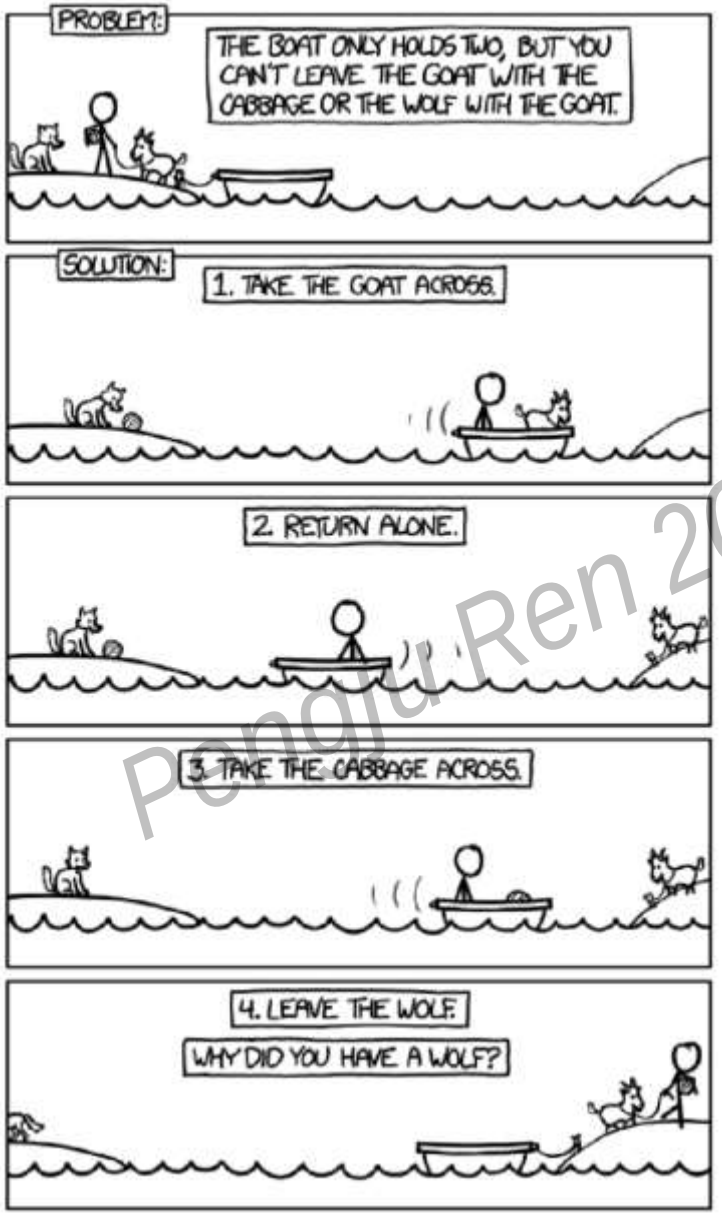
Data Structures and Algorithms

Lecture 08 – Farmer Wolf Goat and Cabbage (Searching and Backtracking)

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Problem Solving



Farmer, wolf, goat, and cabbage river crossing problem

- The *wolf* and the *goat* cannot be left alone together
- The *goat* and the *cabbage* cannot be left alone together
- The *farmer* can carry only one item (or nothing) at a time when crossing the river, and must row the boat himself

Farmer Wolf Goat and Cabbage

State: A state can be represented by a 4-tuple *(F, W, G, C)*, where each component takes the value 0 (*left*) or 1 (*right*).

Safety Constraints:

- If F ≠ W and W == G, then illegal (wolf and goat together on the bank without the farmer)
- If F ≠ G and G == C, then illegal (goat and cabbage together on the bank without the farmer)

Initial state: All are on the left: *(0, 0, 0, 0)*

Goal state: All are on the right: *(1,1,1,1)*

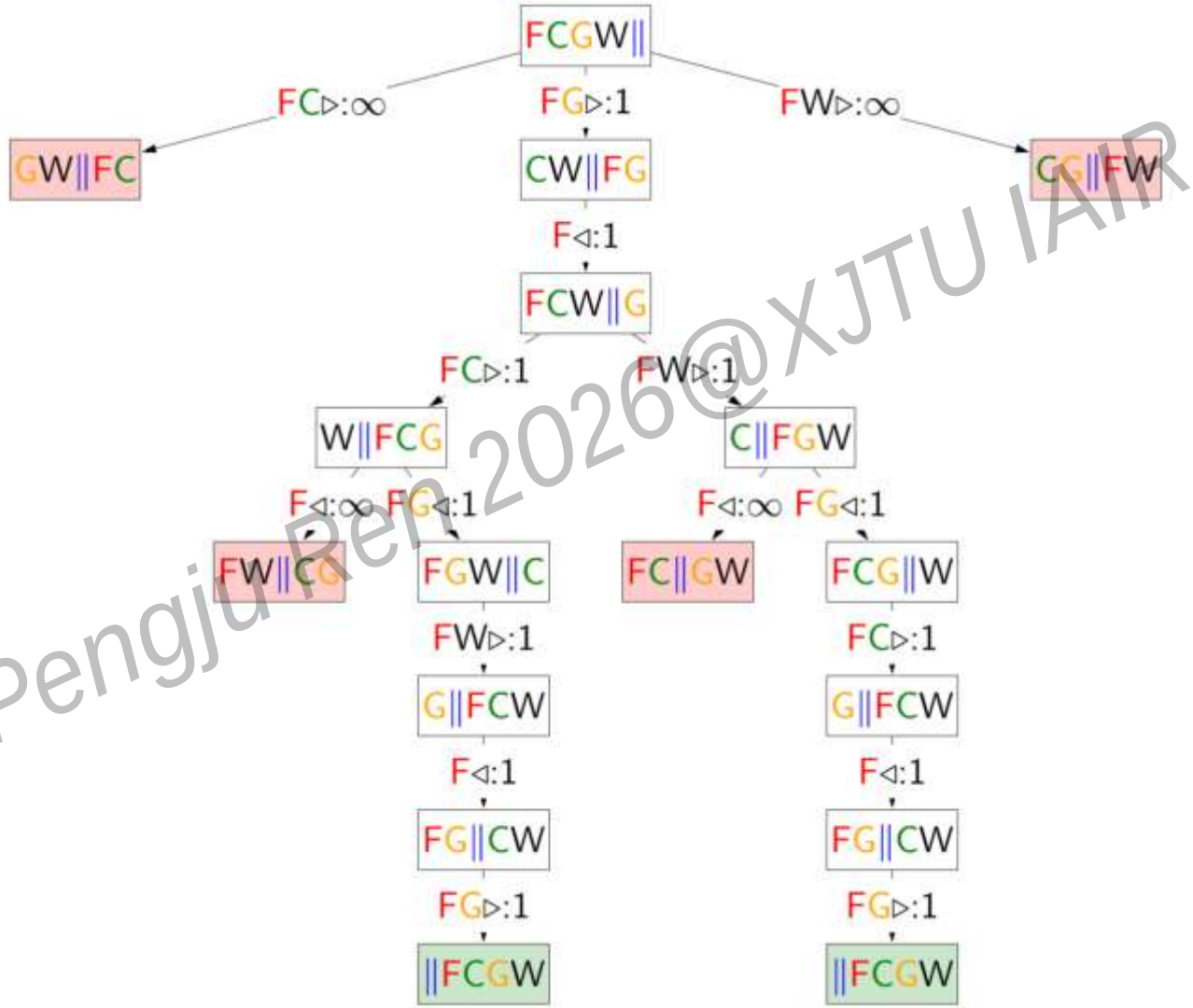
Transition rule: the farmer changes his bank (from 0 to 1 or 1 to 0), and optionally takes at most one item that is on the same bank as him. The new state must satisfy the safety constraints.

