Introduction to AI

Chapter03 Solving Problems by Uninformed Searching(3.1~3.4)

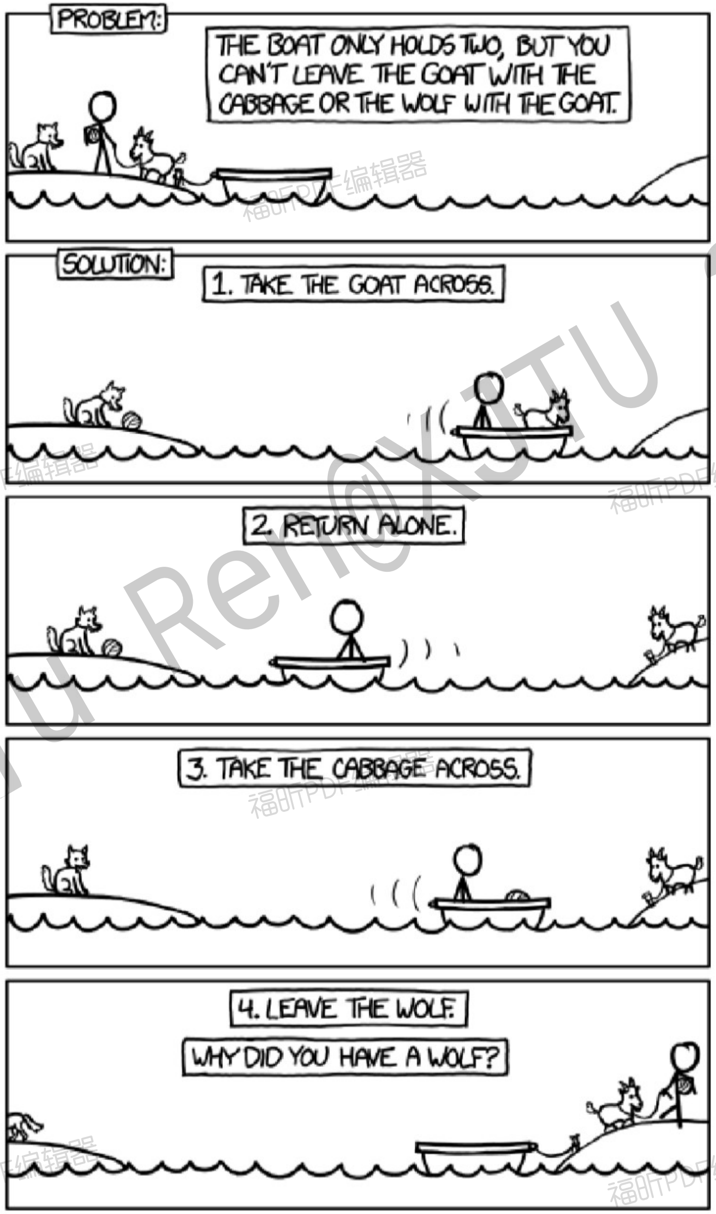
Pengju Ren@IAIR

How an agent can find a sequence of actions that achieves its goals when no single action will do.

Outline

- Problem-solving agents
- Problem types
- Problem formulation
- Search on Trees and Graphs
- Uninformed algorithms
  - Breadth-First
  - Uniform-Cost
  - Depth-First
  - Depth-Limited
  - Iterative Deepening
  - Bidirectional

Question: Famer, wolf, cabbage, and goat ?



Example: Map Navigation

- Currently in East Door of Peking Univ.(EDPU)
- Every 2mins a subway train leaves from
- Formulate goal
  - Be in Beijing Station.
- Formulate problem
  - States: various Subway stations
  - Actions: train between Subway stations
- Find solution
  - Sequence of actions (trains taken between Subway stations, e.g., EDPU, National Library, Xuanwu, Qianmen, Beijing Station)

Example: Map Navigation

Problem formulation: Navigation

- A problem is defined by five components
  ① **Initial state:** In(EDPU)
  ② **Actions:**
  ACTION(In(EDPU))={Go(Zhongguan Cun); Go(WuDao Kou)}
  ③ **Transition model** *RESULT(s; a):*
  RESULT(In(EDPU); Go(ZGC))=In(ZGC).
  Successor S(s): states reachable by a single action.
  $$S(s) = \{s' | \forall \partial \in ACTION(s), s' = RESULT(s, a)\}$$
  ④ **Goal test**: {*In(Beijing Station)*}
  ⑤ **Path cost** (additive)
  Sum of distances, number of actions executed, etc.
  $c(s, a, s')$ is the *step cost* of taking action *a* in state *s* to reach state *s'*, assumed to be ≥0
- A solution is a sequence of actions leading from the initial state to the goal state.

Problem-Solving Agents

A simple problem-solving agent formulates a goal and a problem, searches for a sequence of actions that solves the problem, and then execute the actions one by one.

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
            state, some description of the current world state
            goal, a goal, initially null
            problem, a problem formulation

