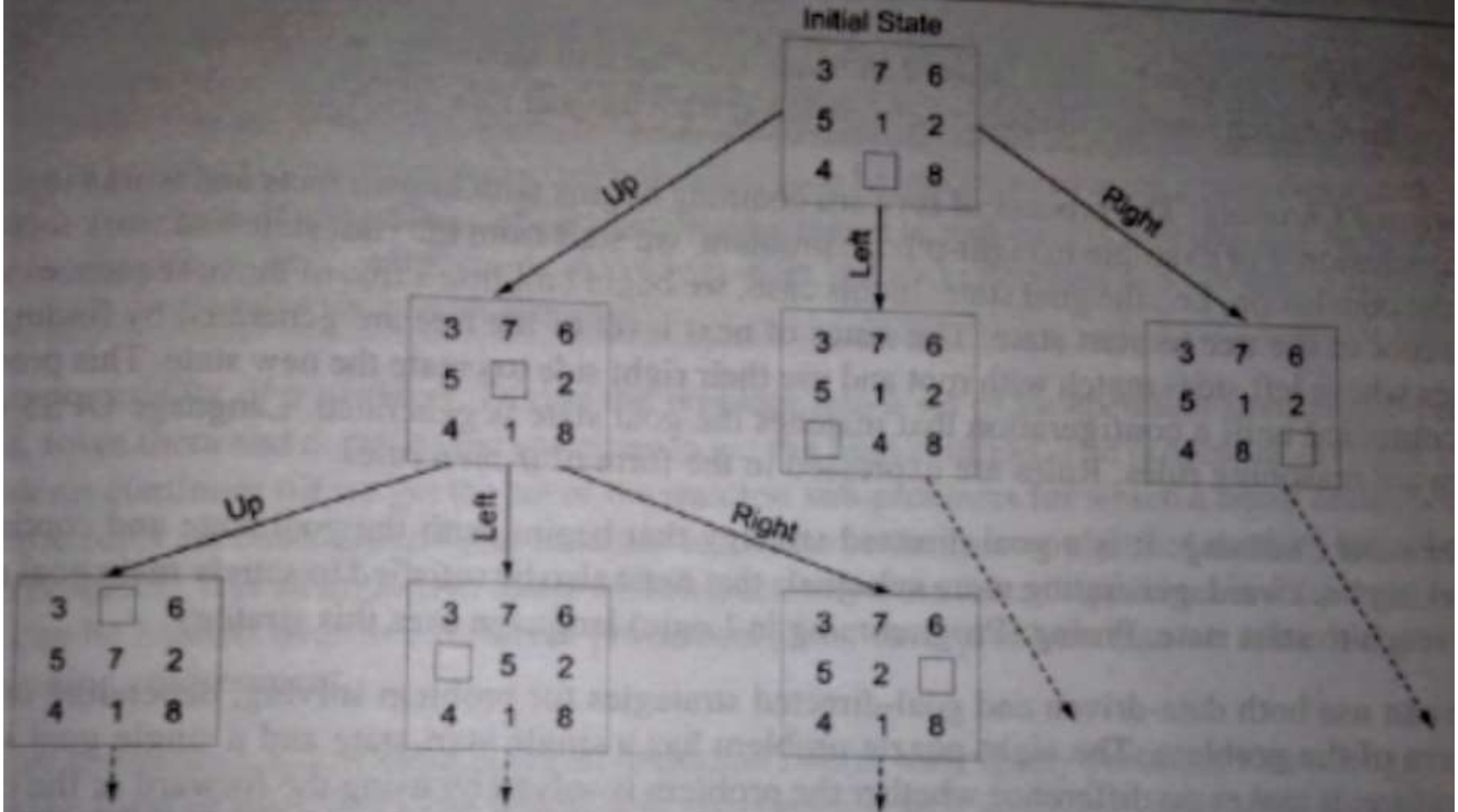

Figure 2.3 Partial Search Tree for Eight Puzzle Problem

Control Strategy

- Control Strategy describes the order of application of rules.
- Should cause motion towards a solution
- There are two directions in which search could proceed:
  - Data Driven: Forward chaining
  - Goal Driven: Backward chaining

Types Of Problem

- Ignorable
- Recoverable
- Irrecoverable

Characteristics of a Problem

- Decomposability
- Role of Knowledge
- Requirement of Solution
  - Absolute
  - Relative