Farmer Wolf Goat and Cabbage

State space:

- Total number of states (without safety constraints): $2^4 = 16$.
- Number of legal states: 10.
- A state is illegal if there exists a bank where (W == G) or (G == C) and the farmer is not on that bank.
  Formally:
  - (F == 0 and W == G and W == 0) or (F == 0 and G == C and G == 0)
  - (F == 1 and W == G and W == 1) or (F == 1 and G == C and G == 1)
- Reachable state space (from initial): 10.

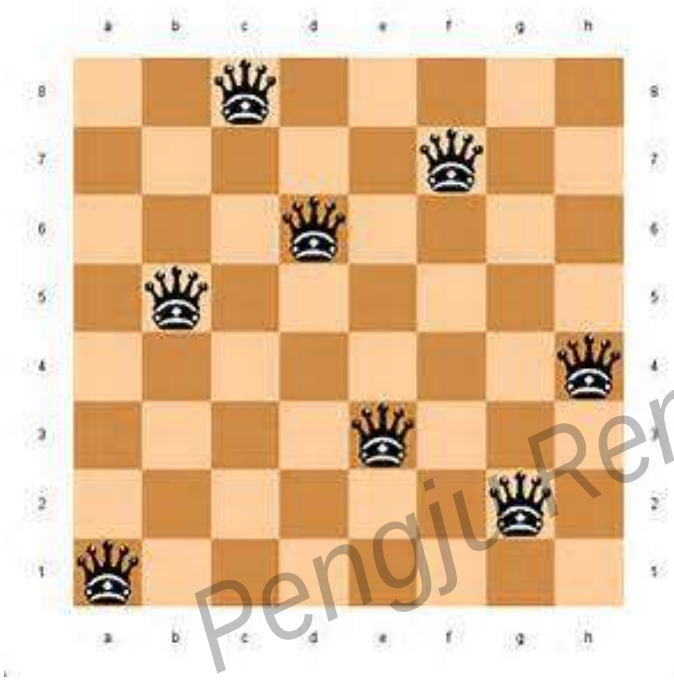
Problem Solving



Solution of Farmer Problem

```
def farmer_dfs():
    start = (0,0,0,0)  # f,w,g,c
    goal = (1,1,1,1)
    def is_safe(state):
        f,w,g,c = state
        if g == w and f != g: return False
        if g == c and f != g: return False
        return True
    def get_moves(state):
        f,w,g,c = state
        nf = 1-f
        moves = [(nf, w, g, c)]
        if w == f: moves.append((nf, 1-w, g, c))
        if g == f: moves.append((nf, w, 1-g, c))
        if c == f: moves.append((nf, w, g, 1-c))
        return [m for m in moves if is_safe(m)]
    visited = set()
    path = []
    def dfs(state):
        if state == goal:
            return True
        visited.add(state)
        for nxt in get_moves(state):
            if nxt not in visited:
                path.append(nxt)
                if dfs(nxt):
                    return True
                path.pop()
        return False
    path.append(start)
    if dfs(start):
        return path
    return None
```