    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← RECOMMENDATION(seq, state)
    seq ← REMAINDER(seq, state)
    return action
```

Note: this is offline problem solving (is uninformed or with complete knowledge) ;
Online problem solving involves acting without complete knowledge.

Problem types

Deterministic, fully observable => **single-state problem**
Agent knows exactly which state it will be in; solution is a sequence

Non-observable => **conformant problem**
Agent may have no idea where it is; solution (if *any*) is a sequence

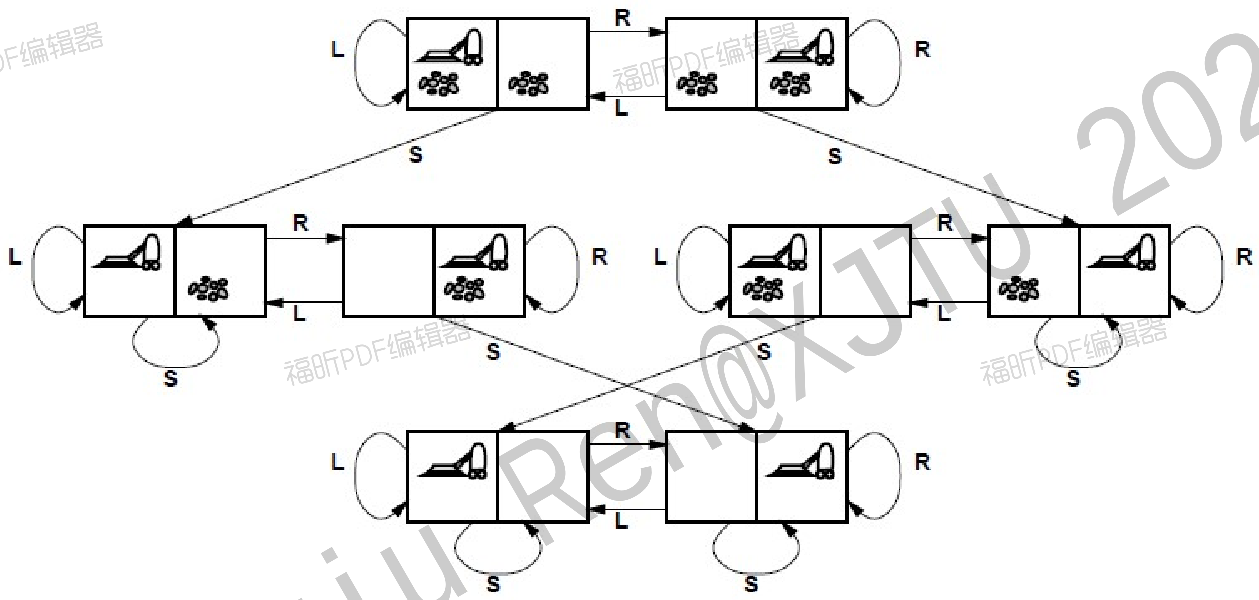
Nondeterministic **and/or** **partially observable** => **contingency problem**
percepts provide new information about current state
solution is a contingent plan or a policy
often interleave search, execution

Unknown state space => **exploration problem** (“online")

Abstraction

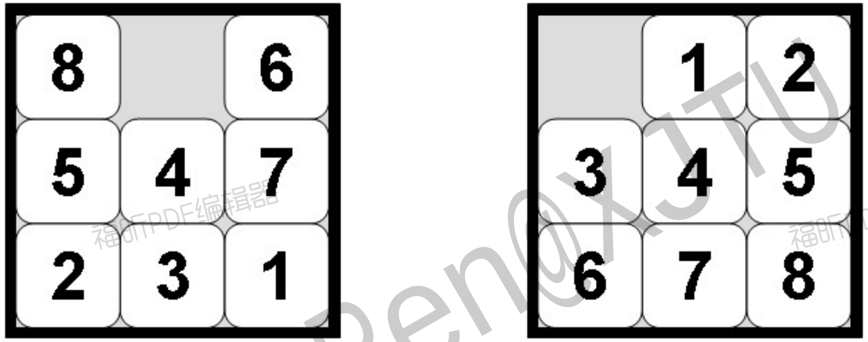
- Real world is absurdly complex
  State space must be **abstracted** for problem solving.
- (Abstract) state = subset of real states
- (Abstract) action = complex combination of real actions
  Go(ZGC) represents a complex set of possible routes, detours, rest, stops, interrupt, etc.
- For guaranteed realizability, **any** real state "in EDBU" must get to **some** real state "in ZGC"
- (Abstract) solution = set of real paths that are solutions in the real world
- Each abstract action should be "easier" than the original problem!

E.g. Vacuum World State Space Graph



- **Initial state**: Any one of the above states. (ignore dirt amounts etc.)
- **Actions**: Left, Right, Suck, NoOp
- **Transition model:** The above figure.
- **Goal test:** no dirt
- **Path cost:** 1 per action (0 for NoOp)

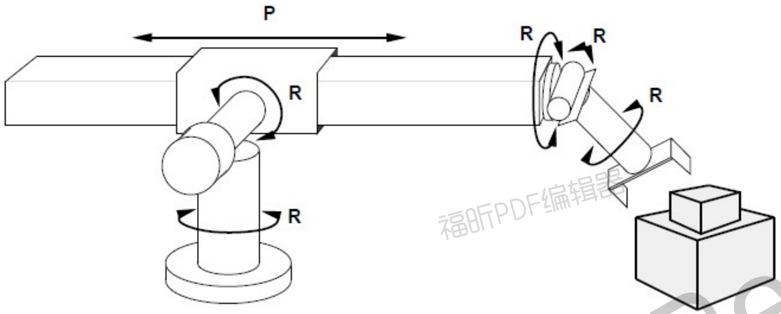
Eg. The eight-puzzle

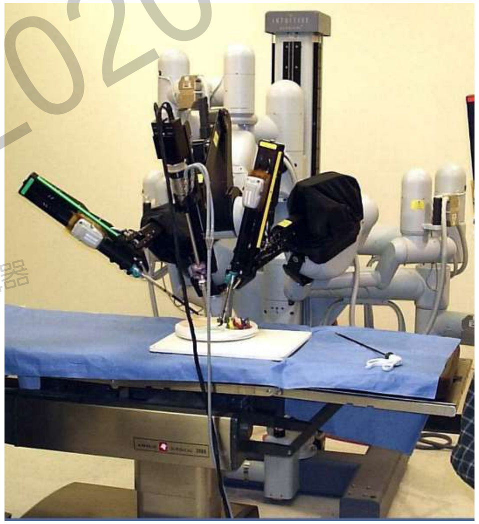


