Solving Problems by Searching

吉建民

USTC
jianmin@ustc.edu.cn

2024 年 3 月 5 日

Used Materials

Disclaimer: 本课件采用了 S. Russell and P. Norvig's Artificial Intelligence –A modern approach slides, 徐林莉老师课件和其他网络课程课件，也采用了 GitHub 中开源代码，以及部分网络博客内容

回顾

- 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

Table of Contents

Problem-solving Agents

Searching for Solutions

Uninformed Search Strategies

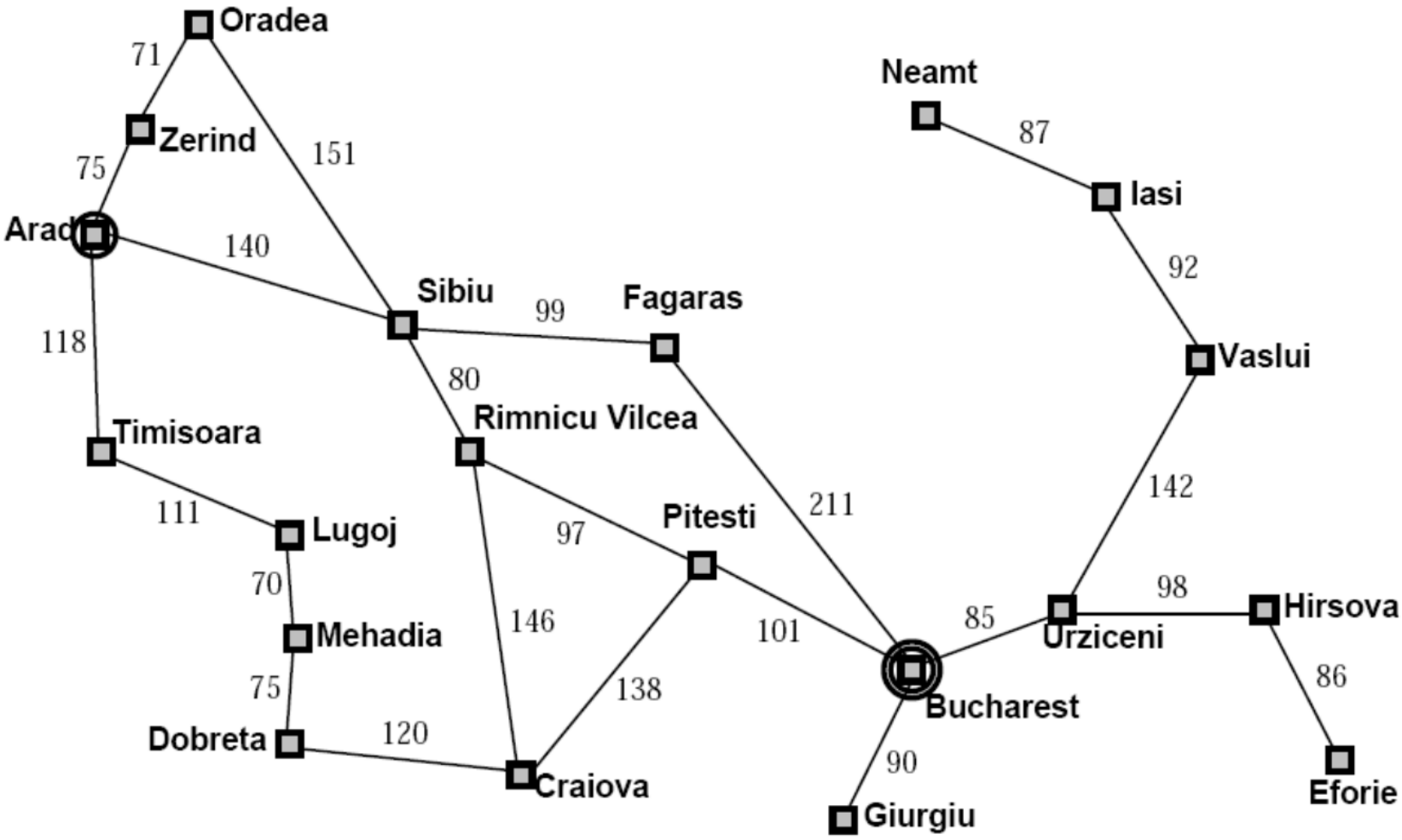
Problem-solving Agents: Agents that Plan Ahead