Same Problems



8-Queens

8		6
5	4	7
2	3	1

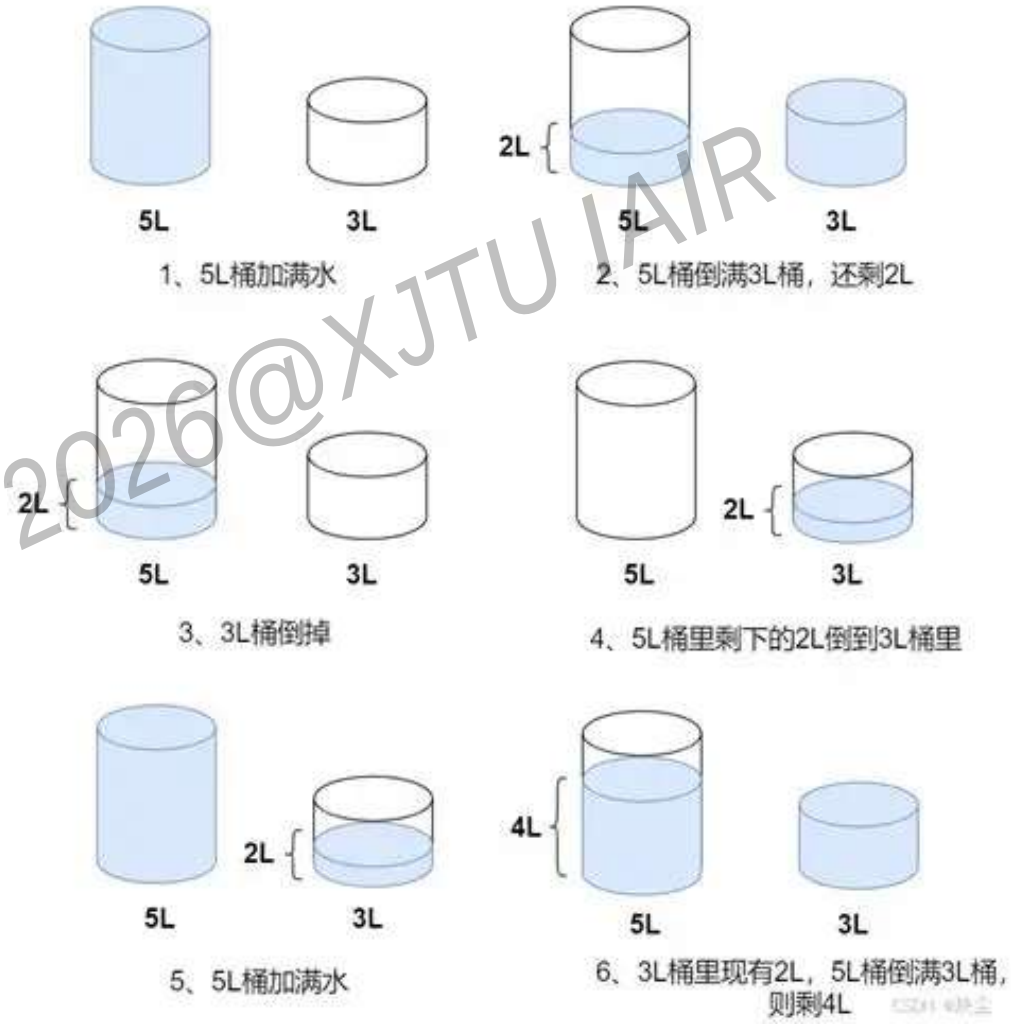
	1	2
3	4	5
6	7	8

8 Puzzle/Huarong Road Game

Same Problems

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

sudoku



Water and Jug Problem

8-Queens

State: An array *Q[0..7]* of length 8, where *Q[i]* indicates the column (0–7) of the queen on row i, with the constraint that no two queens share the same column or diagonal.

Initial state: Empty board, no queen placed (or *Q* array all *-1*).

Goal state: *Q[0..7]* all assigned, and for any two queens:

$Q[i] \neq Q[j]$ (different columns)

$|Q[i] - Q[j]| \neq |i - j|$ (different diagonals)

Transition rule: Place queens row by row. Starting from row 0, in the current row choose a column that does not conflict with already placed queens, then move to the next row. If no safe column exists in the current row, backtrack.

8-Queens

State space:

- If considering all possible board configurations (not row-by-row), the total number is $\binom{64}{8} \approx 4.4 \times 10^9$.
- Without conflict constraints, all possible column permutations give $8^8 = 16\,777\,216$ states, but valid states (no conflicts) are far fewer.
- Different queens on different rows: 8!
- The actual search tree has about 15 720 nodes, and there are only 92 distinct solutions.

Pruning strategy and effect:

Backtracking + constraint checking: after placing a queen, immediately check conflicts with all previously placed queens; if conflict occurs, prune that branch.
Effect: reduces the search space from exponential (8^8 to about 15 720 nodes, improving efficiency by several orders of magnitude, making it feasible to find all **92** solutions in a reasonable time.

Solution of n-Queen Problem

```
def solve_n_queens(n):
    def is_safe(board, row, col):
        for i in range(row):
            if board[i] == col or abs(board[i] - col) == abs(i - row):
                return False
        return True

    def backtrack(row):
        if row == n:
            solutions.append(board[:])
            return
        for col in range(n):
            if is_safe(board, row, col):
                board[row] = col
                backtrack(row + 1)
                board[row] = -1  # 回溯

    board = [-1] * n
    solutions = []
    backtrack(0)
    return solutions  # 返回列位置列表，可转换棋盘
```

8-Puzzle Problem

State: A 3 × 3 grid containing the numbers *1–8* and a blank (represented as 0). Can be represented as a one-dimensional array or a 2D array.

Initial state: Any given arrangement, e.g.

$$\begin{bmatrix} 4 & 2 & 3 \\ 1 & 6 & 8 \\ 5 & 7 & 0 \end{bmatrix}$$

Goal state: Usually

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

Transition rule: Swap the blank (0) with an adjacent tile (***up, down, left, right***) if the move is within bounds.

8-Puzzle Problem

State space: Total permutations $9! = 362\,880$, but due to *parity Invariant*, only half *181 440*（50%）are reachable.

Pruning strategy and effect:

- *A search** + heuristic function (e.g., Manhattan distance, misplaced tiles): expands nodes with the smallest estimated cost first, avoiding blind exploration.
- **Closed set**: records visited states to avoid re-expansion.
- **Bidirectional BFS**: searches simultaneously from the initial and goal states; meeting in the middle gives the shortest path.
  Effect: brute-force BFS may explore hundreds of thousands of nodes, while A* with a good heuristic reduces the node count to a few thousand or even a few hundred, quickly finding the optimal (minimum-move) solution.

Solution of 8-Puzzle Problem

```
goal = (1,2,3,4,5,6,7,8,0)
def solvable(puzzle):
    inv = sum(1 for i in range(9) for j in range(i+1,9)
              if puzzle[i] and puzzle[j] and puzzle[i] > puzzle[j])
    return inv % 2 == 0
def move(state, pos, new_pos):
    lst = list(state)
    lst[pos], lst[new_pos] = lst[new_pos], lst[pos]
    return tuple(lst)
def bfs(start):
    if not solvable(start): return None
    q = deque([(start, start.index(0), [])])
    seen = {start}
    while q:
        state, zero, path = q.popleft()
        if state == goal: return path
        r, c = divmod(zero, 3)
        for dr, dc, d in [(-1,0,'u'),(1,0,'d'),(0,-1,'l'),(0,1,'r')]:
            nr, nc = r+dr, c+dc
            if 0 <= nr < 3 and 0 <= nc < 3:
                nz = nr*3+nc
                nxt = move(state, zero, nz)
                if nxt not in seen:
                    seen.add(nxt)
                    q.append((nxt, nz, path+[d]))
    return None
```

Introduction to State Machines

A **state machine** is an abstract computational model consisting of:

- A set of possible states S (finite or infinite)
- An initial state $s_0 \in S$
- A set of state transition rules (transition function) $\delta: S \to S$ (or non-deterministic)

Stepwise process: Starting from the initial state, each step moves to the next state according to the rules, forming a sequence of states.