- **Initial state**: The left figure
- **states:** integer locations of tiles (ignore intermediate positions)
- **actions:** move **blank** left, right, up, down (ignore unjamming etc.)
- **goal test:** goal state, the right figure
- **path cost:** 1 per move

Note: optimal solution of Sliding-block Puzzle is NP-hard

Eg. Robotic assembly



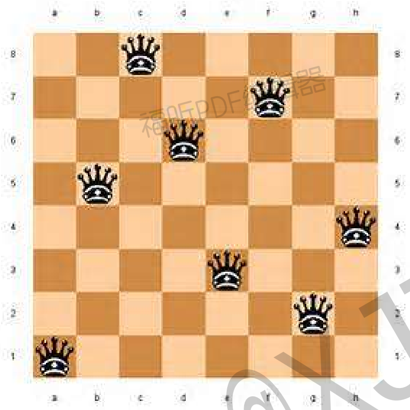


- **Initial state:** real-valued coordinates of robot joint angles
  parts of the object to be assembled
- **Actions:** continuous motions of robot joints
- **Transition model:** Intermedia coordinates of robot joint angles
- **Goal test:** complete assembly
- **Path cost:** time to execute

E.g. Eight-Queen Puzzle

Initial state: No queen on the board.
Actions: Add a queen on the board where the square is empty.
Transition model: Returns the board with a queen added to the specified square.
Goal test: 8 queens are on the board, none attacked.
Path cost: Number of trials.

E.g. Eight-Queen Puzzle



- **States:** Any 0~8 queens on the board.
  State space: $C_{64}^0 + C_{64}^1 + C_{64}^2 + \ldots + C_{64}^8 \simeq 5.1 \times 10^9$
  Solution space: $64 \times 63 \times 62 \times \ldots \times (64-7) \simeq 1.8 \times 10^{14}$
- **States:** One queen per column.
  State space: $8^0 + 8^1 + 8^2 + \ldots + 8^8 \simeq 1.9 \times 10^7$
  Solution space: $8^8 \simeq 1.6 \times 10^7$
- **States:** All possible arrangements of n (0 n 8) queens at leftmost n columns with on queen attacked.
  Actions: Add a queen to the next column with no queen attacked, or backtrack.
  State space: *2057.*

Tree Search Algorithms

Basic idea:

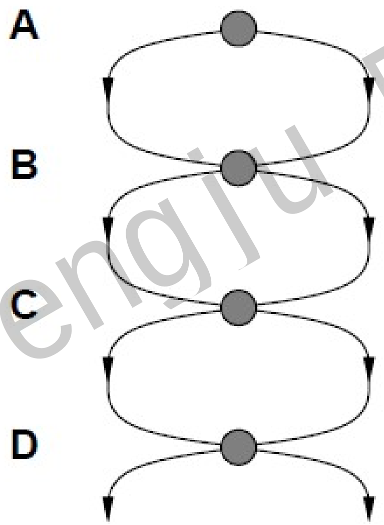
Offine, simulated exploration of state space
by generating successors of already-explored states
(a.k.a. expanding states)

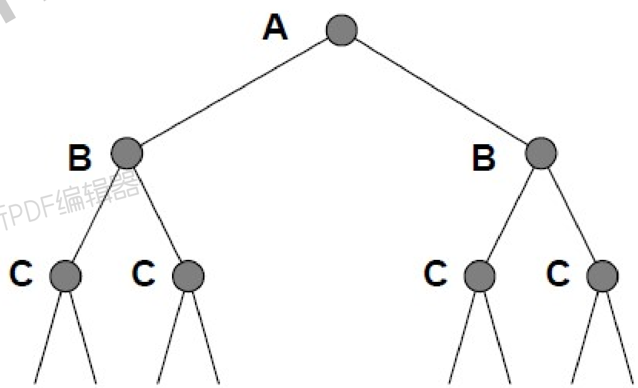
```
TREE-SEARCH(problem)