Exhaustive Searches

- If we select a control strategy where we select a rule randomly from the applicable rules.
- Definitely it causes motion and eventually leads to a solution.
- Possibility that we may arrive to same state several times.
- Because control strategy is not systematic.
- Searches are systematic control strategies eg BFS and DFS (blind searches)

Uninformed and Informed Search Techniques

- Uninformed:
  - Uninformed search algorithms have no additional information on the goal node other than the one provided in the problem definition.
  - The plans to reach the goal state from the start state differ only by the order and length of actions.
  - Eg DFS and BFS
- Informed:
  - Informed Search algorithms have information on the goal state which helps in more efficient searching.
  - This information is obtained by a function that estimates how close a state is to the goal state
  - Eg Greedy search

Breadth First Search

- Expands all states at first level of tree/graph ie one step away from start state
- Then expands second level ie two step away from start state
- And so on until a goal state is reached.
- Always give optimal path or solution

Algorithm

Algorithm *(BFS)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, STATE-X, SUCCs, FOUND;
Output: Yes or No
Method:

- initialize OPEN list with START and CLOSED = ϕ;
- FOUND = false;
- while (OPEN ≠ ϕ and FOUND = false) do
  {
  - remove the first state from OPEN and call it STATE-X;
  - put STATE-X in the front of CLOSED list {maintained as stack};
  - 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***
- Stop

Water Jug Problem

Open list(5,3)

(2,3) (3,0)

Closed List (0,0)
(5,0) (5,3) (2,3)
(0,3)