Floyd's Invariant Principle

Definition: A predicate $P(s)$ is called an **invariant** of a state machine if it satisfies:

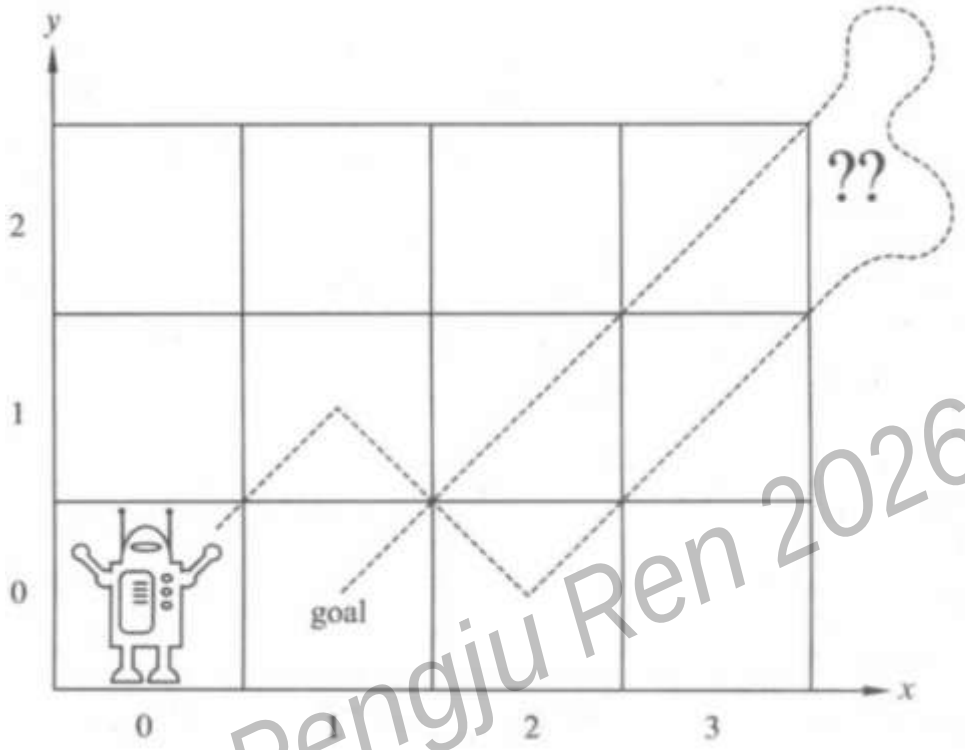
Base: $P(s_0)$ is true (the initial state satisfies P)

Inductive step: For any state s, if $P(s)$ is true and a transition leads to s' ,then $P(s')$ is also true.

In short: ***once true, always true***. An invariant is an "eternal truth" that holds throughout the entire execution.

Robert Floyd (Turing Award 1978) systematized the idea of invariants to prove the correctness of programs (or state machines).

A Robot Moving Diagonally



$(x, y) \rightarrow (x \pm 1, y \pm 1)$

Starting from position (0,0), can it move to (5,0)?

Invariant: the parity of $x + y$ remains unchanged.

Therefore, starting from (x_0, y_0) ,the robot can only reach positions where $x + y$ has the same parity as $x_0 + y_0$

Can Wall-E find its Eva？

Wall-E is wandering in a two-dimensional grid. Starting from (0, 0), it can only move in the following four steps: *1. (+2, -1), 2. (+1, -2), 3. (+1, +1), 4. (-3, 0)*. Can it reach the position Eva's location *(0, 100) or (1000003, 1000005) or (123456789,987654321)*?

Can Wall-E find its Eva？

Wall-E is wandering in a 3-dimensional grid. Starting from (0, 0, 0), it can only move in the following seven steps: *(1,1,1),(1,1,−2),(1,−2,1),(−2,1,1),(2,−1,−1),(−1,2,−1),(−1,−1,2).*

Can it reach the position Eva's location *(2,5,8) or (1000003, 1000004,1000005) or (123456789,987654321,112358130)*?

What's the Invariant of 15-puzzle?

The parity of (the number of inversions of the permutation + the row number of the blank counted from the bottom) remains unchanged.

Water Jug Problem

State: An n-tuple $(w_1, w_2, \ldots, w_n)$ where w_i is the current amount of water in jug i, each jug having capacity C_i ,with $0 \leq w_i \leq C_i$.

Initial state: Often $(0, 0, \ldots, 0)$, For example, two jugs of capacities 5 and 3, initially empty: (0,0).

Goal state: Some jug contains exactly the target amount T (e.g., 4 litres), i.e., $\exists i: w_i = T$.

Transition rule: Each move can be one of:

- Fill jug i to its capacity: $w_i \leftarrow C_i$
- Empty jug i: $w_i \leftarrow 0$
- Pour from jug i into jug j until i is empty or j is full:

$$\textbf{pour} = \min(w_i, C_j - w_j), w_i \leftarrow w_i - \textbf{pour}, w_j \leftarrow w_j + \textbf{pour}$$

Water Jug Problem

State space: Finite. For two jugs, the total number of theoretical states is $(C_1 + 1)(C_2 + 1)$,*but the reachable states are often fewer*. For capacities 5 and 3, theoretical states = 24, but only 8 are reachable.