- Problem-solving agents decide based on evaluating future action sequences
- Search algorithms typically assume
  - Known, deterministic transition model
  - Discrete states and actions
  - Fully observable
  - Atomic representation
    - States of the world are considered as wholes, with no internal structure visible to the problem-solving algorithms
- Usually have a definite goal
- Optimal: Achieve goal at least cost

Problem-solving agents: a kind of goal-based agent using atomic representations

- that use more advanced factored or structured representations are usually called planning agents

从例子开始: Romania



如何找到一条路径，从 Arad 前往 Bucharest ?

从例子开始: Romania

Problem-solving agents: a kind of Goal-based agents

- 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

Problem-solving Agents

```
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 do
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Note: this is offline problem solving; solution executed "eyes closed"

Online problem solving involves acting without complete knowledge.

Problems and Solutions

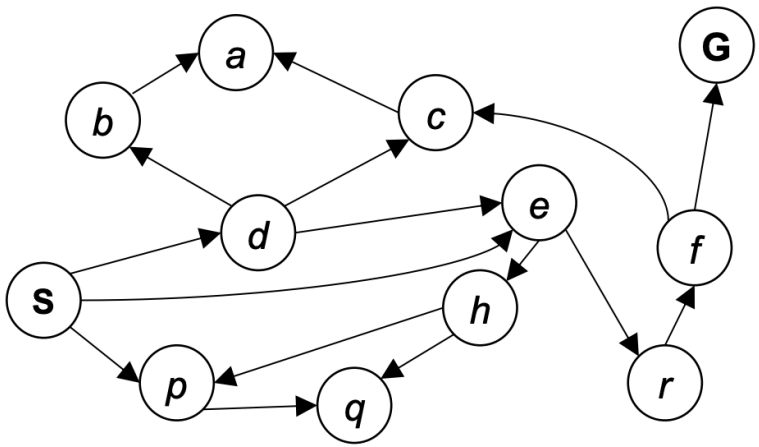