initialize the frontier using the initial state of problem
repeat
    if the frontier is empty
        return failure
    choose a leaf node and remove it from the frontier.
    if the node contains a goal state
        return the corresponding solution
    expand the chosen node
    add the resulting nodes to the frontier
```

Repeated States in Graph Search

- Failure to detect repeated states can turn a linear problem into an exponential one!
- Use a **queue** to record explored states.
- For fast detection of repeated states, **hashing** techniques are usually adopted.

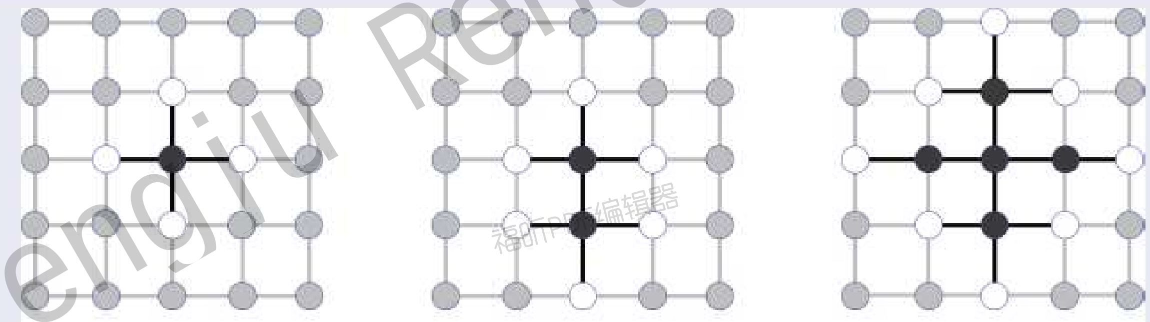




Graph Search, Tree Search and Frontier Separation

The frontier separates the state space into explored and unexplored regions (loop invariant proof).

Separation property of GRAPH-SEARCH



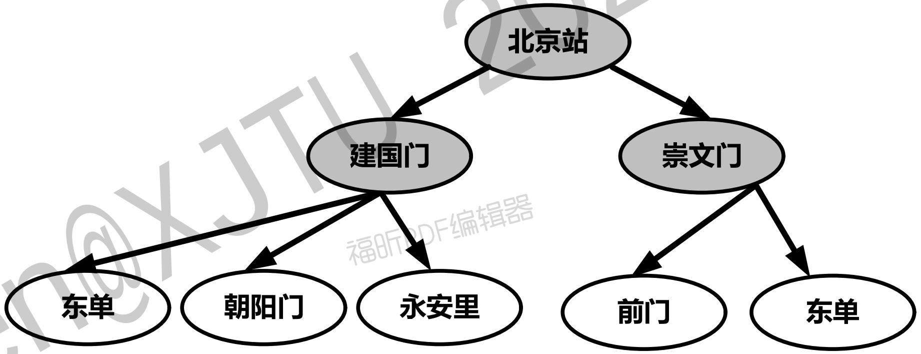
Graph Search Algorithm

GRAPH-SEARCH(*problem*)

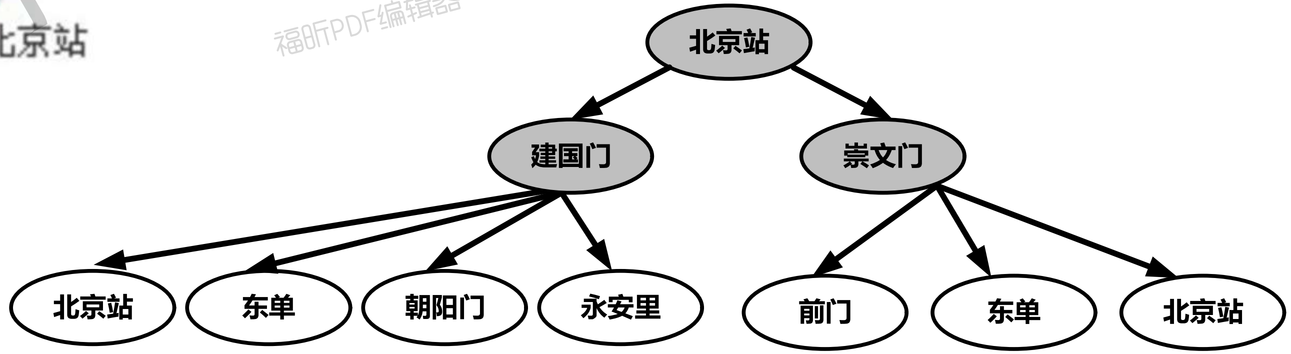
```
initialize the frontier using the initial state of problem
initialize the explored set to be empty
repeat
    if the frontier is empty
        return failure
    choose a leaf node and remove it from the frontier.
    if the node contains a goal state
        return the corresponding solution
    add the node to the explored set
    expand the chosen node
    if not in the frontier or explored set
        add the resulting nodes to the frontier
```

Graph Search Algorithm

Graph Search

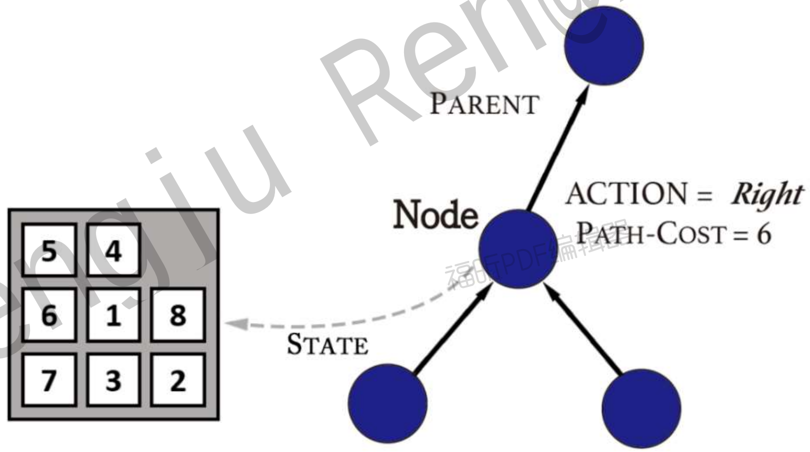


Tree Search



Implementation: States vs. Nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes *parent, children, depth, path cost g(x)*
- States do not have parents, children, depth, or path cost!



The *EXPEND function* creates new nodes, filling in the various fields and using the *SUCCESSOR function* of the problem to create the corresponding states.

Implementation: General Tree Search

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE(node)) then return node
        fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

