

# Last chapter...

1

- Agents interact with environments through actuators and sensors
- The performance measure evaluates the environment sequence
- A perfectly rational agent maximizes expected performance
- PEAS descriptions define task environments
- Environments are categorized along several dimensions:  
observable? deterministic? episodic? static? discrete? single-agent?
- Several basic agent architectures exist:  
reflex, reflex with state, goal-based, utility-based

# 2 Solving Problem By Searching

## Chapter 3



# Many AI Tasks can be Formulated as Search Problems

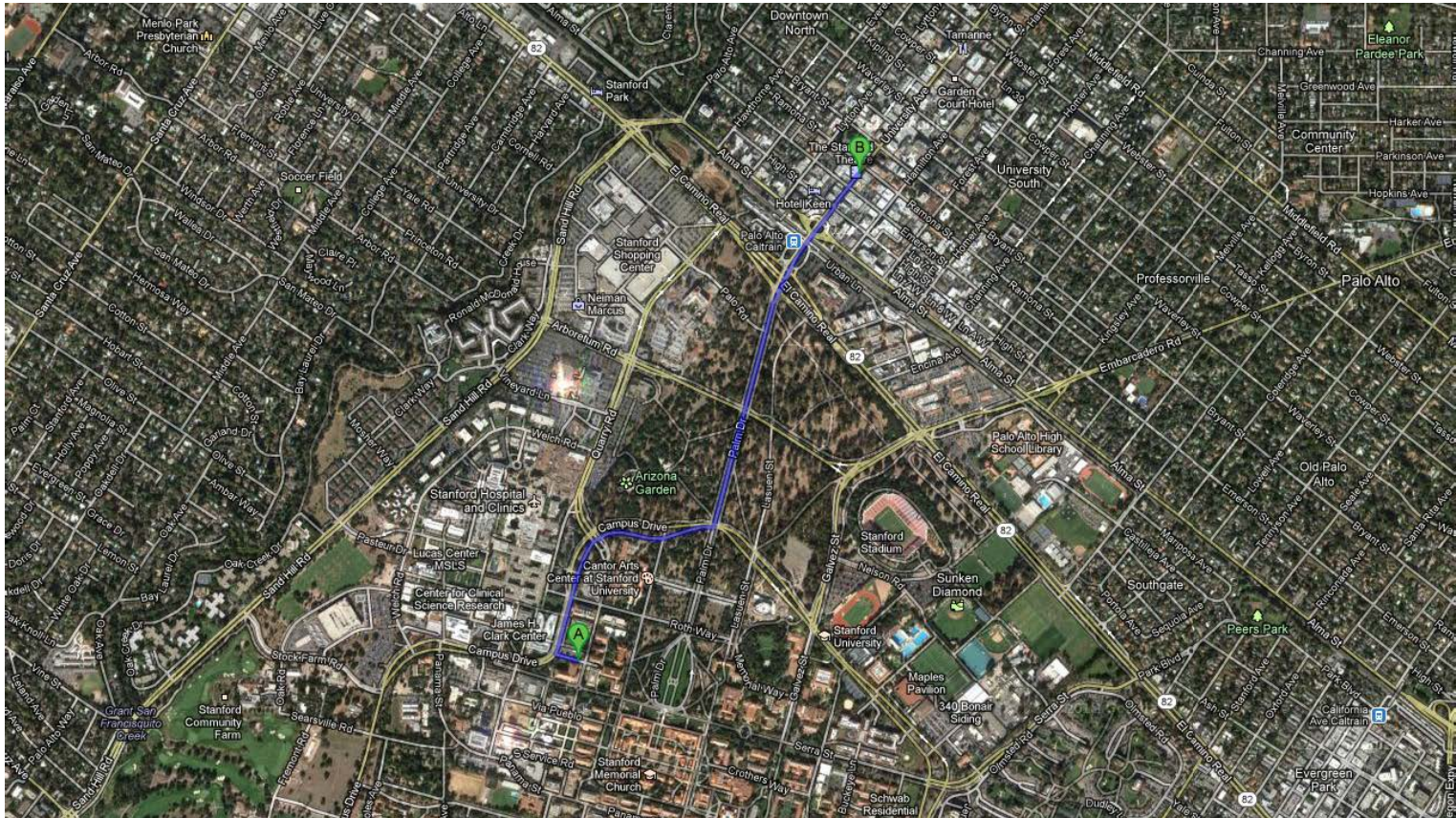
3

- Puzzles
- Games
- Navigation
- Assignment
- Layout
- Scheduling
- Routing



# Search Example: Route Finding

4



<http://staff.ustc.edu.cn/~linlixu/ai2018spring/>

# Search Example: River Crossing

5



## Rules:

- 1) Farmer must row the boat
- 2) Only room for one other
- 3) Without the farmer present:
  - Dog bites sheep
  - Sheep eats cabbage

# Search Example: Solving Puzzles

6



# Search Example: Water Jugs

7

**Given 4-liter and 3-liter pitchers, how do you get exactly 2 liters into the 4-liter pitcher?**





# Outline

8

- Problem-solving agents/问题求解智能体
  - ▣ Goal-based agents
- Problem formulation
- Example problems
- Basic search algorithms



# Problem-solving agents

9

```
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  $\leftarrow$  UPDATE-STATE(state, percept)
  if seq is empty then do
    goal  $\leftarrow$  FORMULATE-GOAL(state)
    problem  $\leftarrow$  FORMULATE-PROBLEM(state, goal)
    seq  $\leftarrow$  SEARCH(problem)
  action  $\leftarrow$  FIRST(seq)
  seq  $\leftarrow$  REST(seq)
  return action
```

Note: this is **offline** problem solving; solution executed “eyes closed.”

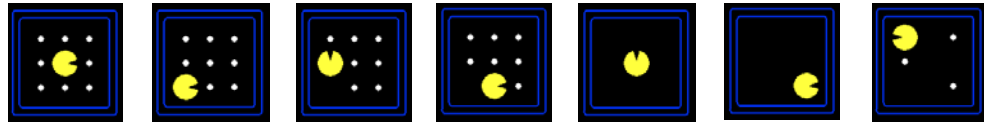
**Online** problem solving involves acting without complete knowledge.

# Search Problems

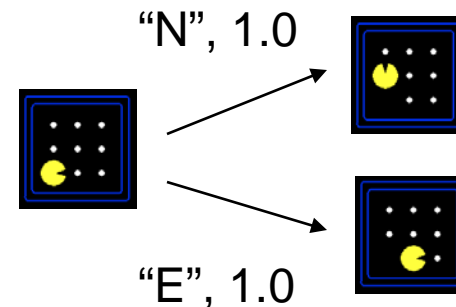
10

- A search problem consists of:

- A state space



- A successor function

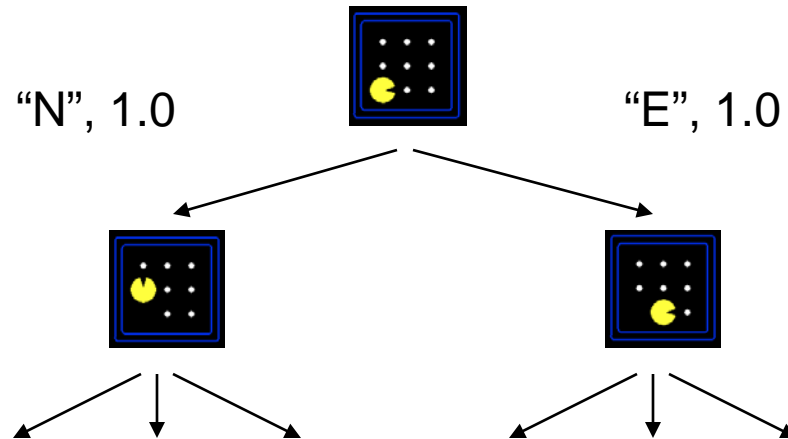


- A start state and a goal test

- A solution is a sequence of actions (a plan) which transforms the start state to a goal state

# Search Trees

11

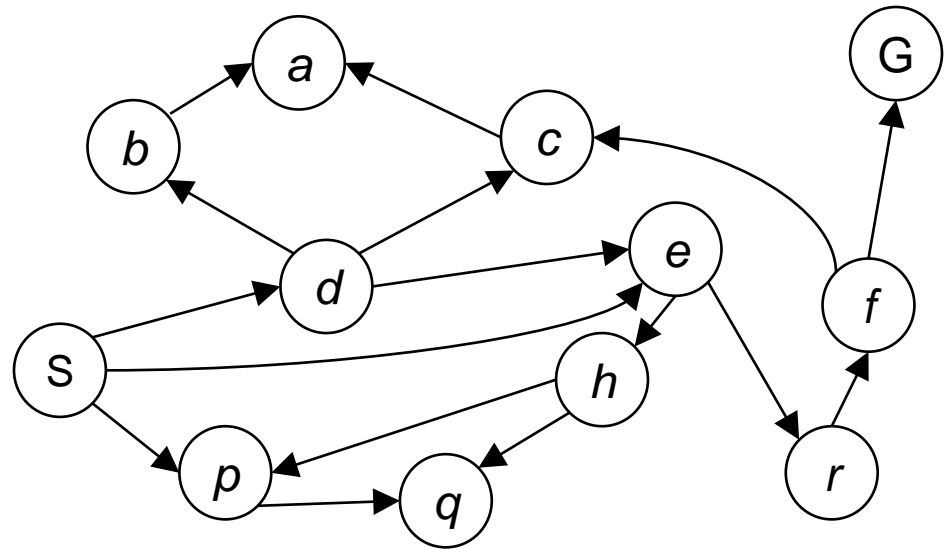


- A search tree:
  - ▣ This is a “what if” tree of plans and outcomes
  - ▣ Start state at the root node
  - ▣ Children correspond to successors
  - ▣ Nodes labeled with states, correspond to PLANS to those states
  - ▣ For most problems, can never actually build the whole tree
    - So, have to find ways of using only the important parts of the tree!

# State Space Graphs

12

- There's some big graph in which
  - ▣ Each state is a node
  - ▣ Each successor is an outgoing arc
- Important: For most problems we could never actually build this graph