- A (search) problem consists of:
  - An initial state s_0
    - $s_0 = In(Arad)$
  - Actions $A(s)$ in each state
    - $A(In(Arad)) = \{Go(Sibiu),\ Go(Timisoara),\ Go(Zerind)\}$
  - A transition model $Result(s, a)$
    - $Result(In(Arad),\ Go(Zerind)) = In(Zerind)$
  - A state space S
    - S can be implicitly defined by s_0, $A(s)$, and $Result(s, a)$
  - A goal test $G(s)$
    - Explicit, e.g., $G(s) = I(s \in \{In(Bucharest)\})$
    - Implicit, e.g., $G(s) = Checkmate(s)$
  - A path cost function that assigns a numeric cost to each path
    - E.g., sum of distances, number of actions executed, etc.
    - A step cost $c(s, a, s')$, assumed to be ≥ 0
- A solution is a sequence of actions leading from the initial state to a goal state
- An optimal solution has least cost among all solutions

Selecting a State Space

- 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., $Go(Zerind) \in A(Arad)$ 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

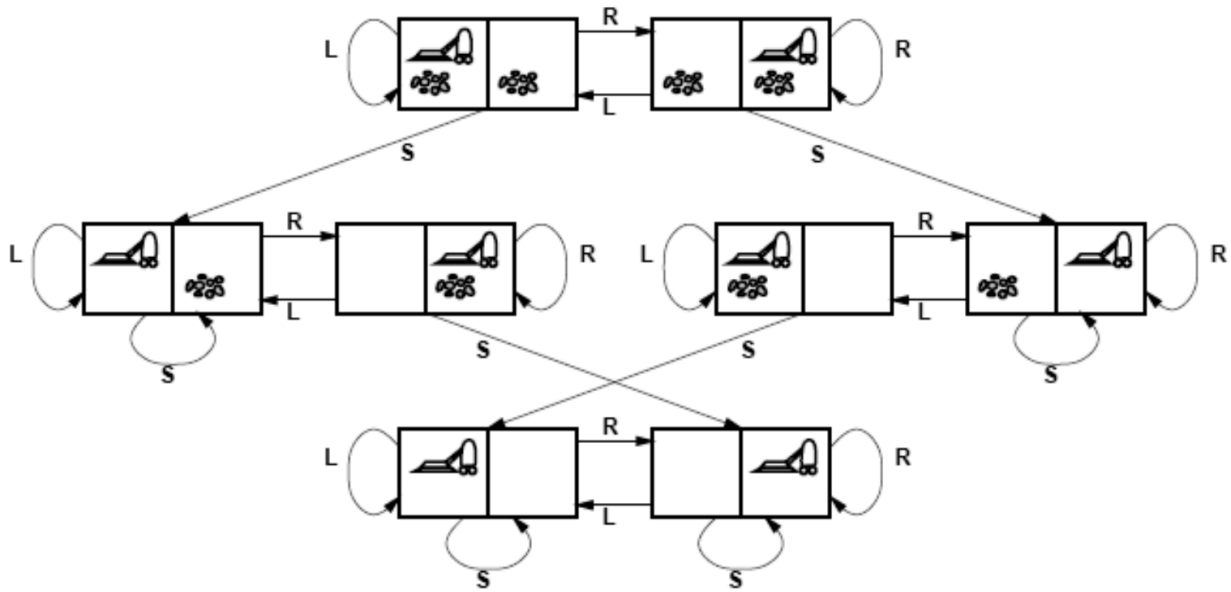
State Space Graphs (状态空间图)

- State space graph: A mathematical representation of a search problem