```
function EXPAND( node, problem) returns a set of nodes
    successors ← the empty set
    for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
        s ← a new NODE
        PARENT-NODE[s] ← node;  ACTION[s] ← action;  STATE[s] ← result
        PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
        DEPTH[s] ← DEPTH[node] + 1
        add s to successors
    return successors
```

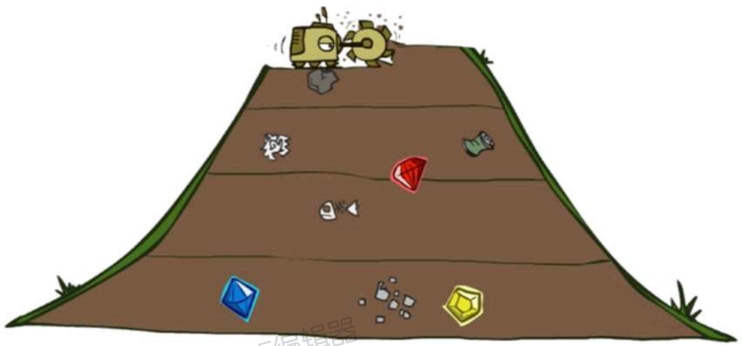
Tree Search Algorithms

- 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
  - 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)**
- **Uniform-cost search**
- **Depth-first search (DFS)**
- **Depth-limited search (DLS)**
- **Iterative deepening search(IDS)**

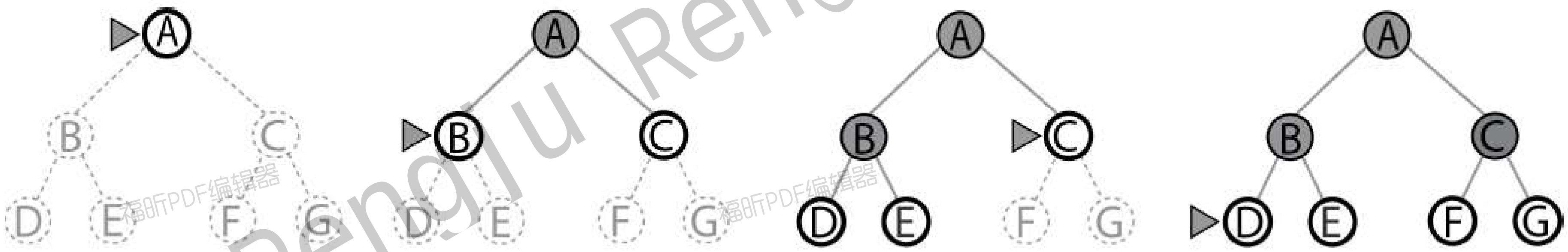


Breadth-First Search (BFS)

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**: $1 + b^1 + b^2 + \ldots + b^d = O(b^d)$ 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.

Uniform-cost search

- Expand **least-cost unexpanded node**