*Laughably tiny search graph  
for a tiny search problem*

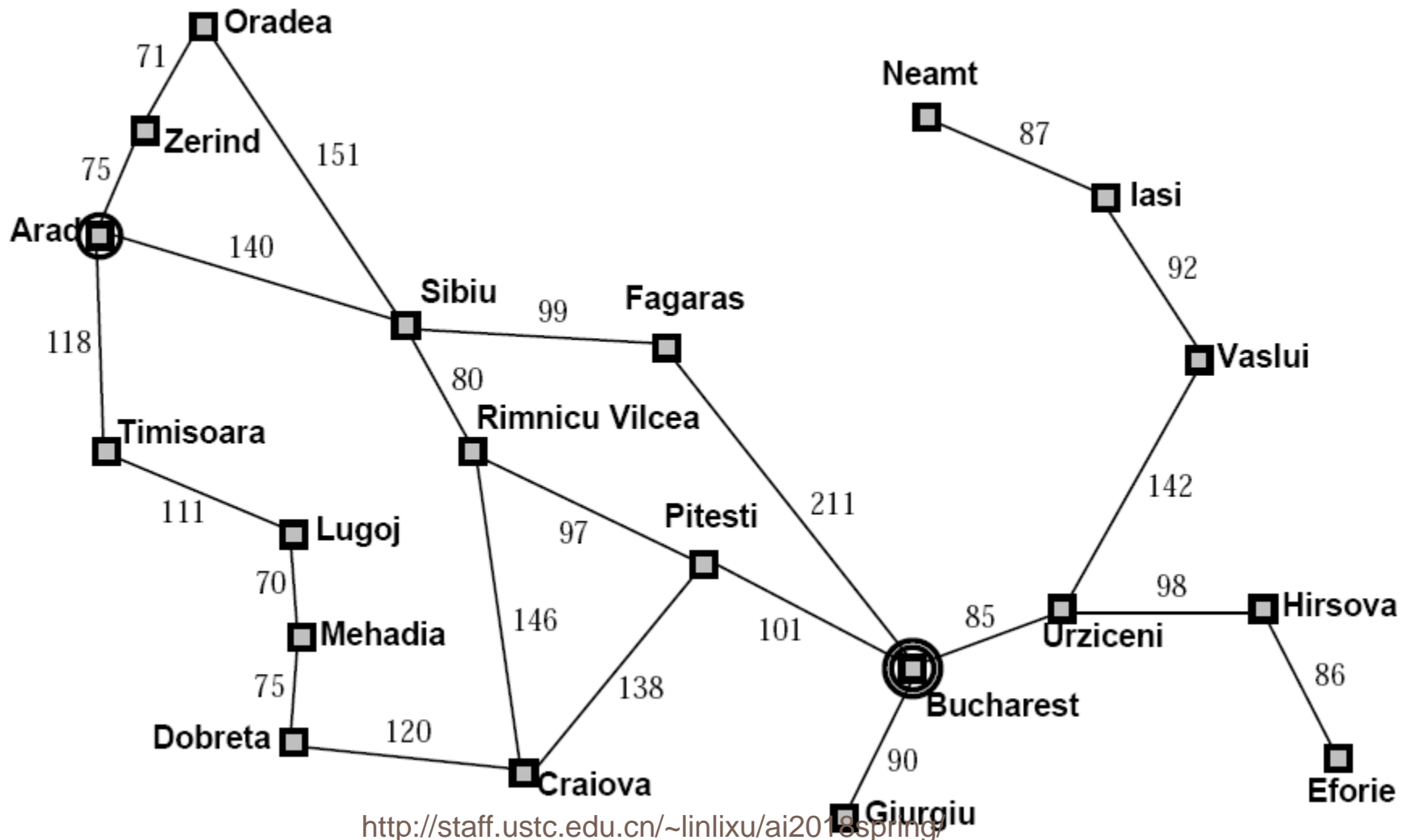
# Example: Romania

13

- On holiday in Romania; currently in Arad.
- Flight leaves tomorrow from Bucharest
  
- **Formulate goal:**
  - ▣ be in Bucharest
  
- **Formulate problem:**
  - ▣ **states** (状态) : various cities
  - ▣ **actions** (行为) : drive between cities
  
- **Find solution:**
  - ▣ sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest

# Example: Romania

14





# Single-state problem formulation

15

A **problem** is defined by four items:

1. **initial state** (初始状态) e.g., “at Arad”
  2. **actions** (行动) or **successor function** (后继函数)  $S(x)$  = set of action–state pairs
    - ▣ e.g.,  $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \dots \}$
  3. **goal test** (目标测试) , can be
    - ▣ **explicit** (明确的) , e.g.,  $x = \text{"at Bucharest"}$
    - ▣ **implicit** (隐含的) , e.g.,  $\text{Checkmate}(x)$
  4. **path cost** (路径损耗) (additive)
    - ▣ e.g., sum of distances, number of actions executed, etc.
    - ▣  $c(x,a,y)$  is the **step cost**, assumed to be  $\geq 0$
- ▣ A **solution** is a sequence of actions leading from the initial state to a goal state

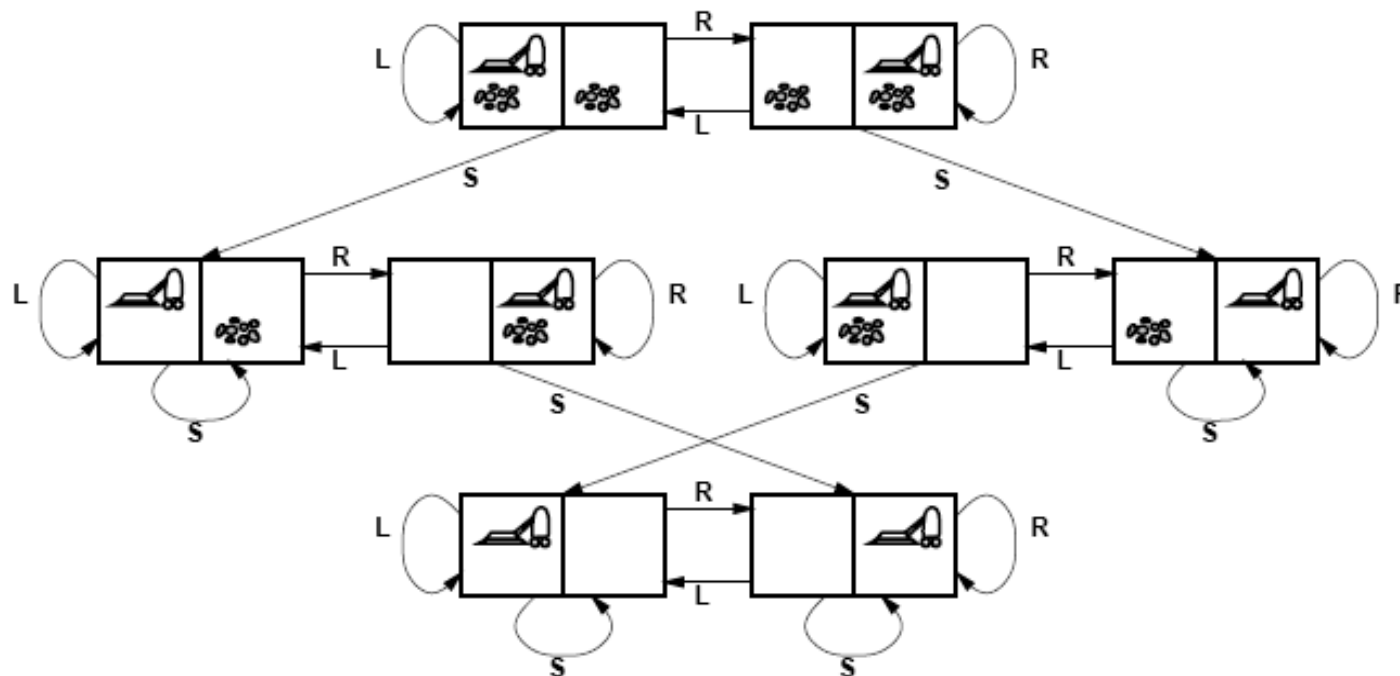
# Selecting a state space

16

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

# Example: vacuum world state space graph

17



states??

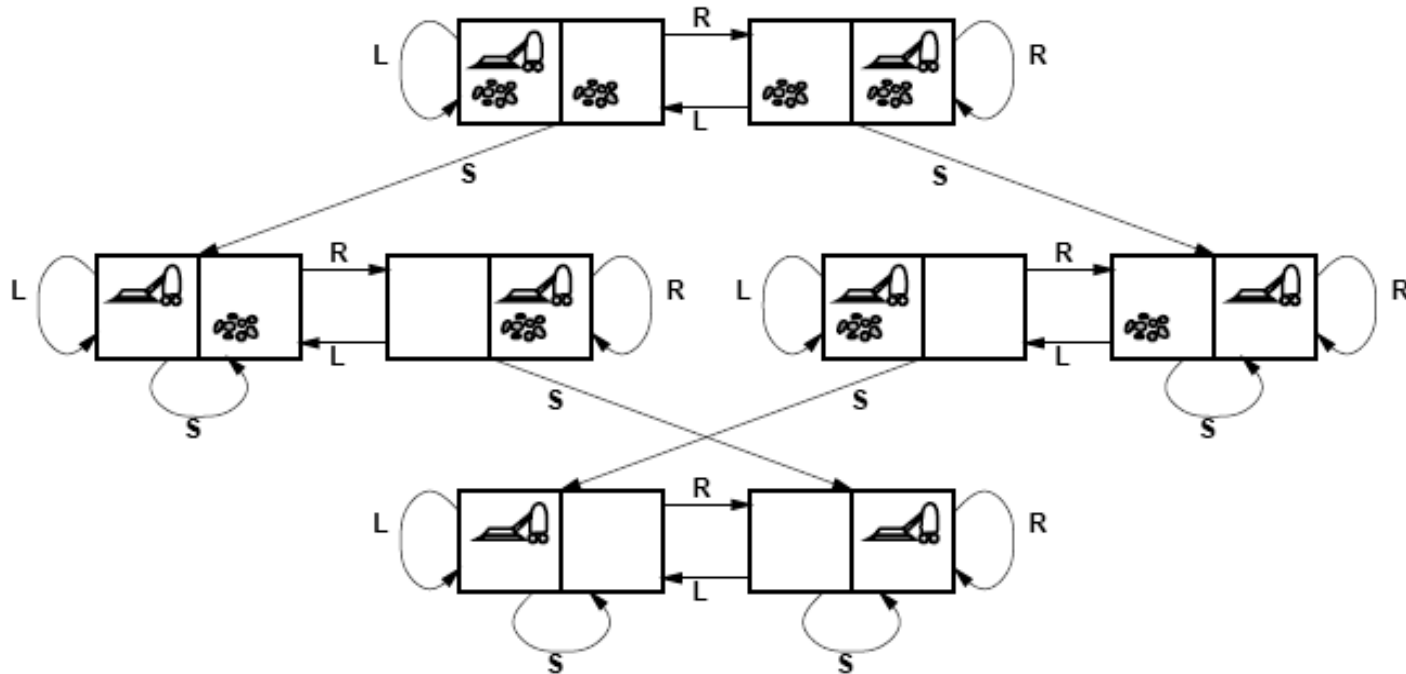
actions??

goal test??

path cost??

# Example: vacuum world state space graph

18



states?? : integer dirt and robot locations (ignore dirt amounts etc.)

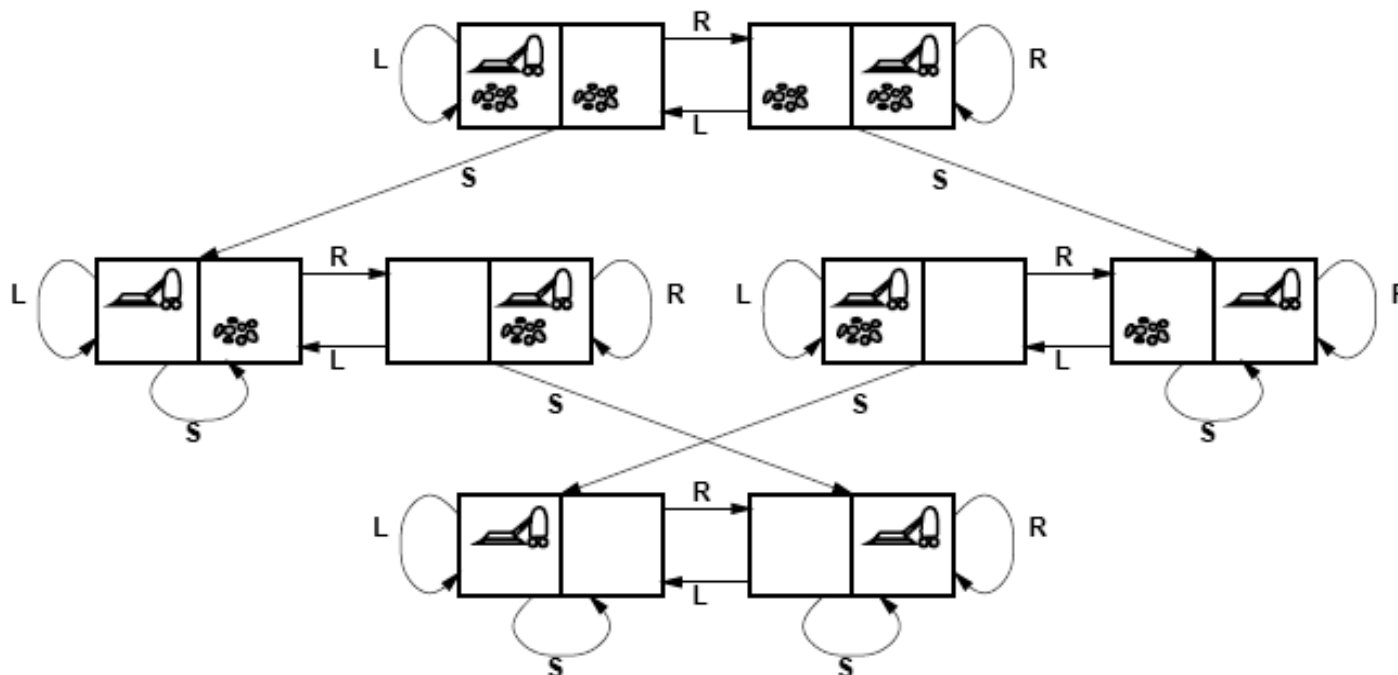
actions??

goal test??

path cost??