  - Nodes are (abstracted) world configurations
  - Arcs represent successors (action results)
  - The goal test is a set of goal nodes (maybe only one)
- In a state space graph, each state occurs only once!
- We can rarely build this full graph in memory (it's too big), but it's a useful idea



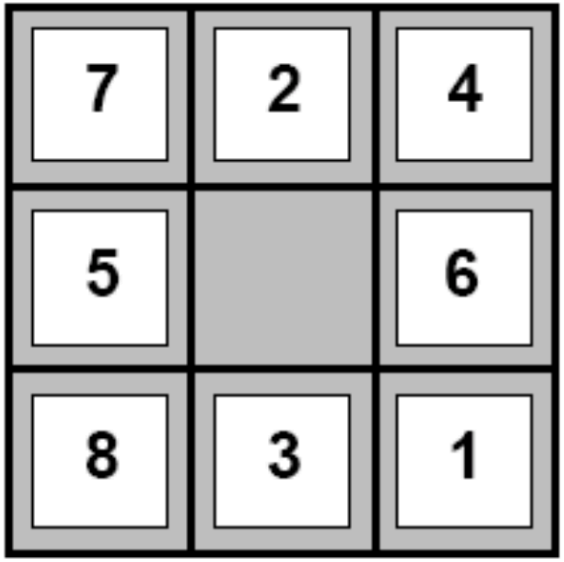
Tiny state space graph for a tiny search problem

Example: Vacuum World State Space Graph

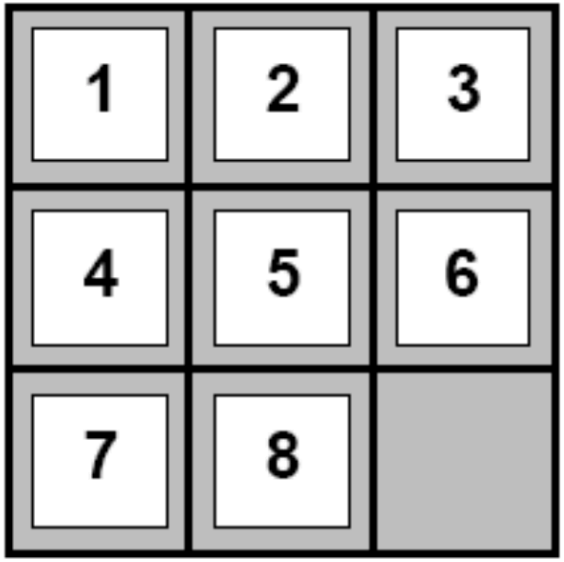


- States: integer dirt and robot locations (ignore dirt amounts etc.)
- Actions: Left, Right, Suck, NoOp
- Goal test: no dirt
- Path cost: 1 per action (0 for NoOp)

Example: The 8-puzzle



Start State

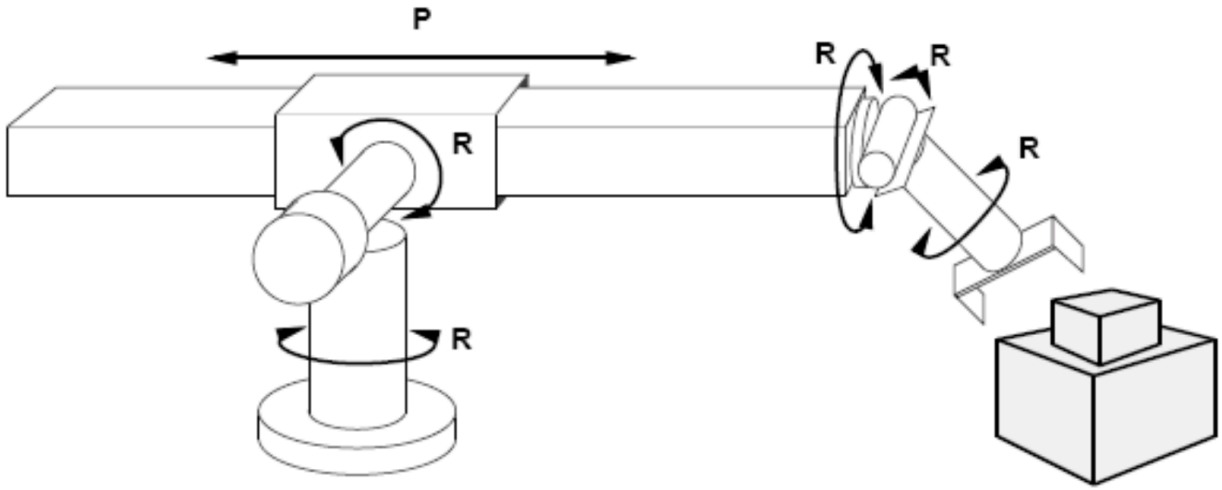


Goal State

- 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

Example: Robotic Assembly



- 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

例子：华容道

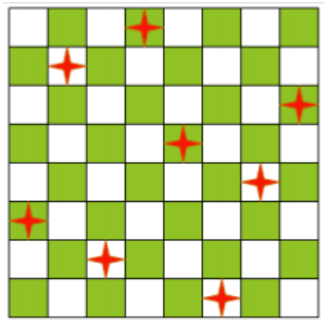

张飞 曹操 赵云
马超 关羽 黄忠
卒 卒 卒
卒
横刀立马(81)
赵云 曹操 卒 卒
关羽 卒 卒
马超 张飞 黄忠
卒
层拦叠障(62)
马超 曹操 黄忠
卒 关羽 卒
卒 张飞 卒
赵云
层层设防(102)
黄忠 曹操 卒 卒
关羽 张飞
赵云 马超
卒 卒
水泄不通(79)
卒 曹操 卒 卒
关羽 张飞
赵云 马超
黄忠