- Implementation:
  fringe = queue ordered by path cost, lowest first
- Equivalent to breadth-first if step costs all equal

Properties of Uniform-cost search:

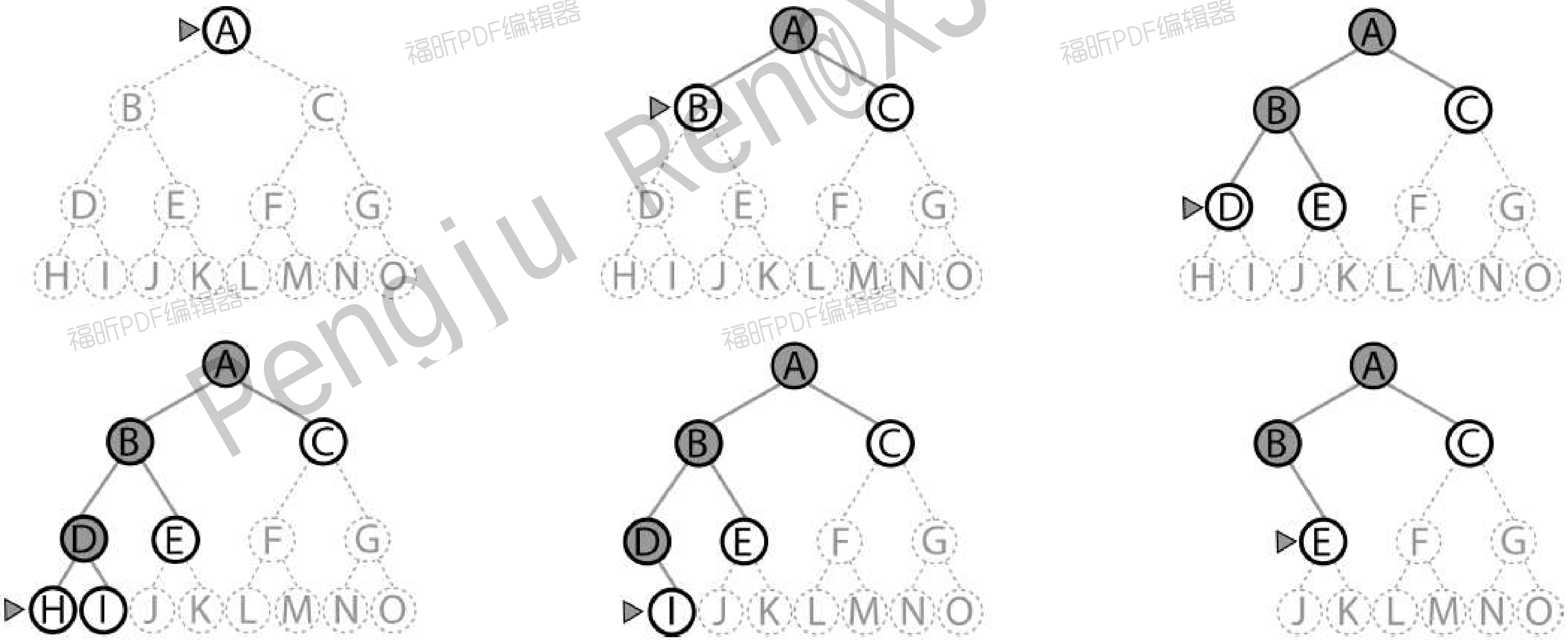
- Completeness: Yes, if step $\text{cost} > \varepsilon > 0$.
- Optimality: Yes - nodes expanded in increasing order of $g(n)$.
- Time complexity: # of nodes with $g < \text{cost}$ of optimal solution. Maximum depth is given by $1 + \lfloor C^*/\varepsilon \rfloor$, where C^* is the cost of the optimal solution. $O(b^{\lfloor C^*/\varepsilon \rfloor})$
- Space complexity: # of nodes with g cost of optimal solution, $O(b^{\lfloor C^*/\varepsilon \rfloor})$

Depth-First Search (DFS)

Expand the deepest unexpanded node.

Implementation:

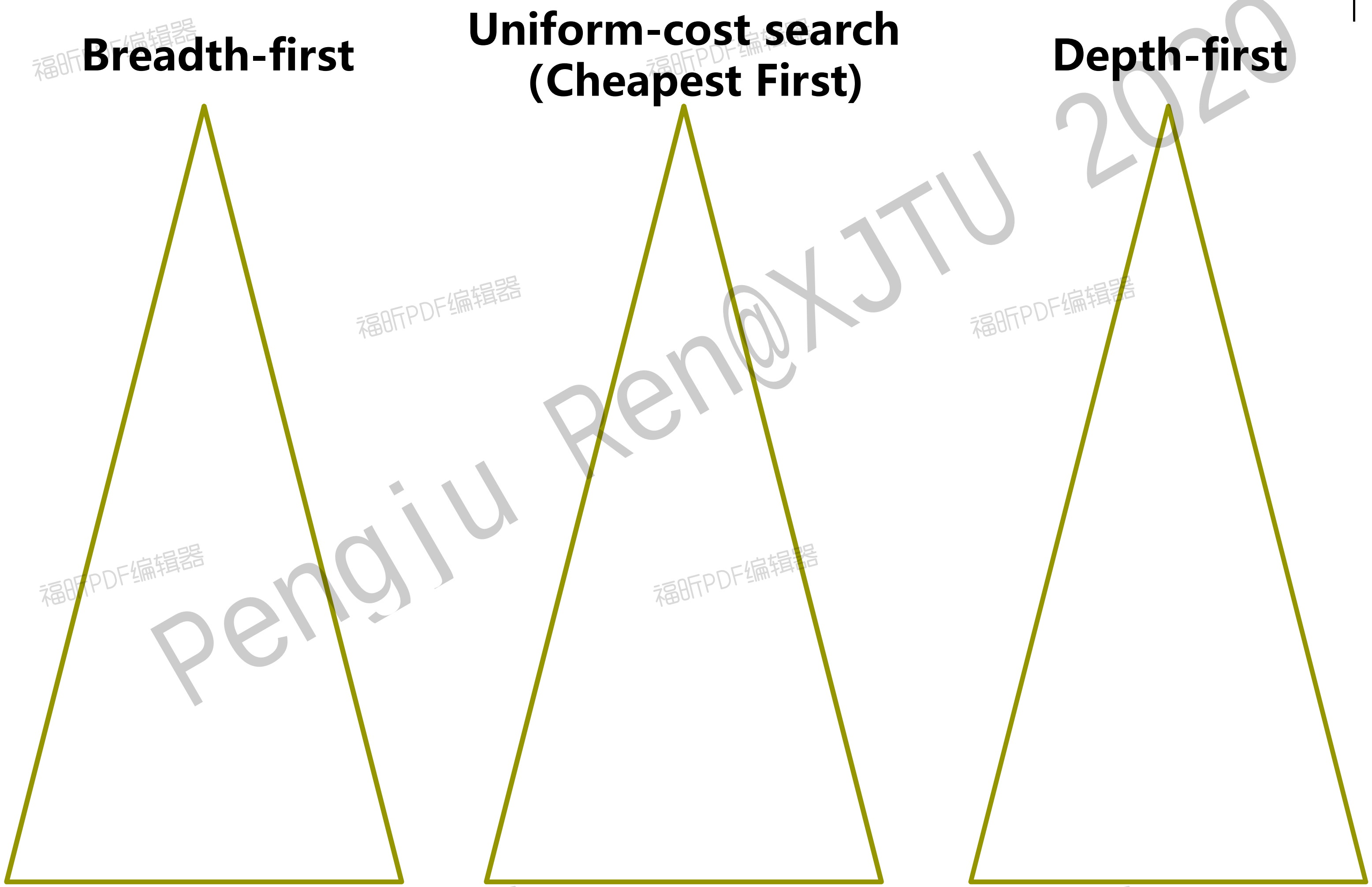
fringe is a LIFO queue, i.e., new successors go at front



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 -> $O(m)$.

Search Comparson



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*)