Pruning strategy and effect:

- **Visited set**: records already explored states to avoid re-expansion (prevents cycles).
- **BFS** or **bidirectional BFS**: guarantees finding the shortest sequence of moves.
- **Effect**: the visited set turns an infinite loop into a finite search; BFS ensures an optimal solution. Because the state space is typically small (often <200 states), solutions are found in milliseconds.

Sudoku Problem (9 × 9)

State: A 9 × 9 integer matrix, each cell containing 0 (*empty*) or 1–9 (*filled*), with some cells fixed as the initial puzzle clues.

Initial state: The given Sudoku puzzle, i.e., some cells contain given numbers, the rest are *empty* (0).

Goal state: All cells contain numbers *1–9*, satisfying:

- Each row has no repeated digits
- Each column has no repeated digits
- Each 3 × 3 subgrid (box) has no repeated digits

Transition rule: Choose an empty cell, try to place a digit from 1–9 that does not violate the row, column, and box constraints for that cell; if valid, move to the next empty cell; otherwise backtrack.

Sudoku Problem (9×9)

State space: Theoretically 9^{81} possible fillings, but most are eliminated by the strong constraints. The number of effective branches drops sharply as digits are placed. Typical 9×9 Sudokus have a unique or very few solutions.

Pruning strategy and effect:

- **Constraint propagation**: after placing a digit, immediately update the candidate sets of affected cells, eliminating impossible values.
- **Minimum Remaining Values (MRV) heuristic**: choose the empty cell with the fewest candidate digits next.
- **Forward checking** during backtracking: if a cell's candidate set becomes empty, prune immediately.
- **Effect**: compresses the exponential explosion to a manageable scale. Modern solvers can solve most Sudokus in milliseconds to seconds without exhaustive enumeration.

Sudoku Problem (9×9)

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Minimum Remaining Values (MRV)
heuristic : choose the variable with the fewest legal values

A4={1, 2, 3, 4, 5, 6, 7, 8, 9}
A6={1, 2, 3, 4, 5, 6, 7, 8, 9}
E3={1, 2, 3, 4, 5, 6, 7, 8, 9}
E6={1, 2, 3, 4, 5, 6, 7, 8, 9}
I6 ={1, 2, 3, 4, 5, 6, 7, 8, 9}

Solution of SudoKu Problem

```
def solve_sudoku(board):
    def find_empty():
        for i in range(9):
            for j in range(9):
                if board[i][j] == 0:
                    return (i, j)
        return None
    def is_valid(num, pos):
        r,c = pos
        # 检查行和列
        for i in range(9):
            if board[r][i] == num or board[i][c] == num:
                return False
        # 检查3x3宫
        br, bc = r//3 * 3, c//3 * 3
        for i in range(br, br+3):
            for j in range(bc, bc+3):
                if board[i][j] == num:
                    return False
        return True
    def backtrack():
        empty = find_empty()
        if not empty:
            return True
        r,c = empty
        for num in range(1,10):
            if is_valid(num, (r,c)):
                board[r][c] = num
                if backtrack():
                    return True
                board[r][c] = 0  # 回溯
        return False
    backtrack()
    return board
```

Tree Search Algorithms

- **A strategy is defined by picking the order of node expansion**
- **Strategies are evaluated along the following dimensions:**
  - **Completeness -** does it always find a solution if one exists?
  - **Optimality -** does it always find a least-cost solution?
  - **Time complexity** - number of nodes generated/expanded
  - **Space complexity** - maximum number of nodes in memory
- **Time and space complexity are measured in terms of**
  - **b -** maximum branching factor of the search tree
  - **d -** depth of the least-cost solution (optimal one)
  - **m -** maximum depth of the state space (may be ∞)

Uninformed search strategies

Uninformed strategies use only the information available in the problem definition.

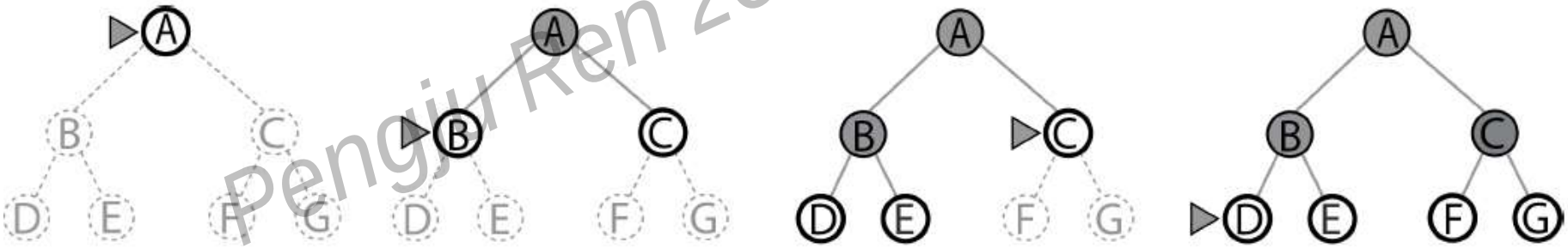
- **Breadth-first search (BFS)**
- **Depth-first search (DFS)**
- **Depth-limited search (DLS)**
- **Iterative deepening search(IDS)**

Breadth-First Search (BFS)

Expand the shallowest unexpanded node.

Implementation:

fringe is a FIFO queue, i.e., new successors go at end



BFS-Map Navigation

Properties of BFS

- **Completeness:** Yes (if b is finite)
- **Optimality:** No, Yes only if the path cost is a *non-decreasing* function of the depth of the node; not optimal in general
- **Time complexity:** $1 + b^1 + b^2 + b^3 + b^d = O(b^d)$ or $O(b^{d+1})$ if goal test is applied after expansion.
- **Space complexity:** $O(b^d)$ (keeps every node in memory)