# Example: vacuum world state space graph

19



states?? : integer dirt and robot locations (ignore dirt amounts etc.)

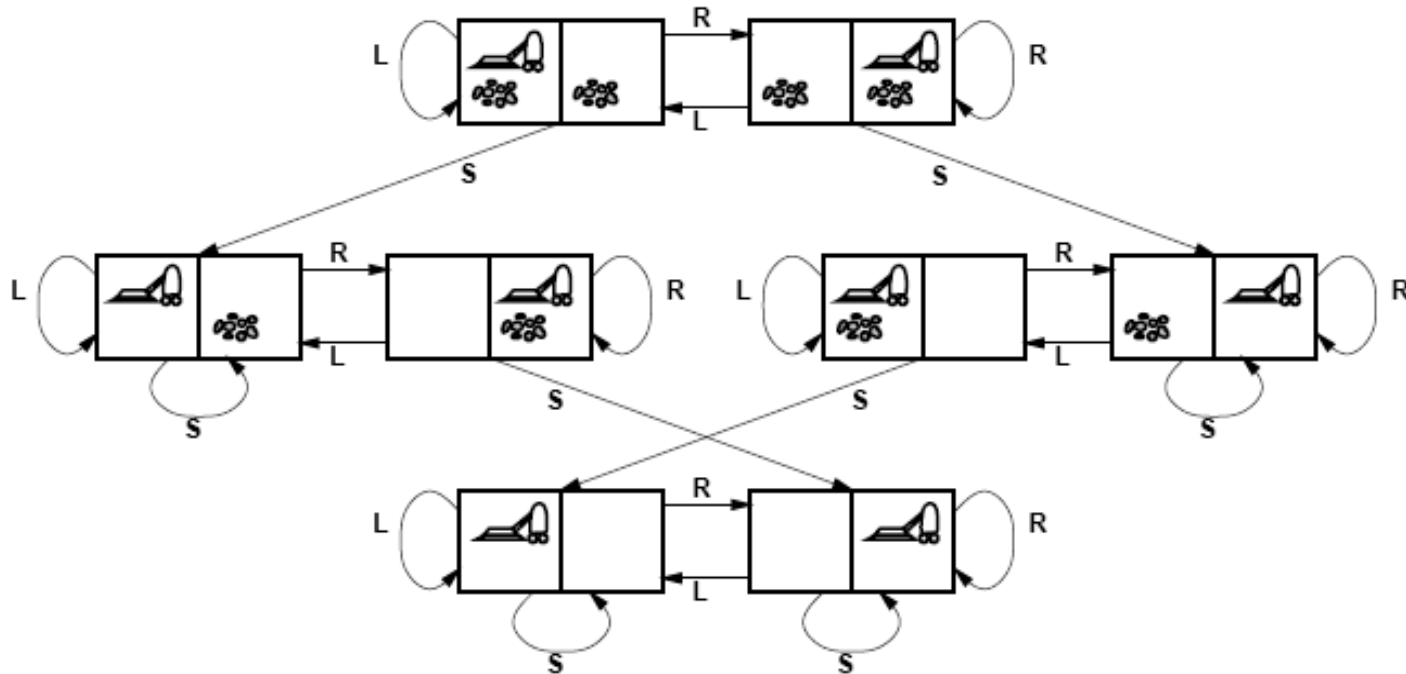
actions?? : *Left* , *Right* , *Suck* , *NoOp*

goal test??

path cost??

# Example: vacuum world state space graph

20



states?? : integer dirt and robot locations (ignore dirt amounts etc.)

actions?? : *Left* , *Right* , *Suck* , *NoOp*

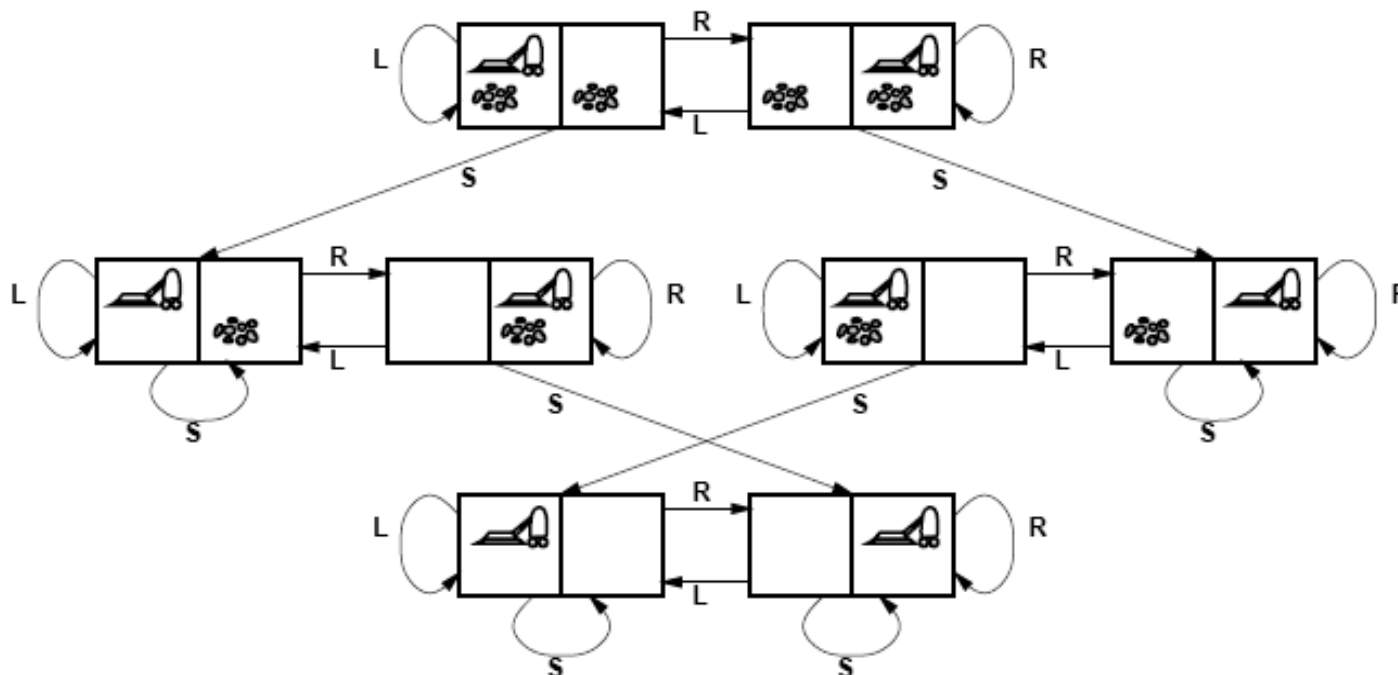
goal test?? : no dirt

path cost??



# Example: vacuum world state space graph

21



states?? : integer dirt and robot locations (ignore dirt amounts etc.)

actions?? : *Left* , *Right* , *Suck* , *NoOp*

goal test?? : no dirt

path cost?? : 1 per action (0 for *NoOp*)

# Example: The 8-puzzle

22

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states??

actions??

goal test??

path cost??

# Example: The 8-puzzle

23

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states?? : integer locations of tiles (ignore intermediate positions)

actions??

goal test??

path cost??

# Example: The 8-puzzle

24

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states?? : integer locations of tiles (ignore intermediate positions)

actions?? : move blank left, right, up, down (ignore unjamming etc.)

goal test??

path cost??

# Example: The 8-puzzle

25

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states?? : integer locations of tiles (ignore intermediate positions)

actions?? : move blank left, right, up, down (ignore unjamming etc.)

goal test?? : = goal state (given)

path cost??

# Example: The 8-puzzle

26

7	2	4
5		6
8	3	1

Start State

1	2	3
4	5	6
7	8	

Goal State

states?? : integer locations of tiles (ignore intermediate positions)

actions?? : move blank left, right, up, down (ignore unjamming etc.)

goal test?? : = goal state (given)

path cost?? 1 per move

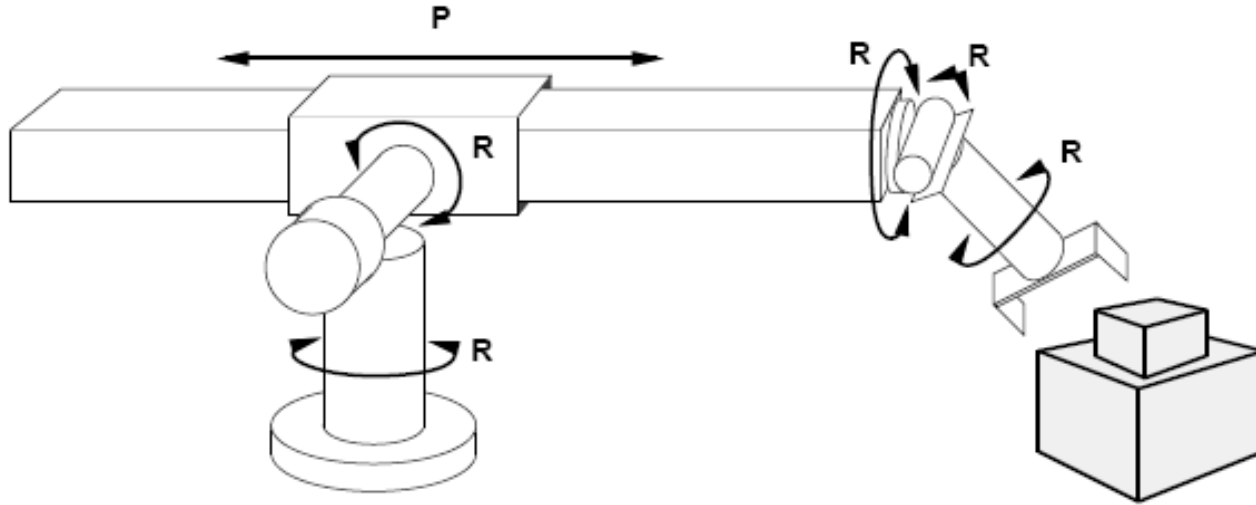
[Note: optimal solution of *n*-Puzzle family is NP-hard]

<http://staff.ustc.edu.cn/~lin/ixu/ai2018spring/>



# Example: robotic assembly

27



states?? : real-valued coordinates (坐标) of robot joint angles

parts of the object to be assembled

actions?? : continuous motions of robot joints

goal test?? : complete assembly **with no robot included!**

path cost?? : time to execute

# Outline

28

- Problem-solving agents/问题求解智能体
  - ▣ Goal-based agents
- Problem formulation
- Example problems
- Basic search algorithms

# Tree search algorithms — 搜索树

29

Basic idea:

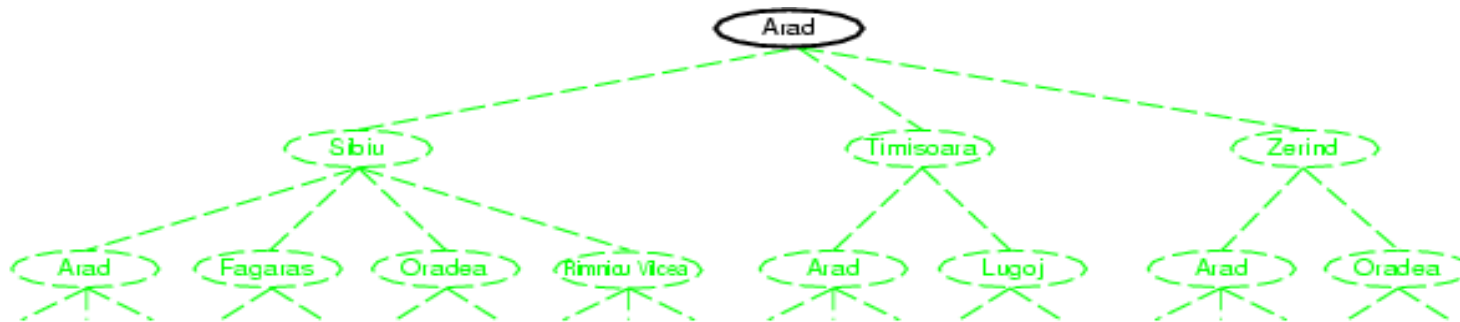
offline, simulated exploration of state space  
by generating successors of already-explored states  
(a.k.a. **expanding** states)

```
function Tree-Search (problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    (根据不同策略选择扩展节点)
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
  end
```

**Strategy** employed determines the type of tree search performed

# Tree search example

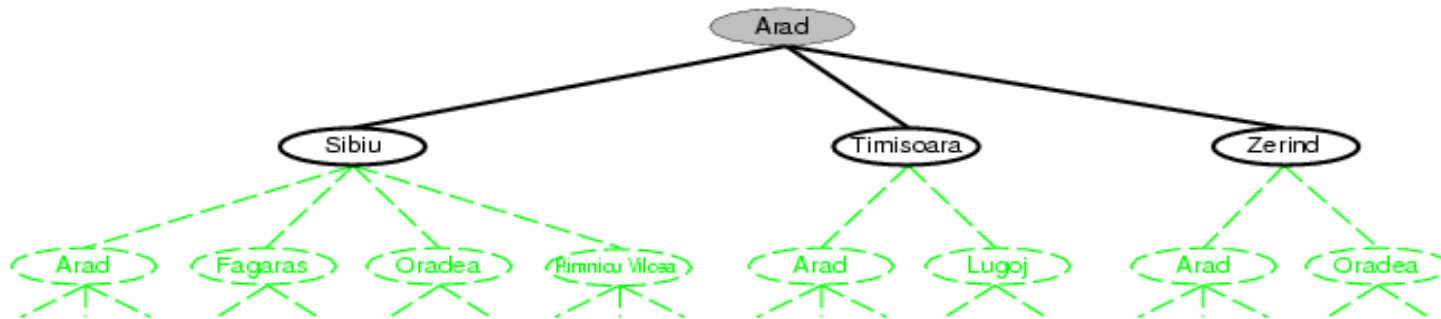
30



```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

# Tree search example

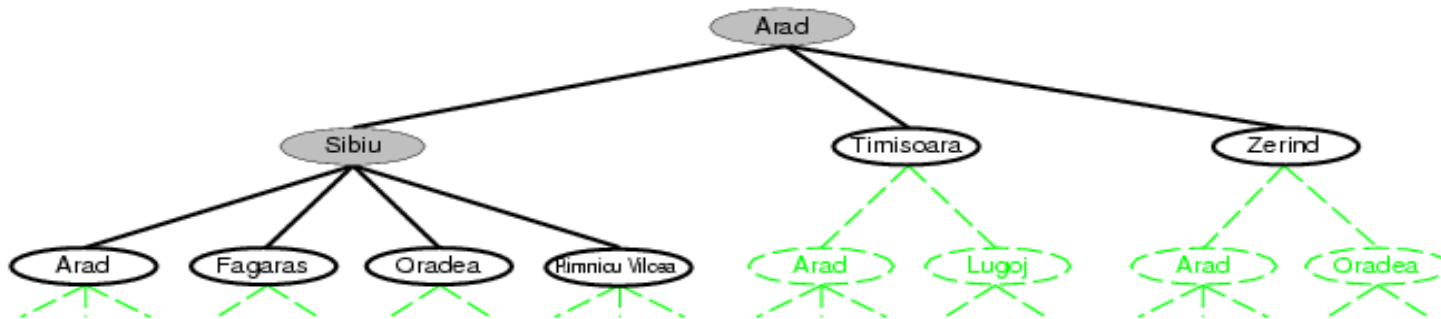
31



```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```

# Tree search example

32



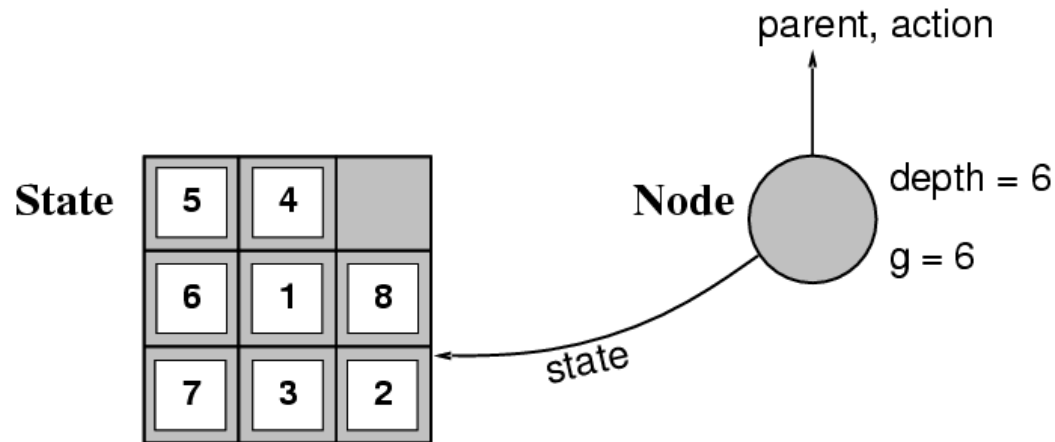
```
function TREE-SEARCH(problem, strategy) returns a solution, or failure
  initialize the search tree using the initial state of problem
  loop do
    if there are no candidates for expansion then return failure
    choose a leaf node for expansion according to strategy
    if the node contains a goal state then return the corresponding solution
    else expand the node and add the resulting nodes to the search tree
```



# Implementation: states vs. nodes

33

- A **state** is a (representation of) a physical configuration
- A **node** is a data structure constituting part of a search tree includes **state**, **parent node**, **action**, **path cost**  $g(n)$ , **depth**



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

# Implementation: general tree search

34

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

---

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

# Search strategies

35

- A search strategy is defined by picking the **order of node expansion**
- Strategies are evaluated along the following dimensions:
  - ▣ **completeness** (完备性) : does it always find a solution if one exists?
  - ▣ **time complexity** (时间复杂度) : number of nodes generated
  - ▣ **space complexity** (空间复杂度) : maximum number of nodes in memory
  - ▣ **optimality** (最优性) : does it always find a least-cost solution?
- Time and space complexity are measured in terms of
  - ▣  $b$ : maximum branching factor of the search tree — 分支因子
  - ▣  $d$ : depth of the least-cost solution — 最浅的目标节点的深度
  - ▣  $m$ : maximum depth of the state space (may be  $\infty$ ) — 最大深度

# Uninformed (无信息的) search strategies

36

**Uninformed** search strategies use only the information available in the problem definition

- Breadth-first search (广度优先搜索)
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