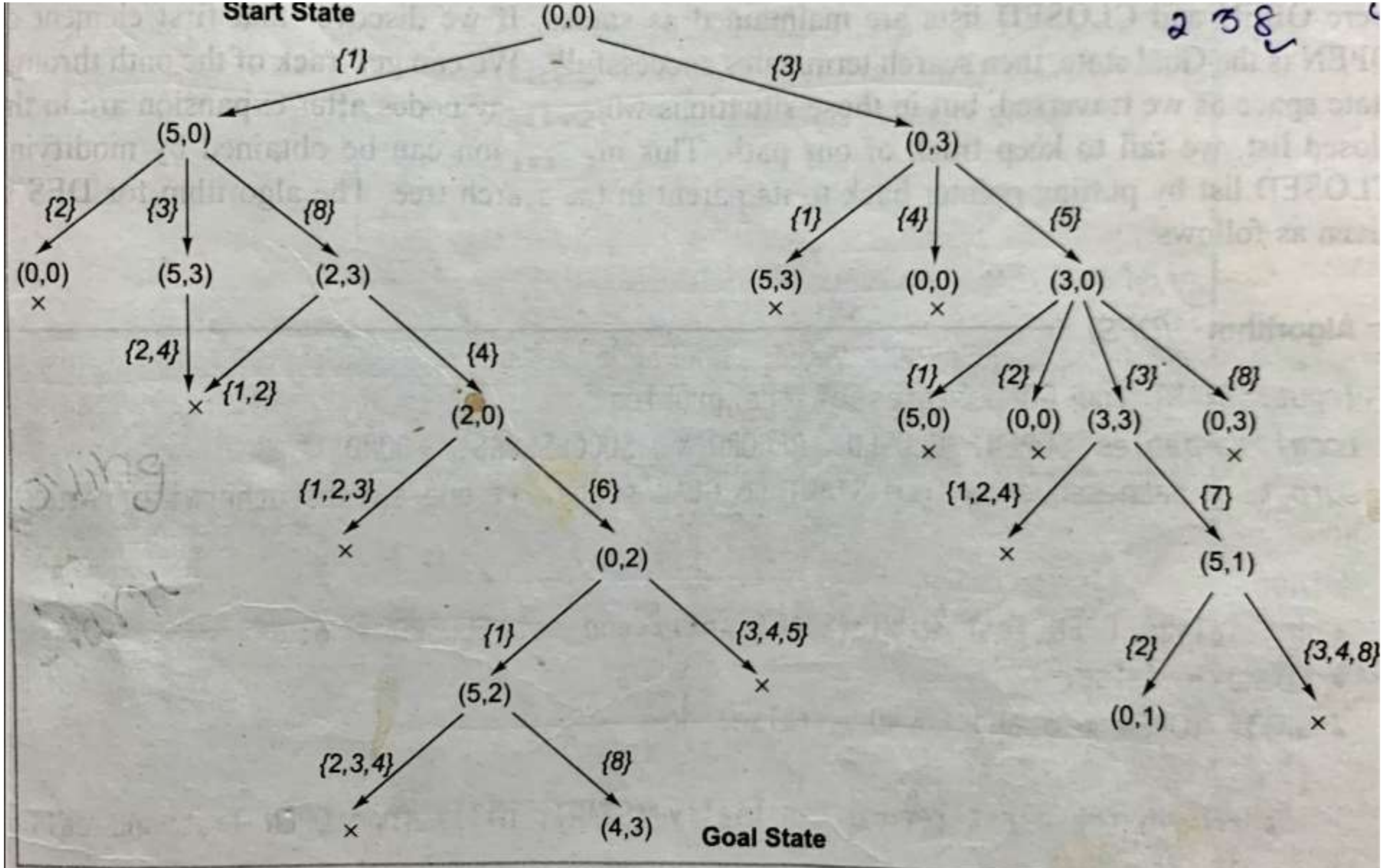

Figure 2.4 Search Tree Generation using BFS

Solution Path

(0,0)->(5,0)->(2,3)->(2,0)->(0,2)->(5,2)->(4,3)

Depth First Search

- Go as far down as possible into the search tree/graph before backing up and trying alternatives.
- Works by generating a descendent of the most recently expanded node until some cut off is reached and then back tracks to next most recently expanded node and generate one of its descendants.
- DFS is memory efficient

Algorithm

Algorithm *(DFS)*

Input: START and GOAL states of the problem
Local Variables: OPEN, CLOSED, RECORD_X, SUCCESSORS, FOUND
Output: A path sequence from START to GOAL state, if one exists otherwise return No
Method:

- initialize OPEN list with (START, nil) and set CLOSED = ϕ;
- FOUND = false;
- while (OPEN $\neq$ ϕ and FOUND = false) do

  {
  - remove the first record (initially (START, nil)) from OPEN list and call it RECORD-X;
  - put RECORD-X in the front of CLOSED list (maintained as stack);
  - if (STATE_X of RECORD_X= GOAL) then FOUND = true else

    {
    - perform EXPAND operation on STATE-X producing a list of records called SUCCESSORS; create each record by associating parent link with its state;
    - remove from SUCCESSORS any record that is already in the CLOSED list;
    - insert SUCCESSORS in the front of the OPEN list /* Stack */

    }

  }/* end while */
- if FOUND = true then return the path by tracing through the pointers to the parents on the CLOSED list else return ***No***
- Stop

Water Jug Problem

Water Jug Problem

Search tree generation using DFS	OPEN list	CLOSED list
Start State (0, 0) ↓ {1}	[((0,0), nil)]	
(5, 0) {2} × {3}	[((5,0), (0,0))]	[((0,0), nil)]
(5, 3) ↓ {2}	[((5,3), (5,0))]	[((5,0), (0,0)), ((0,0), nil)]
(0, 3) {1,4} × {5}	[((0,3), (5,3))]	[((5,3), (5,0)), ((5,0), (0,0)), ((0, 0), nil)]
(3, 0) ↓ {3}	[((3,0), (0,3))]	[((0,3), (5,3)), ((5,3), (5,0)), ((5,0), (0,0)), ((0,0), nil)]
(3, 3) {1,2,4} × {7}	[((3,3), (3,0))]	[((3,3), (3,0)), ((0,3), (5,3)), ((5,3), (5,0)), ((5,0), (0,0)), ((0,0), nil)]
(5, 1) ↓ {2}	⋮	⋮
(0, 1) {1,2,4} ×		
(1, 0) {1,2} × {3}		
(1, 3) {1,2,4} × {5}		
(4, 0) Goal state	[(4,0), (1,3))]	[((4,0),(1,3)), ((1,3), (1,0)), ((1,0), (0,1)), ((0,1), (5,1)), ((5,1), (3,3)), ((3,3), (3,0)),((3,0), (0,3)),((0,3), (5,3)), ((5,3), (5,0)), ((5,0), (0,0)), ((0,0), nil)]

Figure 2.5 Search Tree Generation using DFS

Solution Path

(0,0)->(5,0)->(5,3)->(0,3)->(3,0)->(3,3)->(5,1)->(0,1)->(1,0)->(1,3)->(4,0)

Comparing BFS and DFS

BFS	DFS
Effective when search has low branching factor.	Effective when there are few sub trees in the search tree.
Efficient when tree is infinitely deep.	
Requires lot of memory.	Memory efficient.
Superior when goal exists in upper right portion of search tree.	Best when goal exists in lower left part of search tree.
BFS gives optimal solution.	May not give optimal solution.