过五关(34)
曹操 赵云
马超 黄忠
关羽 卒 张飞
峰回路转(138)
张飞 曹操 卒 卒
赵云 马超 黄忠 卒
关羽
一路进军(58)
卒 关羽 卒
马超 曹操 黄忠
卒 张飞 卒
赵云
井中之蛙(68)

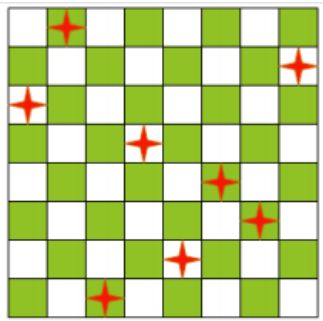
例子：华容道

华容道 $\rightarrow$ 搜索问题

- States: 棋盘的任意一种布局都是状态图的节点，所有状态的集合组成状态空间。
- Actions: 棋子的合法移动
- Goal test: 曹操是否跑出来了
- Path cost: 1 per move

例子：n 皇后问题


正确的解


错误的解

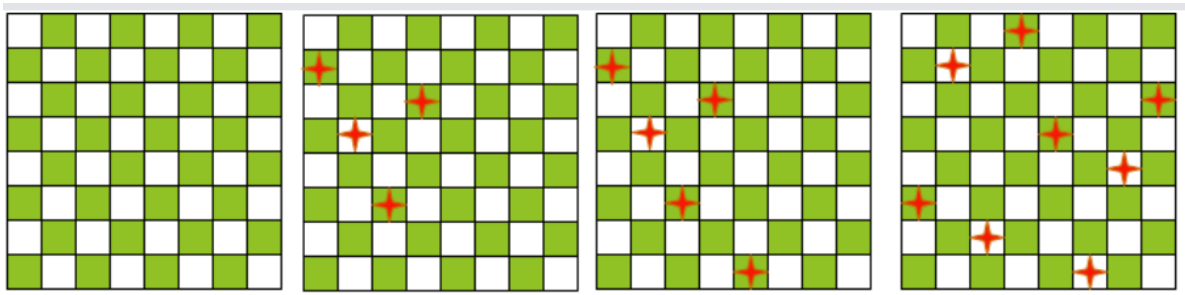
n 皇后问题 $\rightarrow$ 搜索问题

- States: 摆放 n 个皇后的棋盘
- Actions: 将某个皇后移动到相邻方格的行动
- Goal test: 皇后互相不攻击的棋盘布局
- Path cost: 1 per move

构建“好的”状态空间

- 何为“好的”状态空间
  - 尽可能让状态都可达，可行，合法
  - 状态数目越少越好，可以规避大量不可达的状态
  - 能够方便的设计后继函数与搜索算法
- 例如：n 皇后问题
  - 状态定义为：任何位置都可以放置皇后，$(N^2)!/(N^2-N+1)!$ 个状态（8 皇后为 3130929607680）
  - 状态定义为：每行放置一个皇后，N^N 个状态（8 皇后为 16777216，显著减少）

一种 n 皇后新的定义方式



- States: 从左往右，第 0, 1, ..., k 列依次放置皇后，且互不攻击
- Actions: 在第 k+1 列放置第 k+1 个皇后，且前 k+1 列的皇后不互相攻击
- Initial state: 0 个皇后在棋盘上
- Goal test: 8 个皇后在棋盘上
- 此时状态数仅为 2057 个，远少于上两种

Table of Contents

Problem-solving Agents

Searching for Solutions

Uninformed Search Strategies

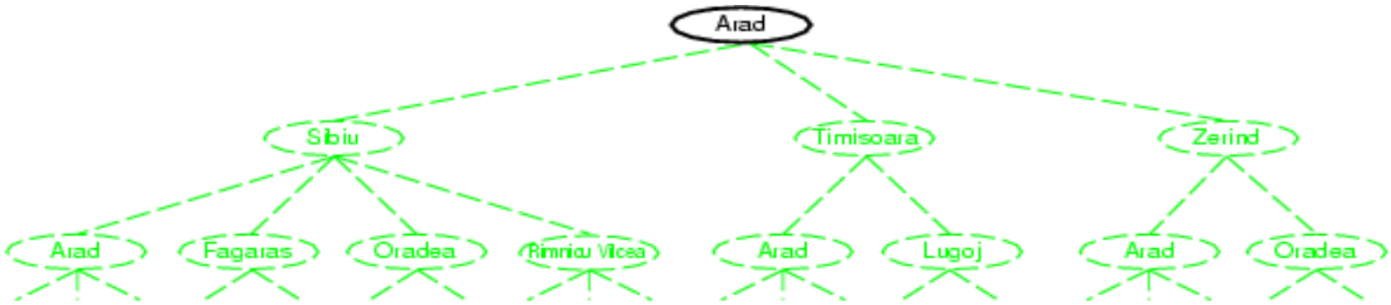
Tree Search Algorithms — 搜索树

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
```