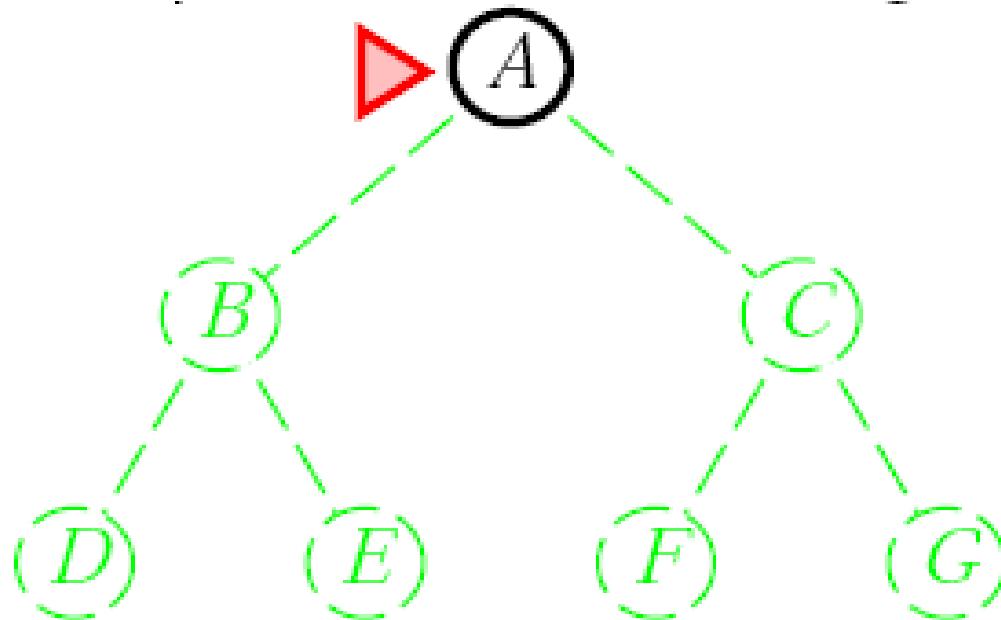
# Breadth-first search

37

- Expand shallowest unexpanded node

- **Implementation:**

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



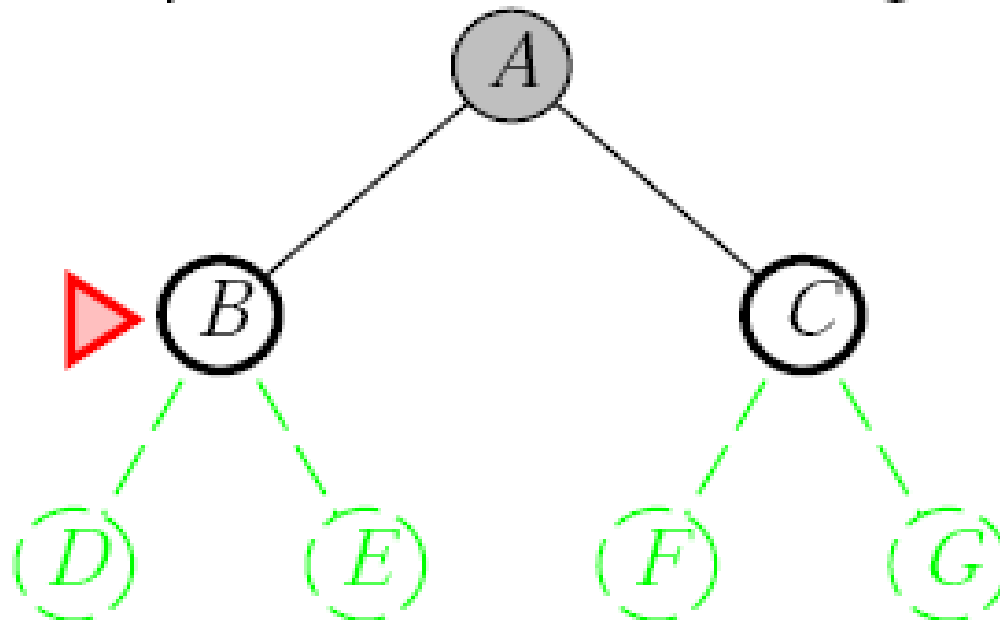
# Breadth-first search

38

- Expand shallowest unexpanded node

- **Implementation:**

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



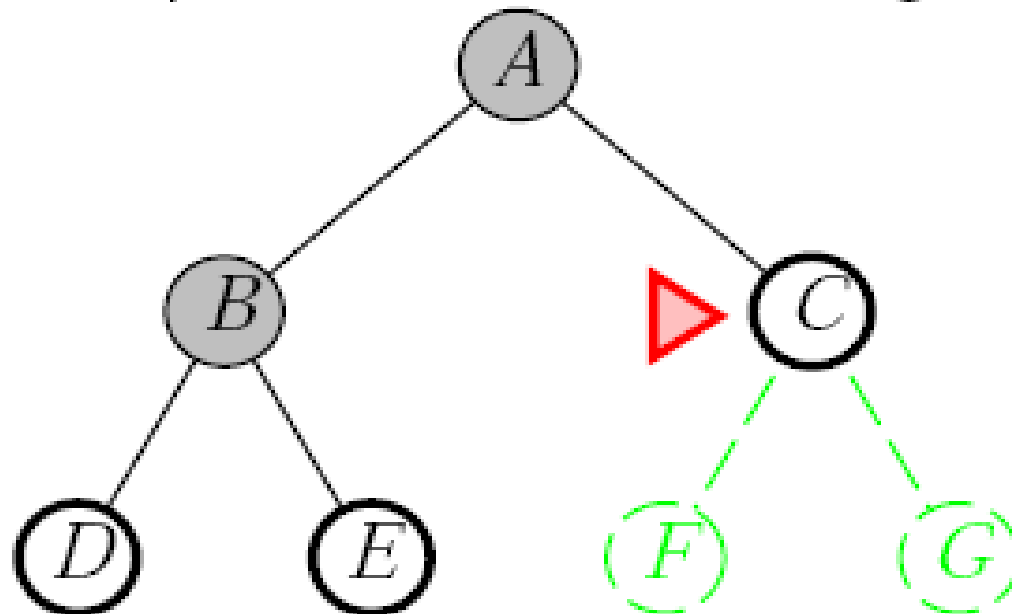
# Breadth-first search

39

- Expand shallowest unexpanded node

- **Implementation:**

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



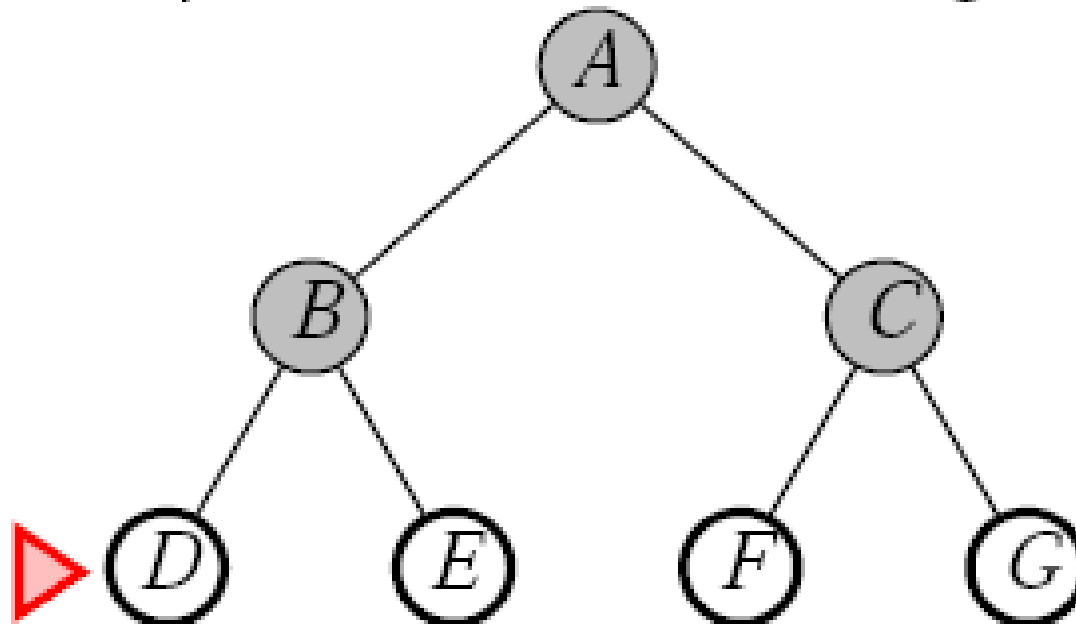
# Breadth-first search

40

- Expand shallowest unexpanded node

- **Implementation:**

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





# Properties of breadth-first search

41

- Complete??

# Properties of breadth-first search

42

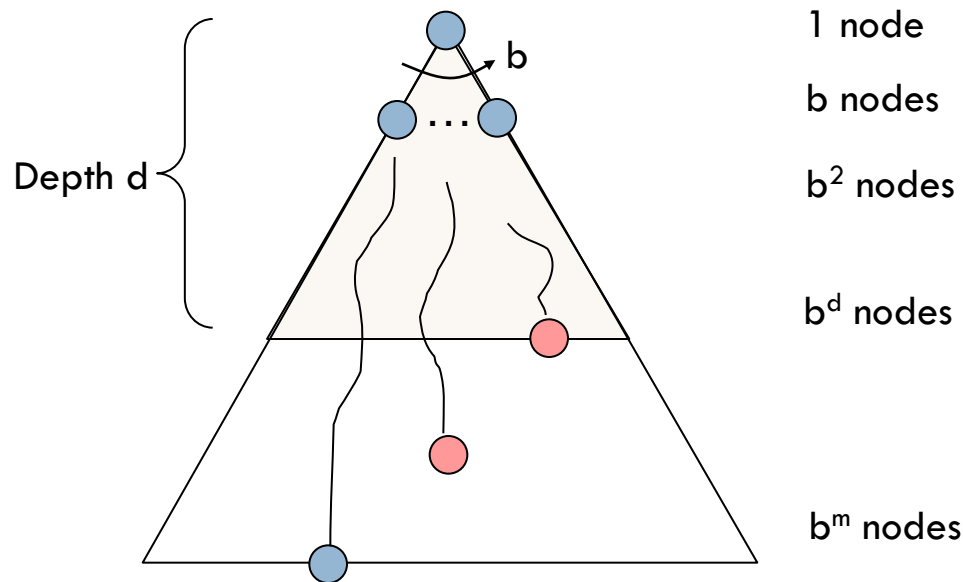
- Complete?? Yes (if  $d$  is finite)
- Time??

b: Branching factor  
d: Solution depth  
m: Maximum depth

# Properties of breadth-first search

43

- Complete?? Yes (if  $d$  is finite)
- Time??  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$ , i.e., exp. in  $d$
- Space??

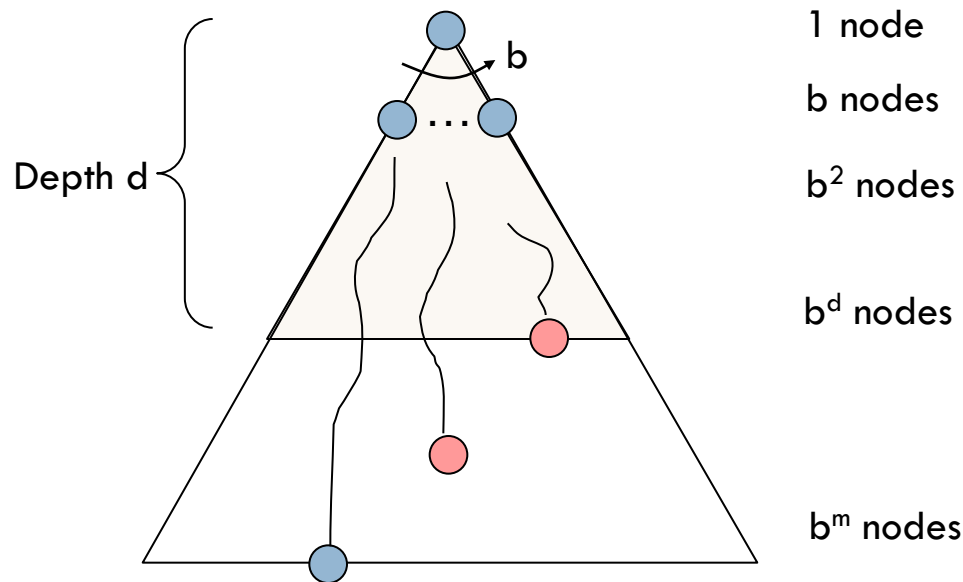


$b$ : Branching factor  
 $d$ : Solution depth  
 $m$ : Maximum depth

# Properties of breadth-first search

44

- Complete?? Yes (if  $d$  is finite)
- Time??  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$ , i.e., exp. in  $d$
- Space??  $O(b^{d+1})$  (keeps every node in memory)
- Optimal??



$b$ : Branching factor  
 $d$ : Solution depth  
 $m$ : Maximum depth

# Properties of breadth-first search

45

- Complete?? Yes (if  $d$  is finite)
- Time??  $1 + b + b^2 + b^3 + \dots + b^d + b(b^d - 1) = O(b^{d+1})$ , i.e., exp. in  $d$
- Space??  $O(b^{d+1})$  (keeps every node in memory)
- Optimal?? Yes (if cost = 1 per step); not optimal in general
- **Space** is the big problem; can easily generate nodes at 100MB/sec  
so 24hrs = 8640GB.

b: Branching factor  
d: Solution depth  
m: Maximum depth

# Uninformed search strategies

46

**Uninformed** search strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search (代价一致搜索)
- Depth-first search
- Depth-limited search
- Iterative deepening search

# Uniform-cost search

47

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

Complete? Yes, if step cost  $\geq \epsilon$

Time? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\text{ceiling}(C^*/\epsilon)})$   
where  $C^*$  is the cost of the optimal solution

Space? # of nodes with  $g \leq$  cost of optimal solution,  $O(b^{\text{ceiling}(C^*/\epsilon)})$

Optimal? Yes – nodes expanded in increasing order of  $g(n)$

# Uninformed search strategies

48

**Uninformed** search strategies use only the information available in the problem definition

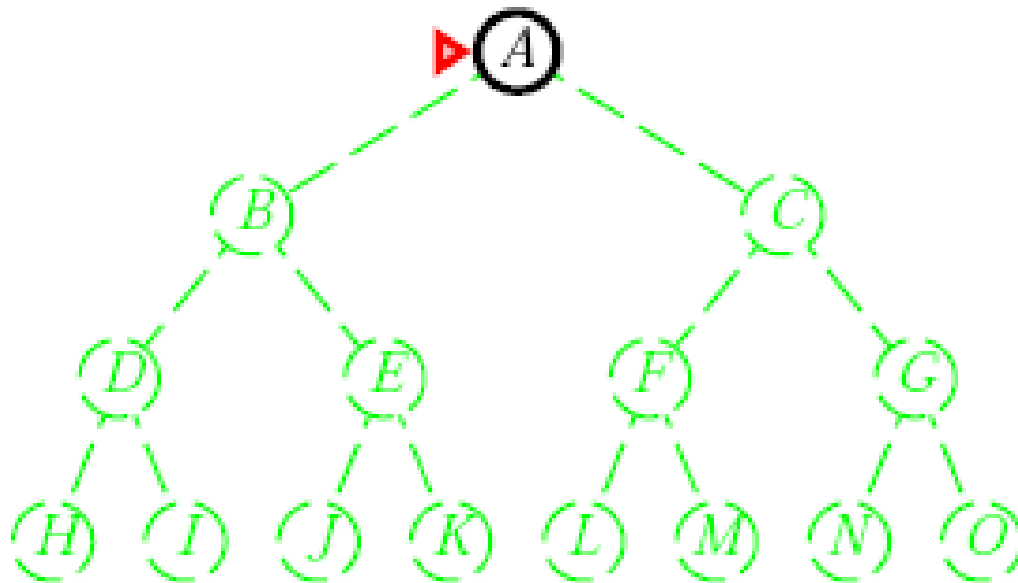
- Breadth-first search
- Uniform-cost search
- Depth-first search (深度优先搜索)
- Depth-limited search
- Iterative deepening search