Depth First Iterative Deepening

- Takes advantage of both BFS and DFS.
- Expands all nodes at a given depth before expanding any nodes at a greater depth.
- Guarantees to find the optimal path from start state to goal state.

Algorithm *(DFID)*

```
Input: START and GOAL states
Local Variables: FOUND:
Output: Yes or No
Method:
• initialize d = 1 /* depth of search tree */ . FOUND = false
• while (FOUND = false) do
  {
   • 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 */
• if FOUND = true then return Yes otherwise return No
• Stop
```

Example

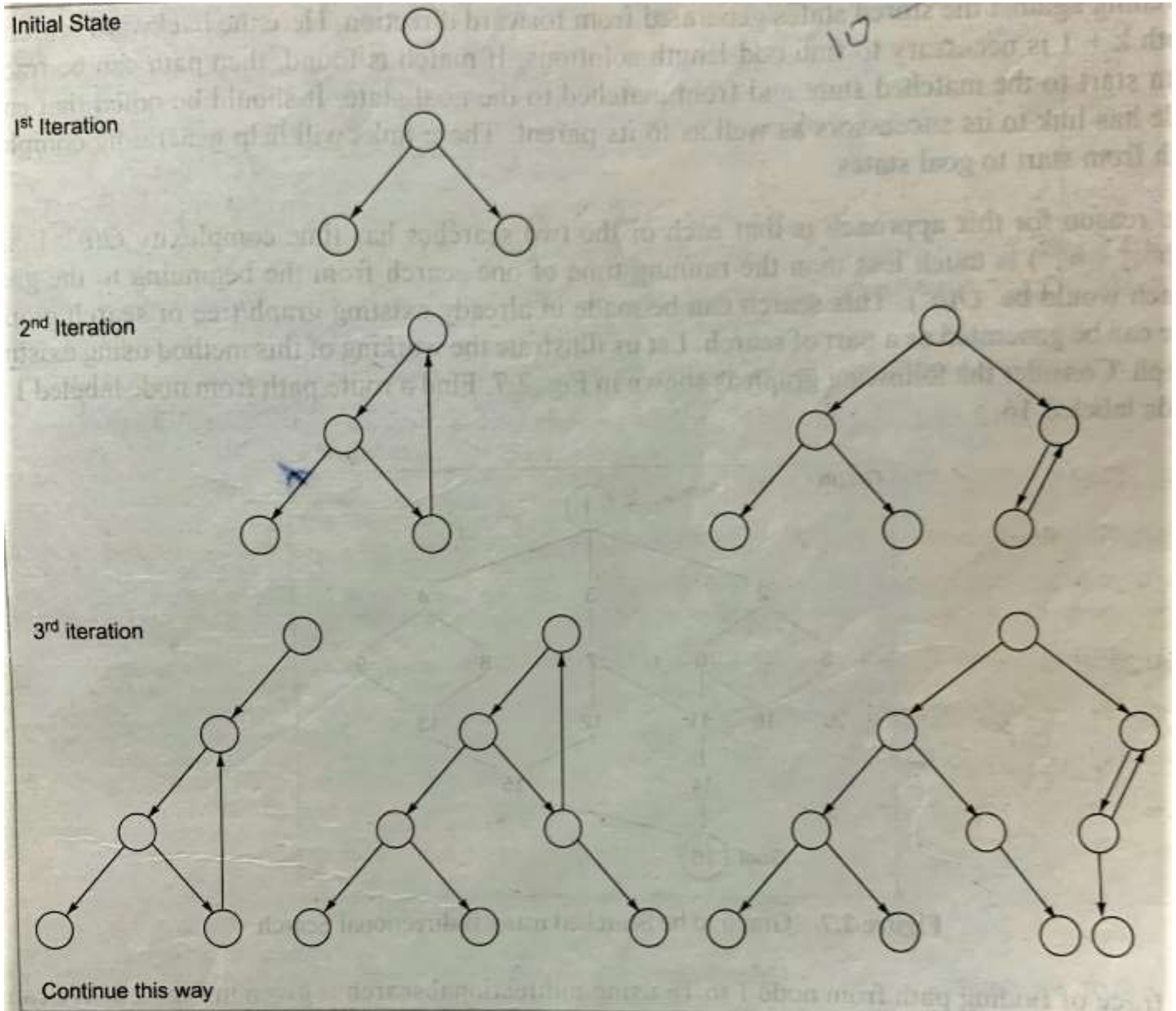

Figure 2.6 Search Tree Generation using DFID

Complexity

- $O(b^d)$
- Space used is $O(d)$

Bidirectional Search

- Runs two simultaneous searches.
  - One moves in forward direction start to goal state.
  - Other moves in backward direction goal to start state.
- Stops when two meet in the middle.
- Useful for problems that have single start and single goal state.