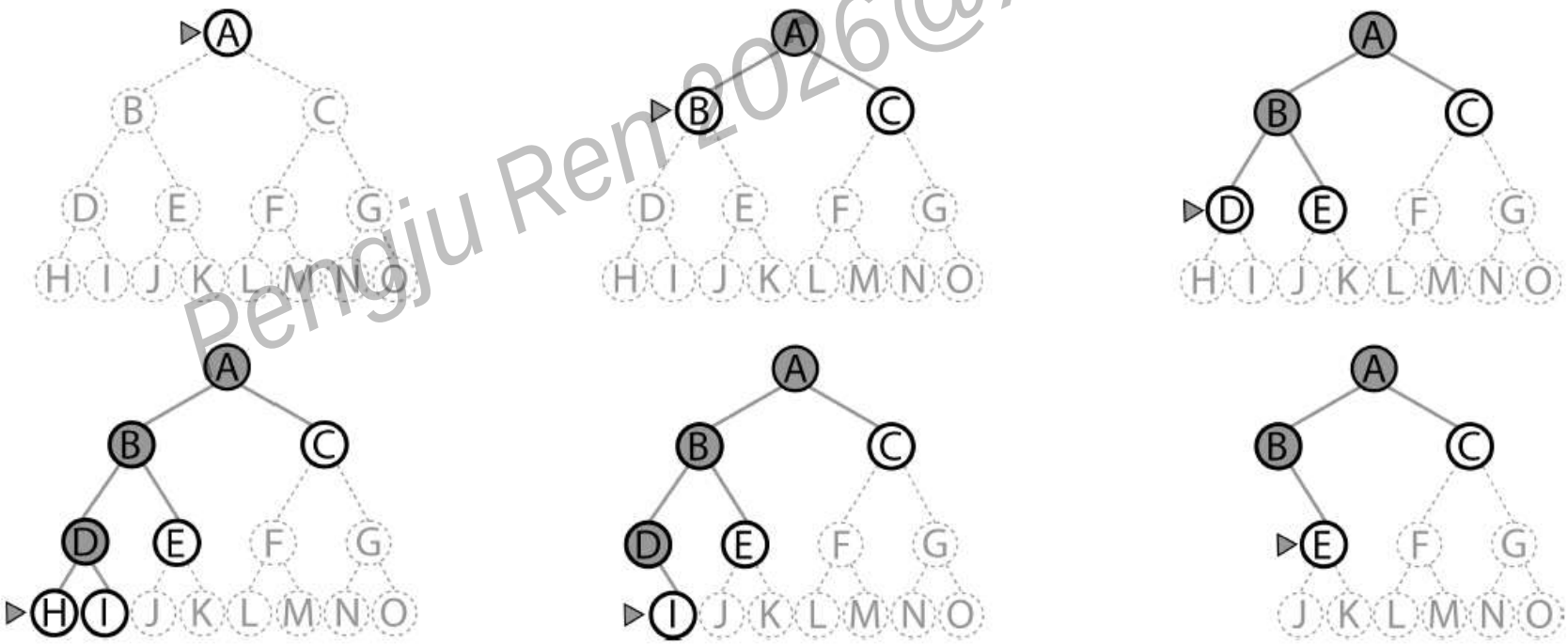
Space is the big problem; can easily generate nodes at 100MB/sec so 24hrs = 8640GB.

Depth-First Search (DFS)

Expand the deepest unexpanded node.

Implementation:

fringe is a LIFO queue, i.e., new successors go at front



Properties of DFS

- **Completeness:** No, fails in infinite-depth spaces, spaces with loops. Modify to avoid repeated states along path -> complete in finite spaces.
- **Optimality: No**
- **Time complexity :** $O(b^m)$ terrible if m is much greater than d.

But if solutions are dense, may be much faster than breadth-first

- **Space complexity:** $O(bm)$ linear space!

Backtracking technique only generate one successor instead of all successors

Depth-Limited Search (DLS)

- DFS never terminates if m -> ∞.
- DLS = DFS with depth limit *l* ,
- Nodes at depth *l* have no successors
- Recursive implementation:

RECURSIVE-DLS(*node*, *problem*, *limit*)

```
1  if problem.GOAL-TEST(node.state)
2        return SOLUTION(node)
3  elseif limit == 0
4        return cutoff
5  else
6        cutoff_occurred = FALSE
7        for each action in problem.ACTIONS(node.state)
8              child = CHILD-NODE(problem, node, action)
9              result = RECURSIVE-DLS(child, problem, limit − 1)
10             if result == cutoff
11                   cutoff_occurred = TRUE
12             elseif result ≠ failure
13                   return result
14       if cutoff_occurred
15             return cutoff
16       else
17             return failure
```

Properties of DLS

- **Completeness:** Not complete if $l < d$; complete otherwise.
- **Optimality:** Not optimal in general (even if $l > d$).
- **Time complexity :** $O(b^l)$
- **Space complexity:** $O(bm)$ linear space
- **Two termination conditions:**

  failure: no solution.

  cutoff : no solution within the depth limit.

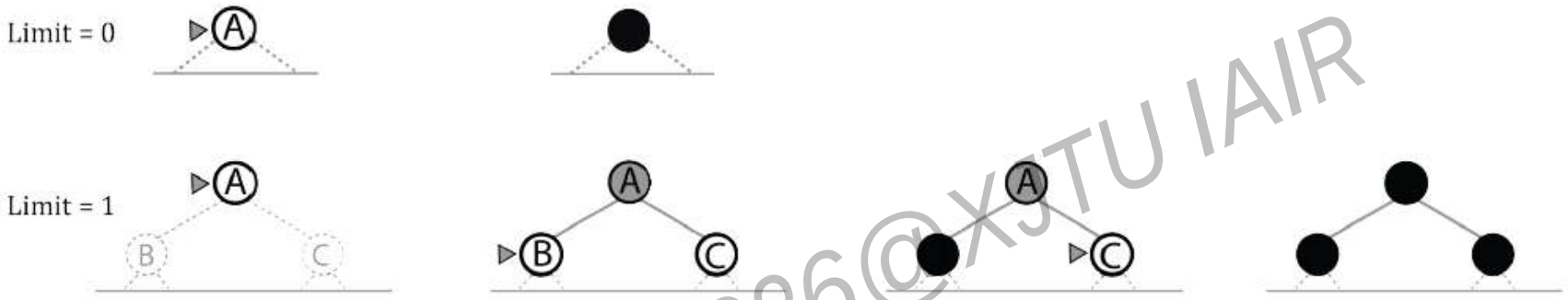
Iterative-Deepening Search (IDS)

- Call DLS iteratively with increasing depth limit.
- Seems to be wasteful, but actually not.
- Combine the benefits of BFS and DFS.