# Depth-first search

49

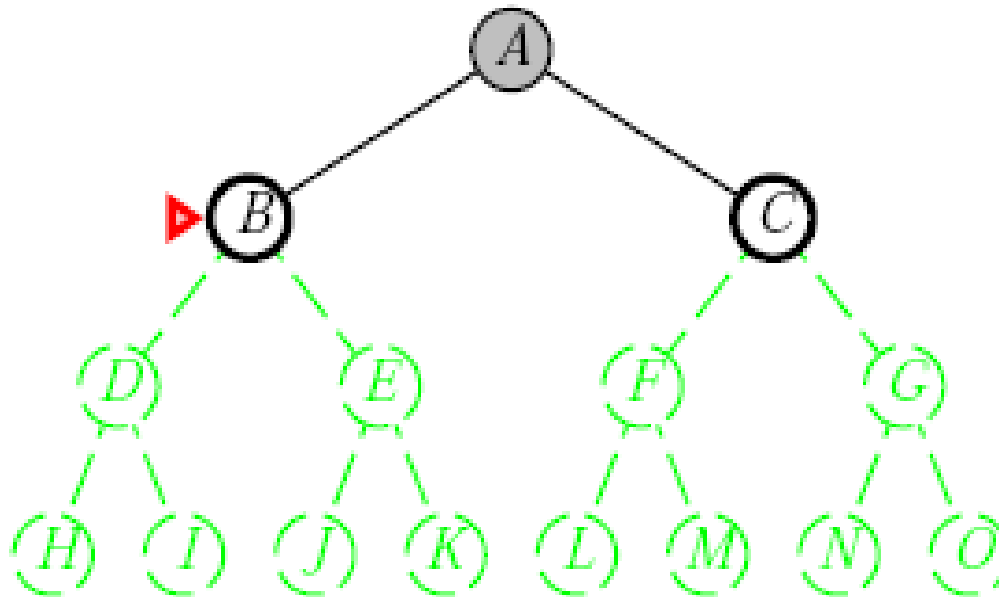
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

50

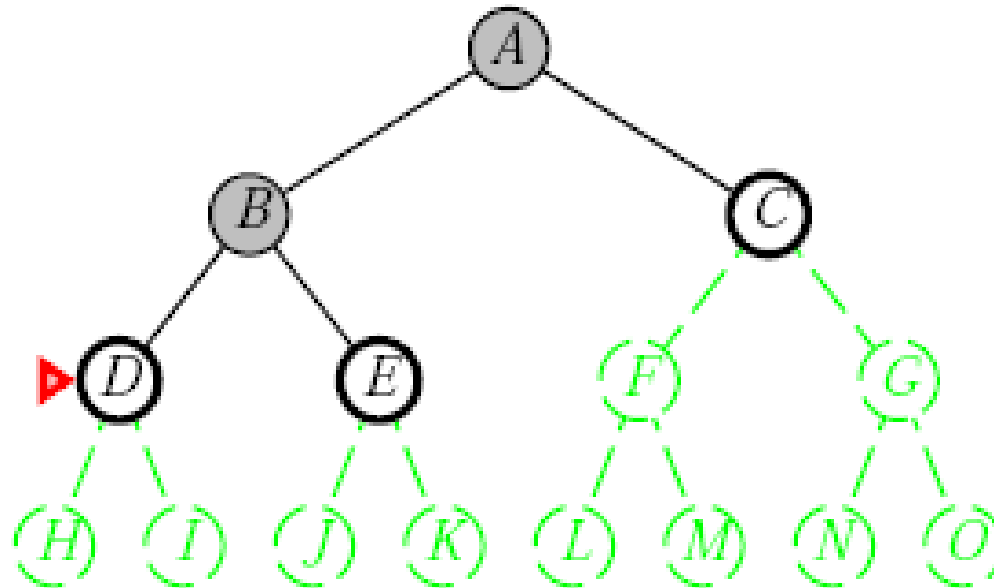
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

51

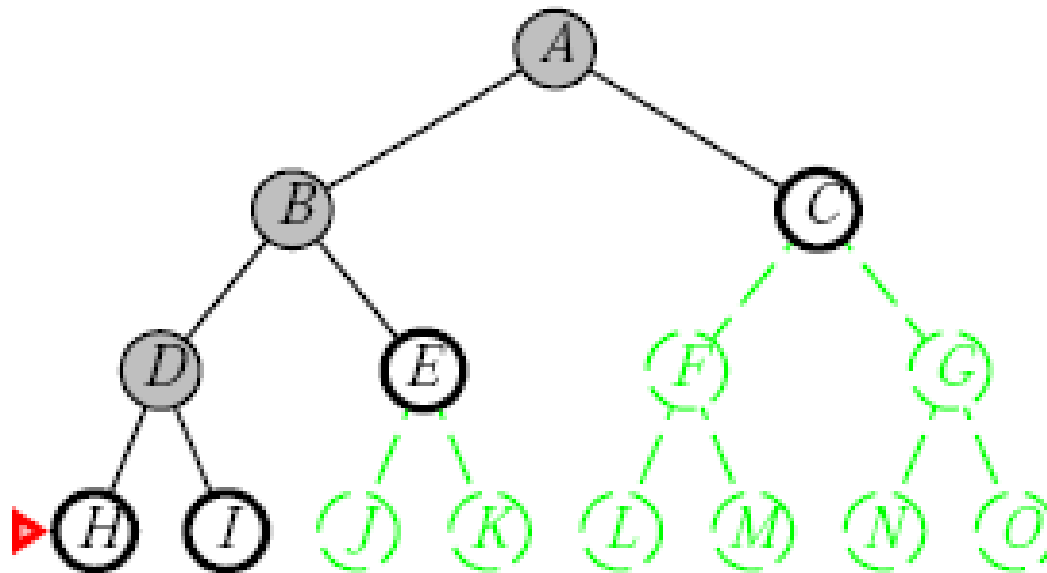
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

52

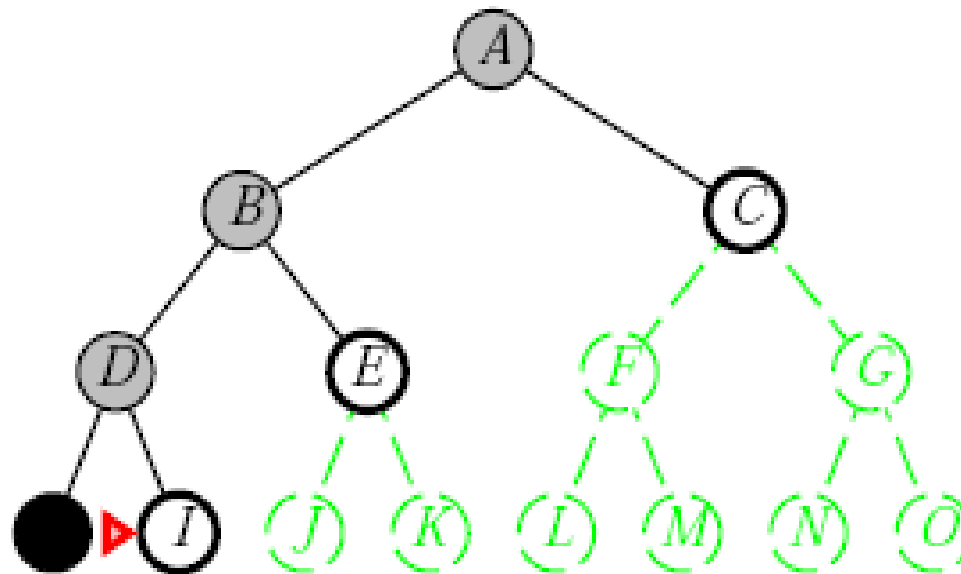
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

53

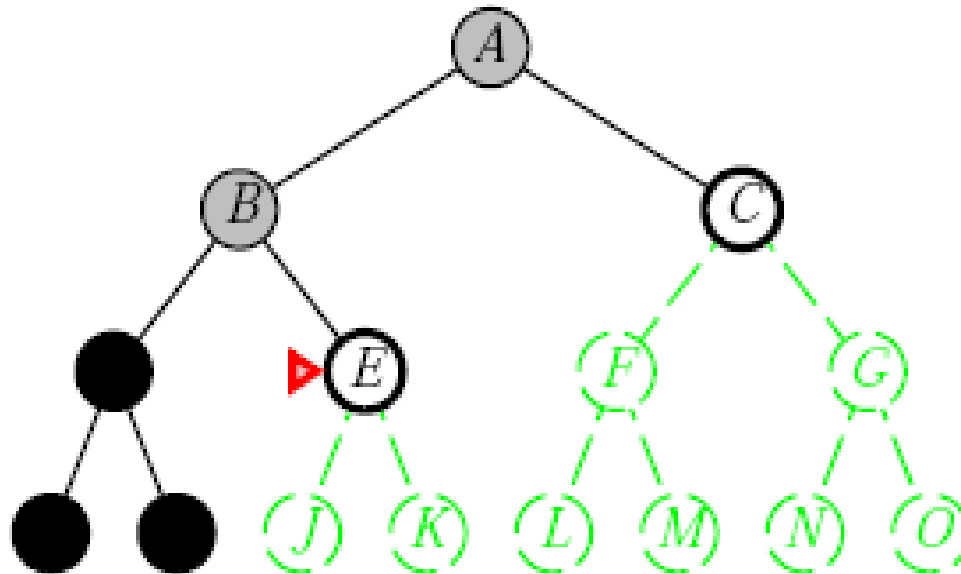
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

54

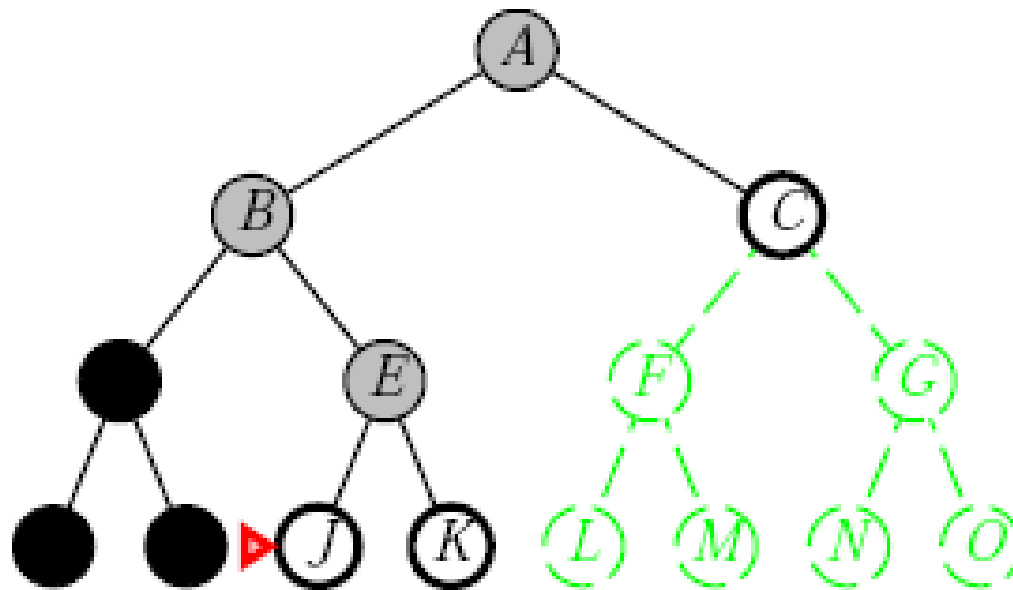
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