Example Graph

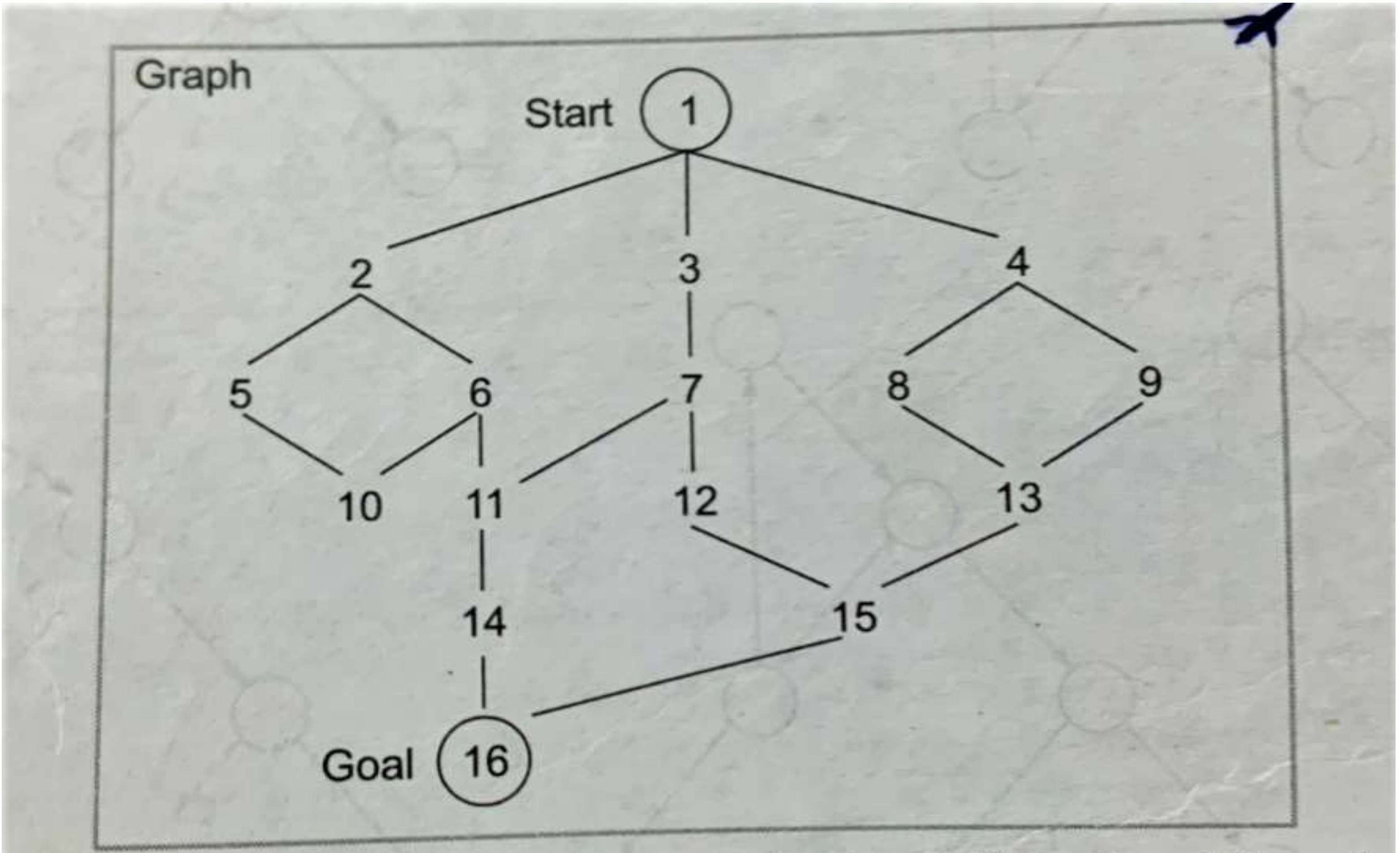

Figure 2.7 Graph to be Searched using Bidirectional Search

Trace of Bi-directional Search

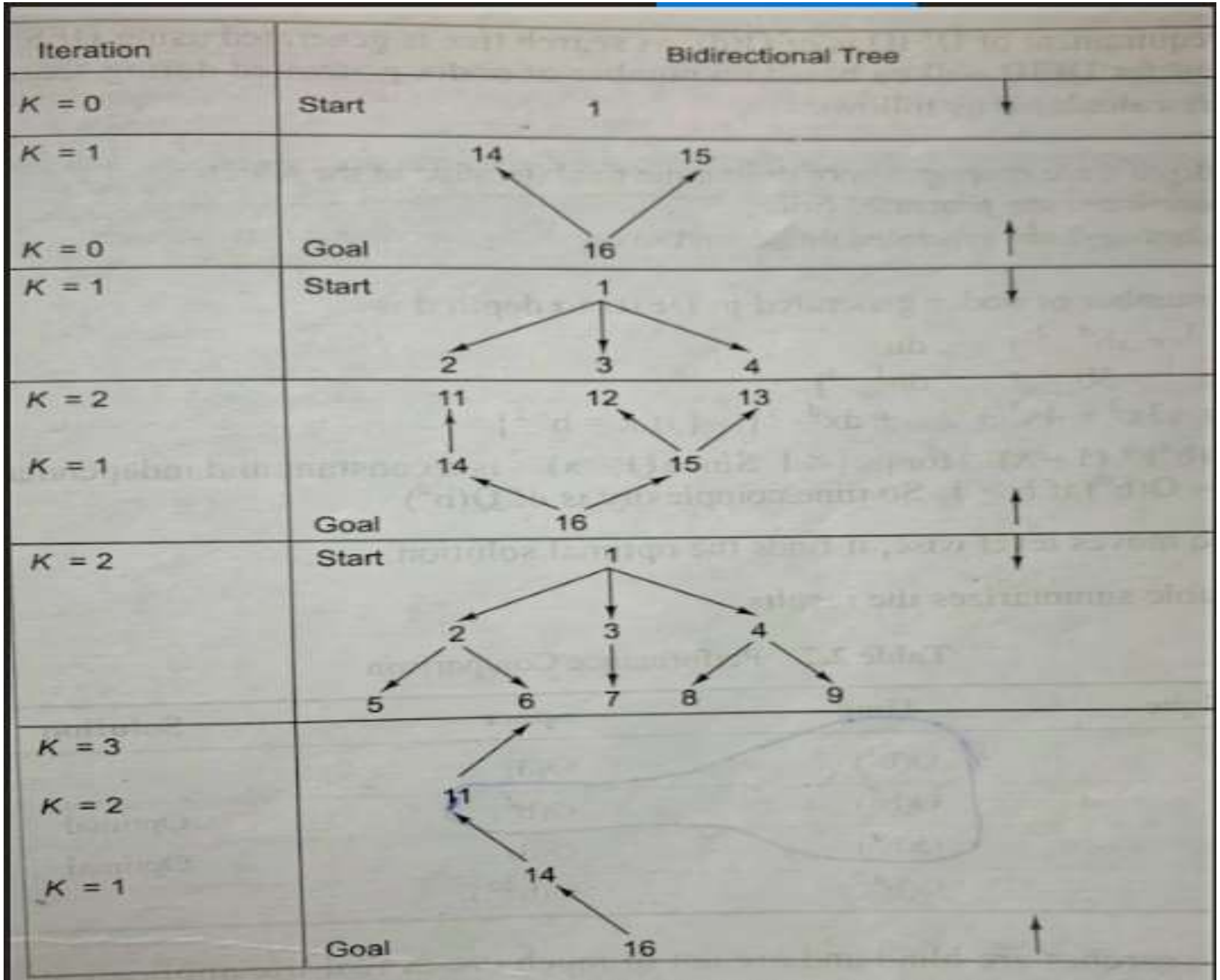

Figure 2.8 Trace of Bidirectional Space

Complexity

- O(b^d/2 + b^d/2)

Performance Comparison of all Search Techniques

Search Technique	Time	Space	Solution
DFS	O(b^d)	O(d)	-
BFS	O(b^d)	O(b^d)	Optimal
DFID	O(b^d)	O(d)	Optimal
Bi-directional	O(b^d/2)	O(b^d/2)	-

Any Queries?