```
if problem.GOAL-TEST(node.state)
      return SOLUTION(node)
elseif limit == 0
      return cutoff
else
      cutoff_occurred = FALSE
      for each action in problem.ACTIONS(node.state)
            child = CHILD-NODE(problem, node, action)
            result = RECURSIVE-DLS(child, problem, limit − 1)
            if result == cutoff
                  cutoff_occurred = TRUE
            elseif result ≠ failure
                  return result
      if cutoff_occurred
            return cutoff
      else
            return failure
```

Properties of DLS

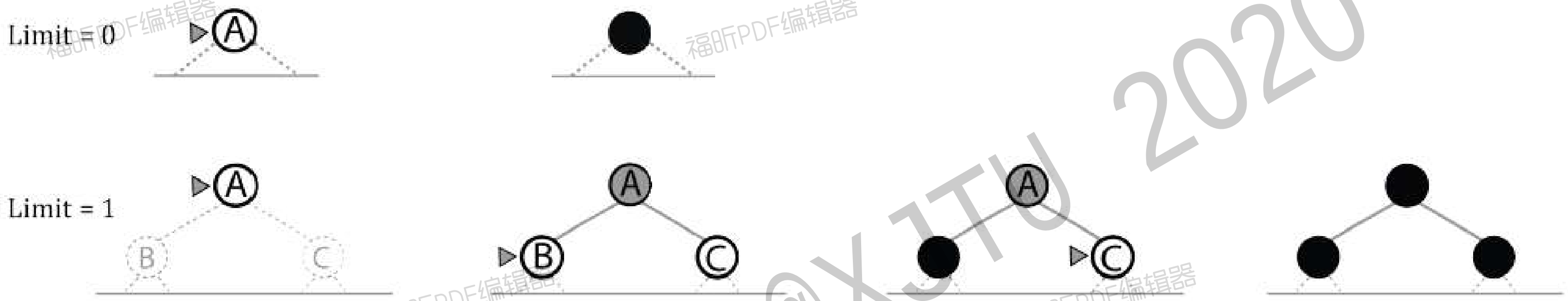
- **Completeness:** Not complete if l < d; complete otherwise.
- **Optimality:** Not optimal in general (even if l > d).
- **Time complexity** : $O(b^l)$
- **Space complexity:** $O(bl)$ linear space
- **Two termination conditions:**
  - ***failure:*** no solution.
  - ***cutoff :*** no solution within the depth limit.

Iterative-Deepening Search (IDS)

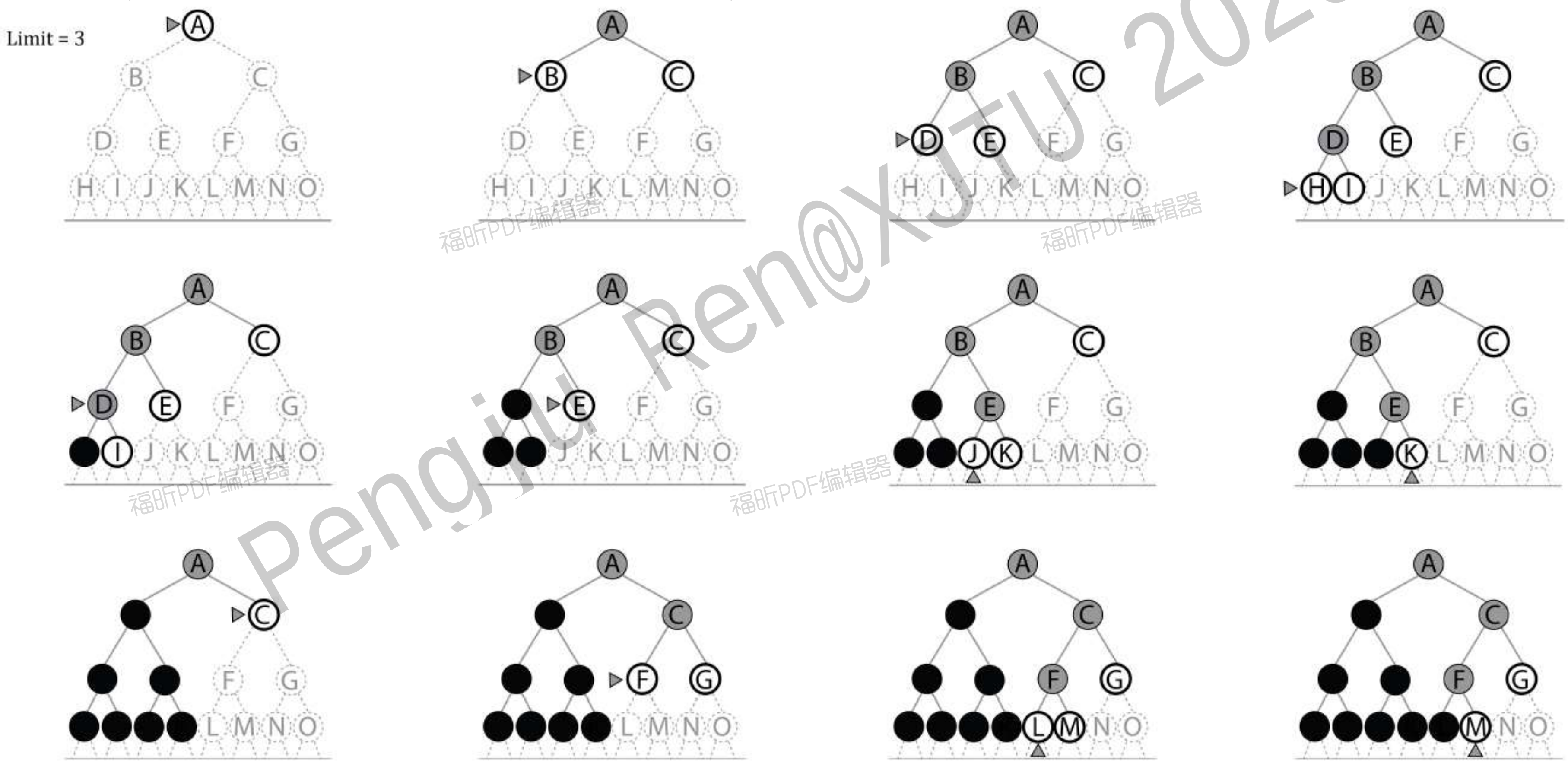
- Call DLS iteratively with increasing depth limit.
- Seems to be wasteful, but actually not.
- Combine the benefits of BFS and DFS.