55

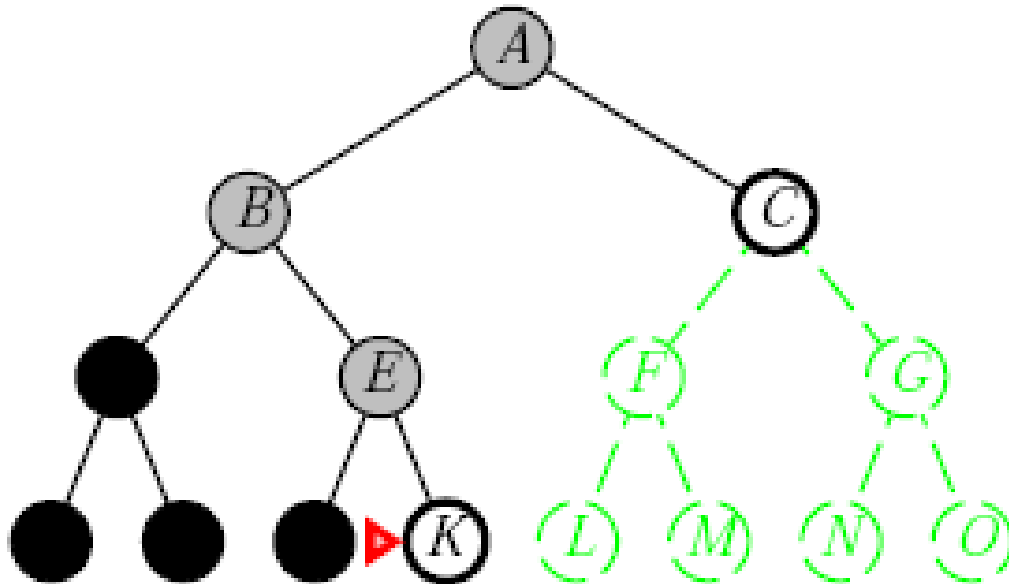
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

56

- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front

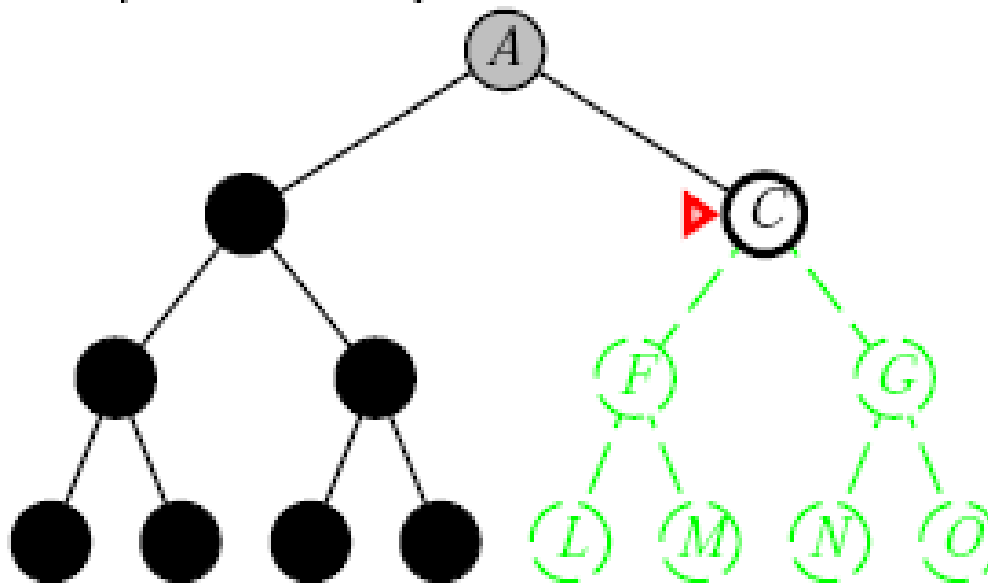




# Depth-first search

57

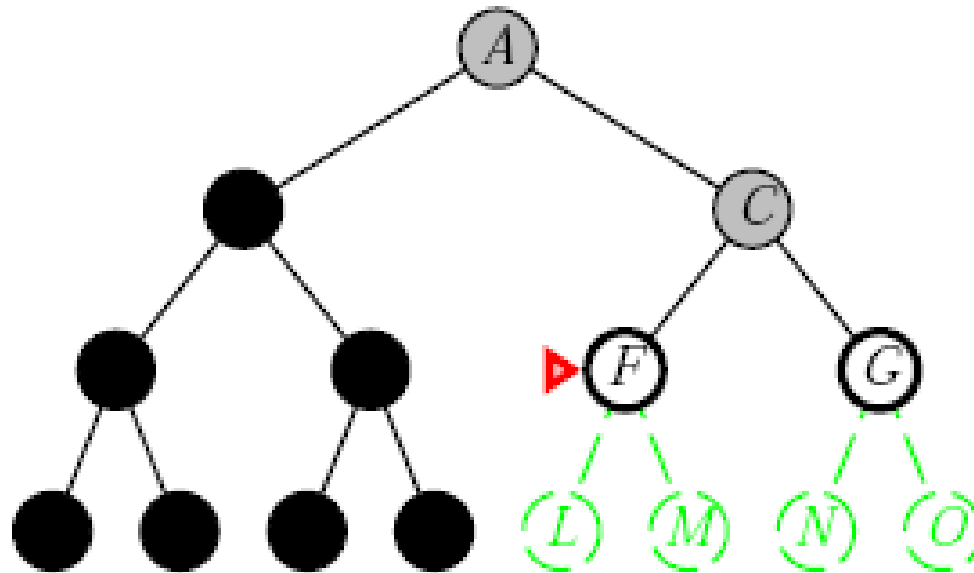
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

58

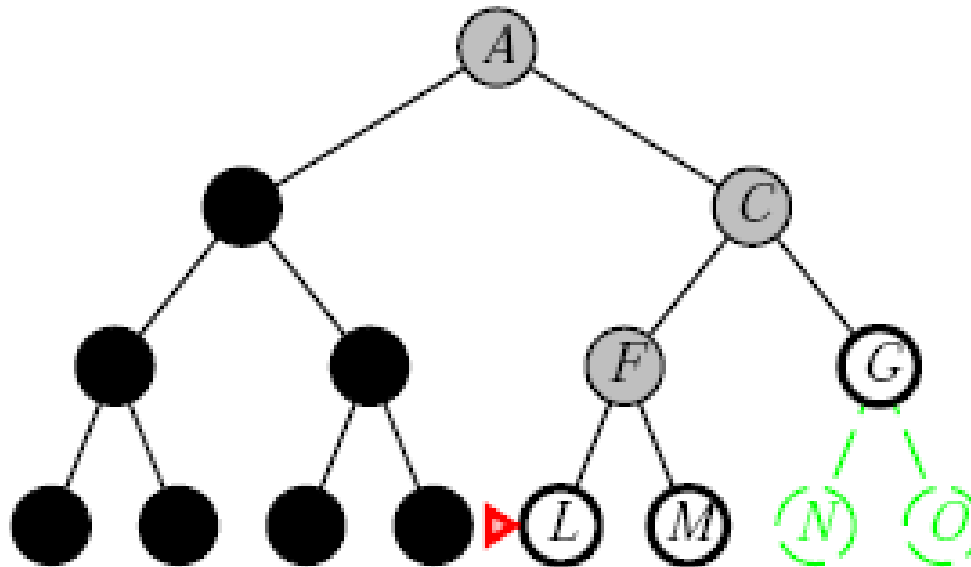
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

59

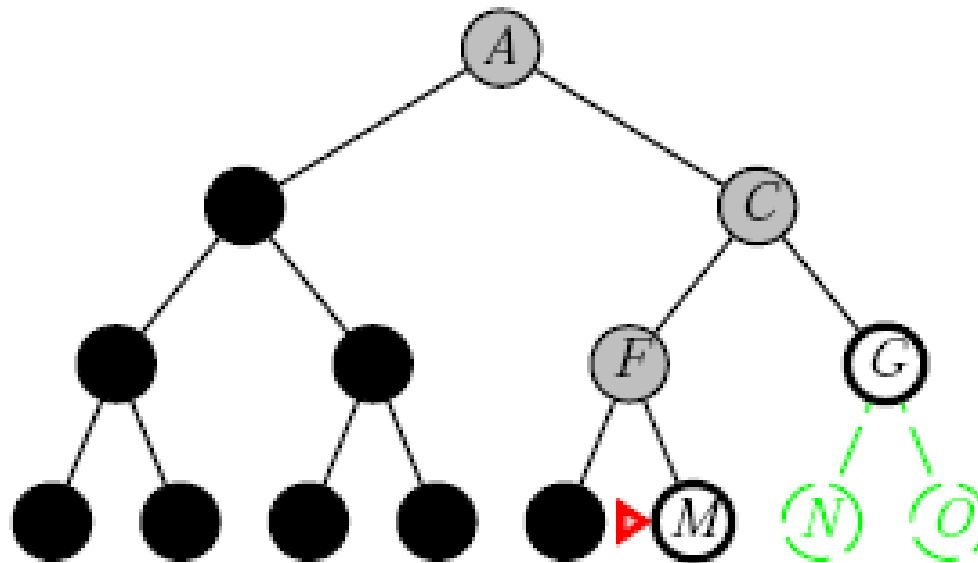
- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Depth-first search

60

- Expand deepest unexpanded node
- **Implementation:**
- *fringe* = LIFO queue, i.e., put successors at front



# Properties of depth-first search

61

- Complete?? No: fails in infinite-depth spaces, spaces with loops

Modify to avoid repeated states along path

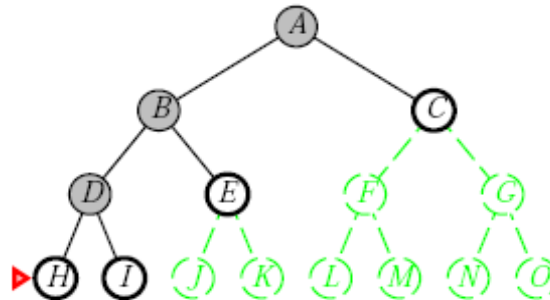
⇒ complete in finite spaces

- Time??  $O(b^m)$ : terrible if  $m$  is much larger than  $d$

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

- Space??  $O(bm)$ , i.e., linear space!