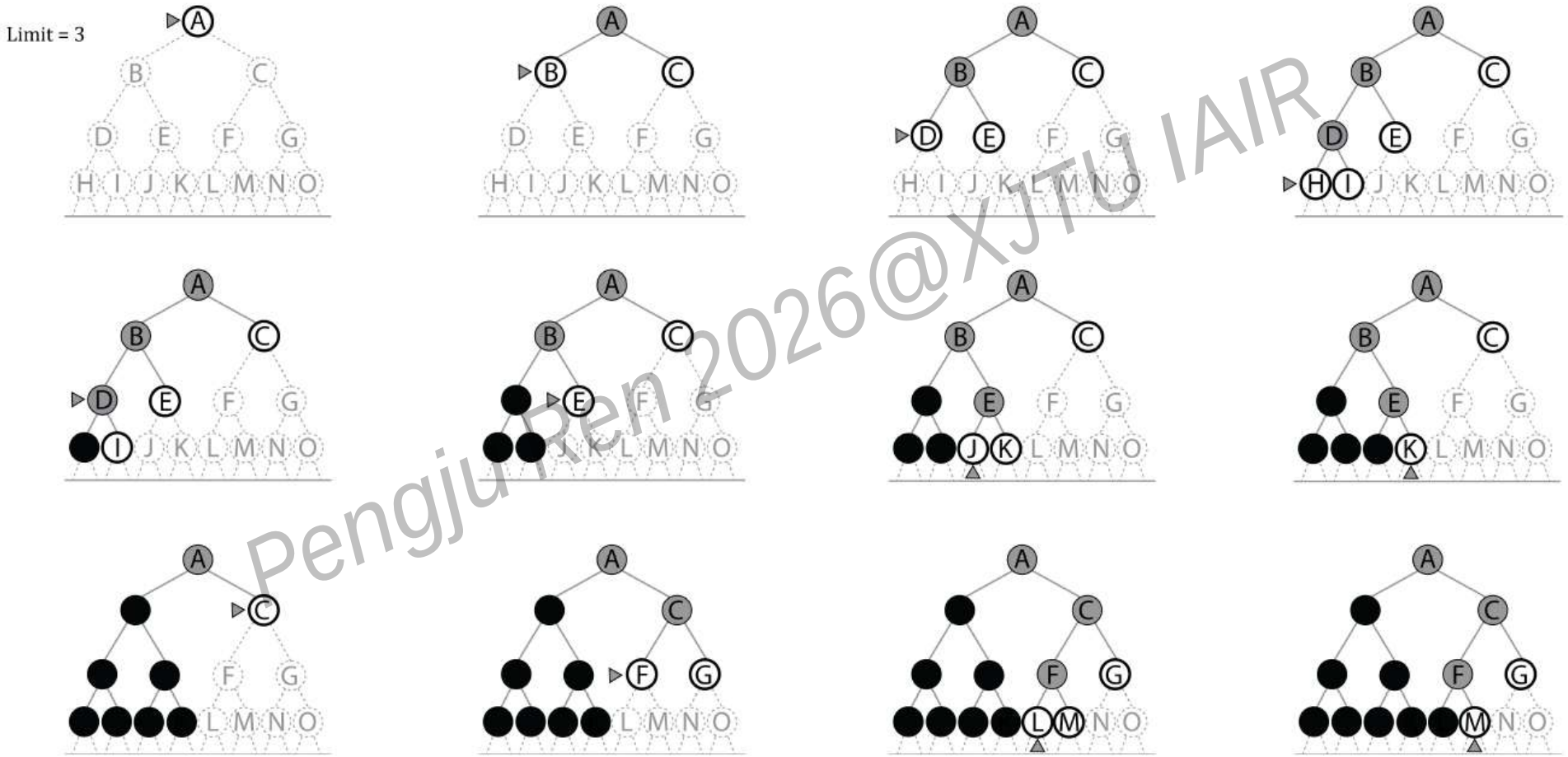
ITERATIVE-DEEPENING-SEARCH(*problem*)

```
1  for depth = 0 to ∞
2      result = DEPTH-LIMITED-SEARCH(problem, depth)
3      if result ≠ cutoff
4          return result
```

Iterative-Deepening Search (IDS)



Iterative-Deepening Search (IDS)