```
ITERATIVE-DEEPENING-SEARCH(problem)
for depth = 0 to ∞
    result = DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff
        return result
```

Iterative-Deepening Search (IDS)



Iterative-Deepening Search (IDS)



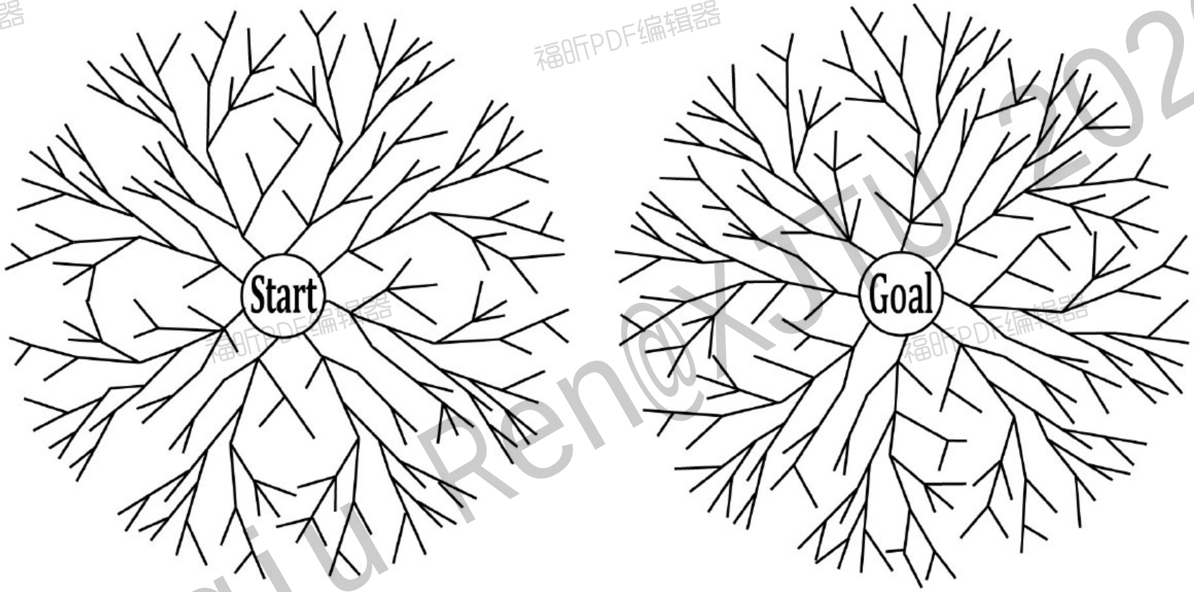
Properties of IDS

- **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 :** $O(b^1 + b^2 + \cdots \ldots b^d) \approx O(b^d)$
- **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.

Summary of Algorithms

Criterion	BFS	Uniform-Cost	DFS	DLS	IDS	Bi-Directional
Completeness	Yes[a]	Yes[b]	No	No[c]	Yes[a]	Yes[d]
Optimality	Yes[e]	Yes	No	No	Yes[e]	Yes[e]
Time Complexity	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

[a] if b is finite

[b] if b is finite and step cost $\geq \epsilon$

[c] unless $\ell \geq d$

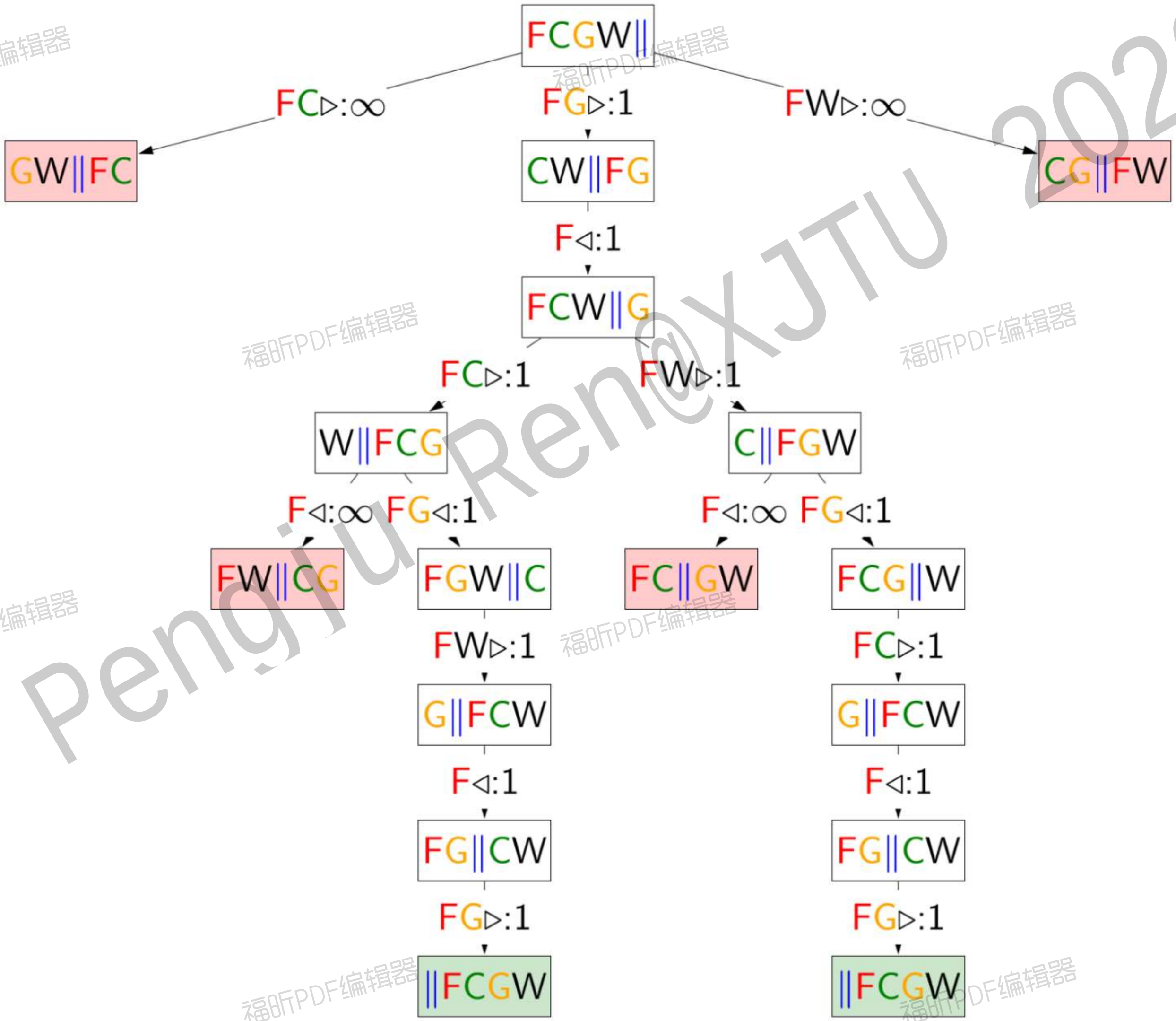
[d] if b is finite and both direction use complete search like BFS

[e] if all steps costs are identical

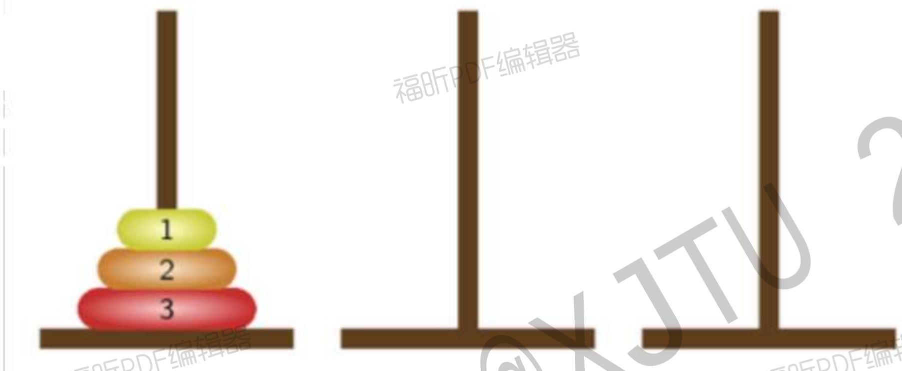
Summary

- Problem formulation usually requires abstracting away real-world details to dene a state space that can feasibly be explored.
  - Initial state.
  - Actions.
  - Transition model.
  - Goal test.
  - Path cost.
- Graph search can be exponentially more efficient than tree search.
- Variety of uninformed search strategies judged on the basis of
  - completeness
  - optimality
  - time and space complexity.
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms.

Question to FCGW



Quiz: Towers of Hanoi



(a) Propose a state representation for the problem ?

Disc 1: (peg, pos) … disc N: (peg,pos)

(b) What is the size of this state space?

3x3x3… = 3^N

(c) What is the start state ?

Disc 1: A, disc N: A

(d) From a given state, what actions are legal ?

-find top disc on each peg

-can only move top disc to another peg if disc is smaller

(e) What is the goal test ?