- Optimal?? No



b: Branching factor  
d: Solution depth  
m: Maximum depth

# Uninformed search strategies

62

**Uninformed** search strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search (深度有限搜索)
- Iterative deepening search

# Depth-limited search

63

= depth-first search with depth limit  $l$ ,  
i.e., nodes at depth  $l$  have no successors

- Solves infinite-depth path problem
- if  $l < d$ , possibly incomplete
- If  $l > d$ , not optimal
- Recursive (递归) implementation

# Uninformed search strategies

64

**Uninformed** search strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search (迭代深入深度优先搜索)



# Iterative deepening search

65

由Depth-limited search演化而成  
每轮增加深度限制

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution
  inputs: problem, a problem
  for depth 0 to  $\infty$  do
    result DEPTH-LIMITED-SEARCH( problem, depth)
    if result  $\neq$  cutoff then return result
  end
```

# Iterative deepening search / =0

66

Limit = 0



# Iterative deepening search / = 1

67

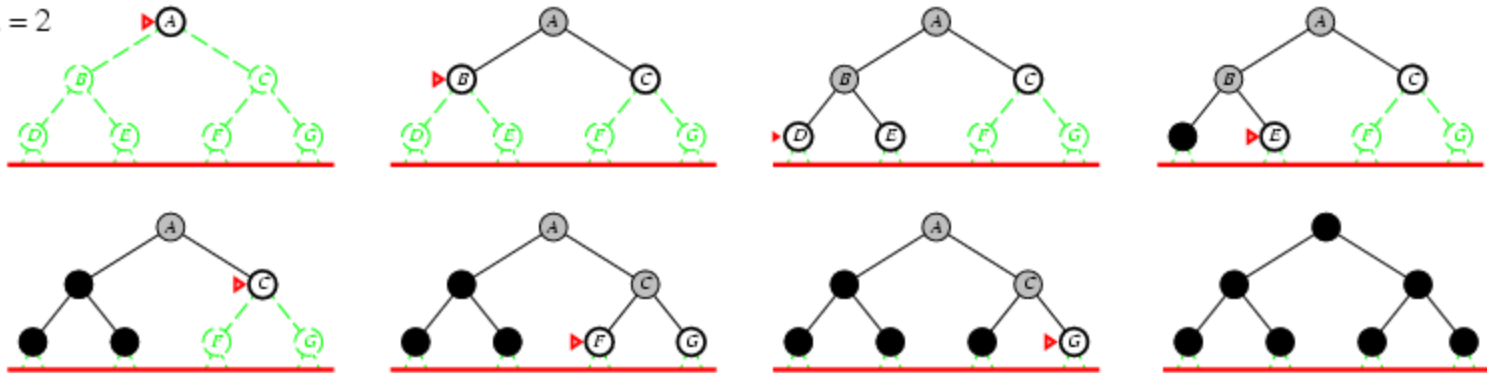
Limit = 1



# Iterative deepening search / =2

68

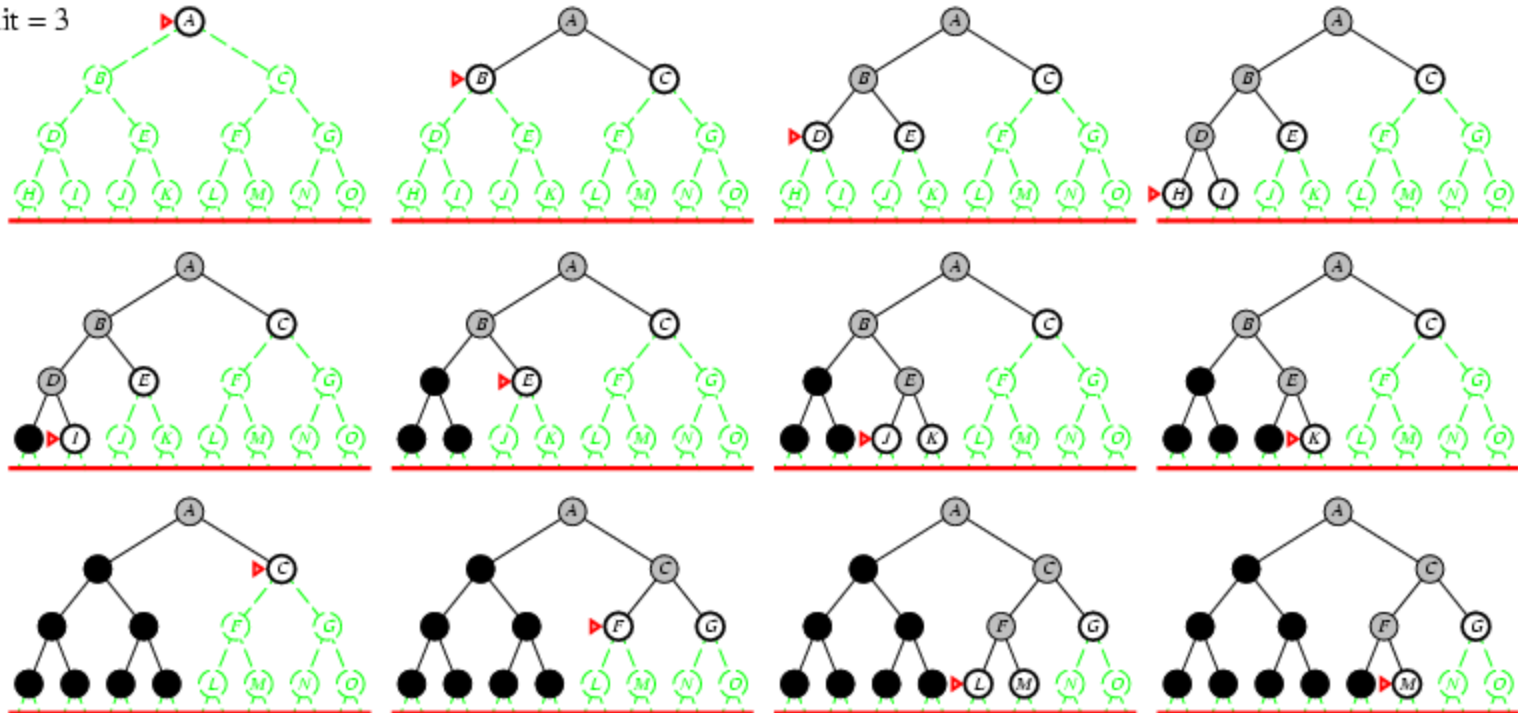
Limit = 2



# Iterative deepening search / =3

69

Limit = 3



# Properties of iterative deepening search

70

- Complete??

# Properties of iterative deepening search

71

- Complete?? Yes
- Time??

# Properties of iterative deepening search

72

- Complete?? Yes
- Time??  $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- Space??



# Properties of iterative deepening search

73

- Complete?? Yes
- Time??  $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- Space??  $O(bd)$
- Optimal??

# Properties of iterative deepening search

74

- Complete?? Yes
- Time??  $(d + 1)b^0 + db^1 + (d - 1)b^2 + \dots + b^d = O(b^d)$
- Space??  $O(bd)$
- Optimal?? Yes, if step cost = 1
- Can be modified to explore uniform-cost tree

Numerical comparison for  $b = 10$  and  $d = 5$ , solution at far right leaf:

- $N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
  - $N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,990 = 1,111,100$
- IDS does better because other nodes at depth  $d$  are not expanded

# Summary of algorithms

75

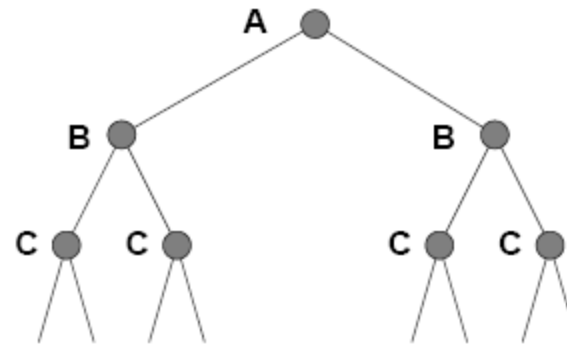
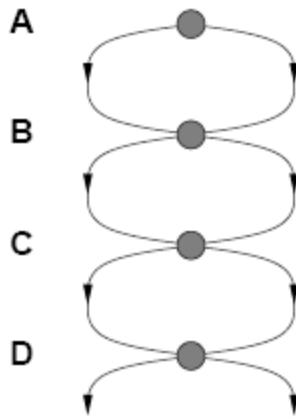
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

b: Branching factor  
d: Solution depth  
m: Maximum depth

# Repeated states

76

- Failure to detect repeated states can turn a linear problem into an exponential one!



# Graph search

77

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed ← an empty set
  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
    if STATE[node] is not in closed then
      add STATE[node] to closed
      fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

# Exercise

78

## Water jugs problem

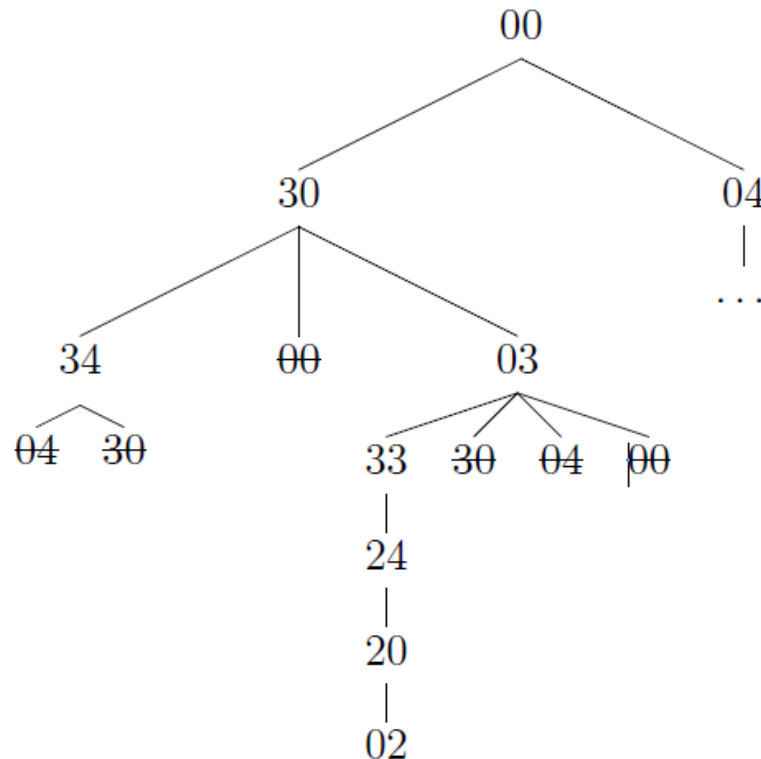
Given a 3 liter jug and a 4 liter jug, where jugs can be filled with water, emptied, or water can be poured from one jug into another until either the source jug is empty or the destination jug is full. Consider a problem where the jugs are initially empty and the goal is to achieve 2 gallons in the 4 gallon jug.



# Exercise

79

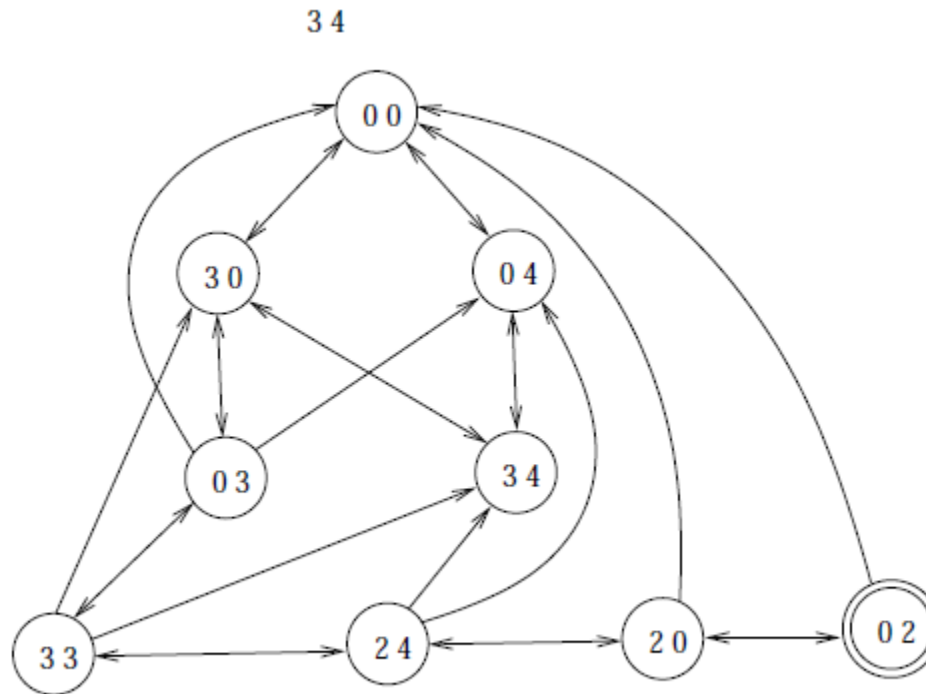
**Solution:** Systematically expand a search tree for the problem to find the solution as follows



# Exercise

80

- In general, the *search graph* for this problem is given as follows





# Summary

81

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms
- Graph search can be exponentially more efficient than tree search

# 作业

82

- 3.7(a,b,d)
- 3.9