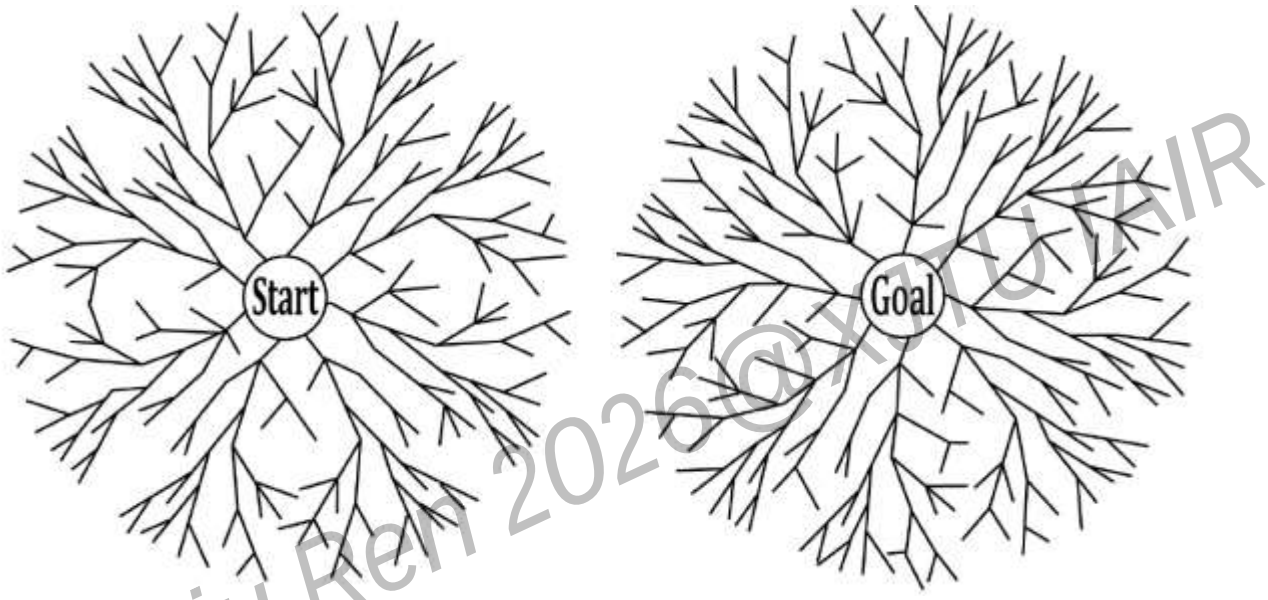
Properties of IDS

- **Completeness:** Not complete if $l < d$; complete otherwise.
- **Optimality:** Not optimal in general (even if $l > d$).
- **Time complexity :** $O(b^l)$
- **Space complexity:** $O(bl)$

Properties of DLS

- **Completeness:** Yes
- **Optimality:** Yes
- **Time complexity :** $1 + b^1 + b^2 + b^3 + b^d = O(b^d)$
- **Space complexity:** $O(bd)$

Bidirectional Search



- Reduce the time complexity from $O(b^d)$ to $O(b^{d/2})$.
- Though the reduction is attractive, how to search backward?
- Need *PREDECESSORS* and known *GOAL*.
- Also, the space complexity increases to $O(b^{d/2})$ as well, can be problematic.

Backtracking v.s Branch and Bound

E.g. Three students in a row: two Boys and one Girl

Backtracking and Branch and Bound are both systematic methods for searching a *state space tree*, commonly used to solve combinatorial optimization and constraint satisfaction problems.

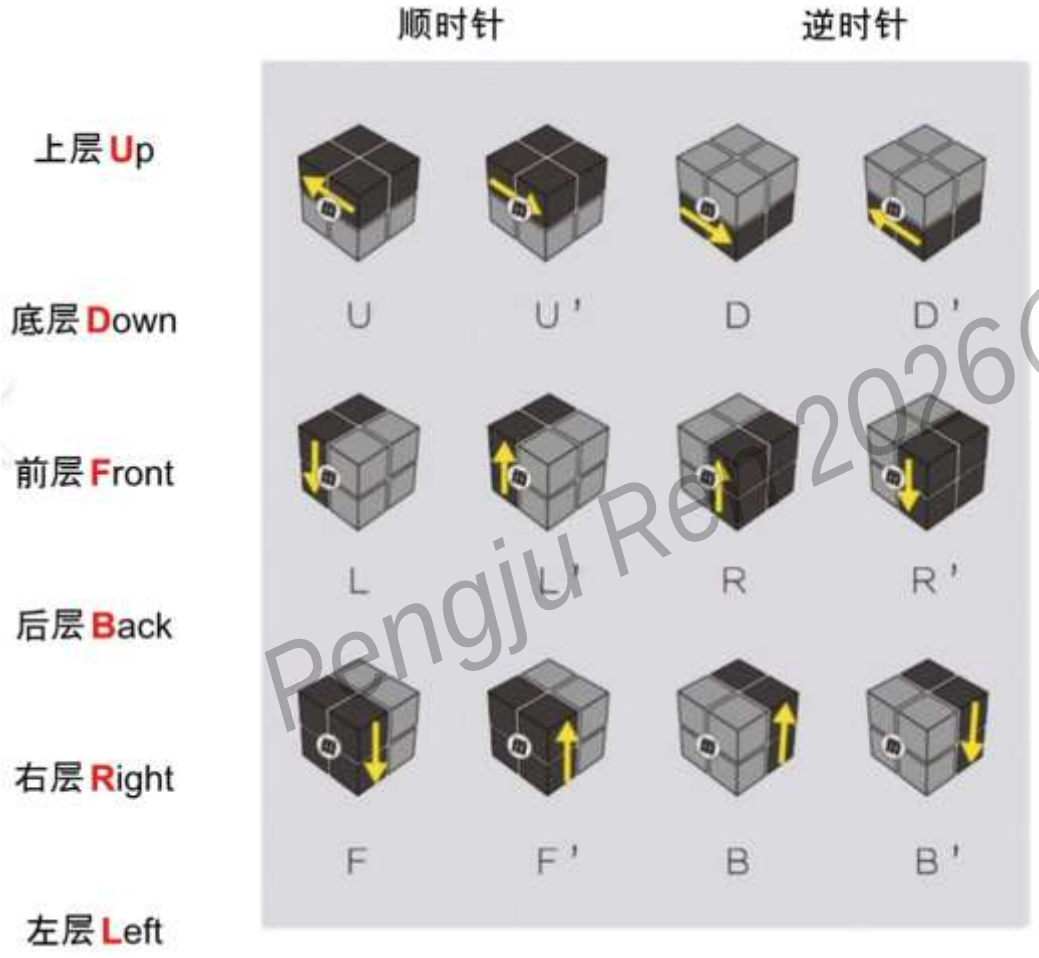
Both use pruning: Both stop exploring a branch when it is found that it cannot lead to a feasible solution or an optimal solution.

Backtracking usually use *DFS*, while Branch and Bound choose *BFS*

Summary

- ***State machine representation*:** Abstract the problem into states, initial state, goal state, and transition rules to form a search tree.
- ***Reachability analysis and invariants:*** Identify quantities that remain unchanged during transitions (e.g., parity, modular constraints) to *quickly detect unsolvable cases* or *prune dead branches*.
- ***Efficient search algorithms:*** *Choose depth-first (DFS), breadth-first (BFS), or heuristic search* based on problem characteristics, balancing memory and time.
- ***Search optimization techniques:*** Use pruning, memorization (visited set), heuristic ordering, and symmetry reduction to dramatically shrink the search space.

Can you solve Cube Game using S&B?



2x2 Cube the State Space

is : $\frac{8! \times 3^7}{24} = 3\,674\,160$

3x3 Cube the State Space

is: $\frac{8! \times 3^7 \times 12! \times 2^{11}}{2}$