Tree Search Example



function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure

initialize the search tree using the initial state of *problem*

loop do

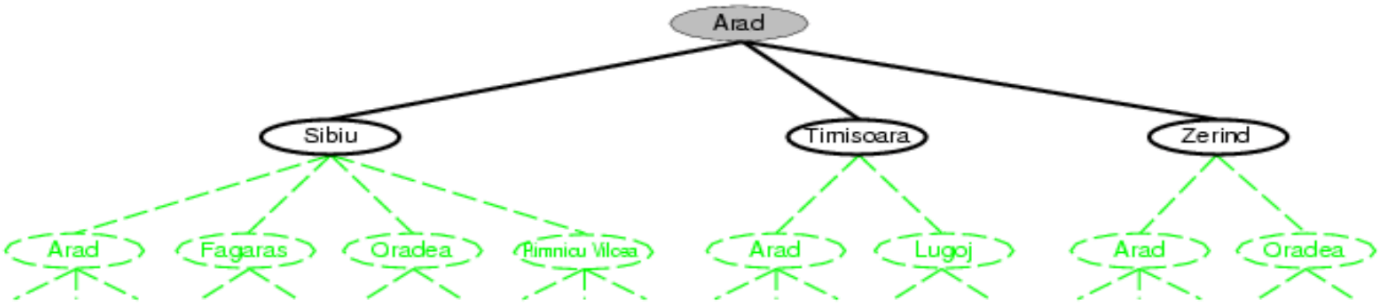
if there are no candidates for expansion **then return** failure

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



function TREE-SEARCH(*problem, strategy*) **returns** a solution, or failure

initialize the search tree using the initial state of *problem*

loop do

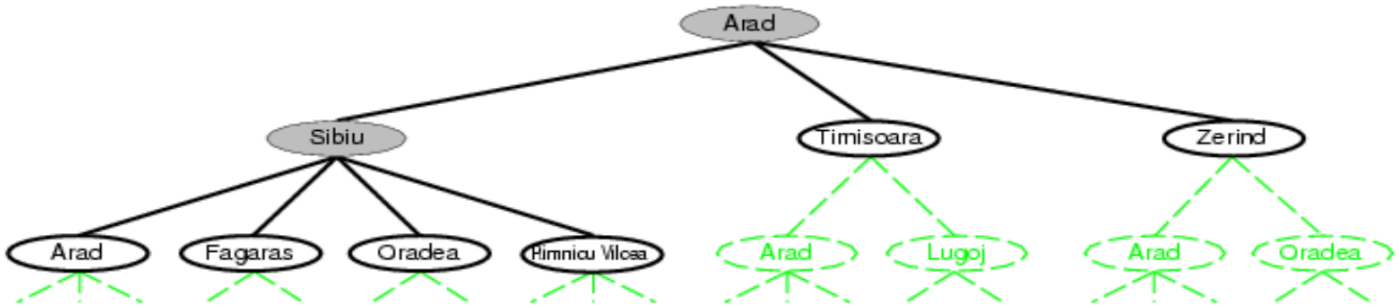
if there are no candidates for expansion **then return** failure

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

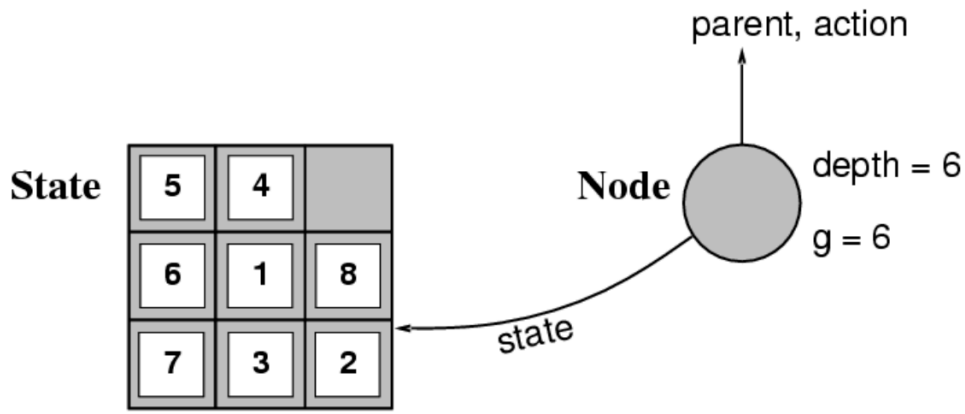
Tree Search Example



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

- 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

```
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 SOLUTION(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
```

General Tree Search vs. Graph Search

function TREE-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    expand the chosen node, adding the resulting nodes to the frontier

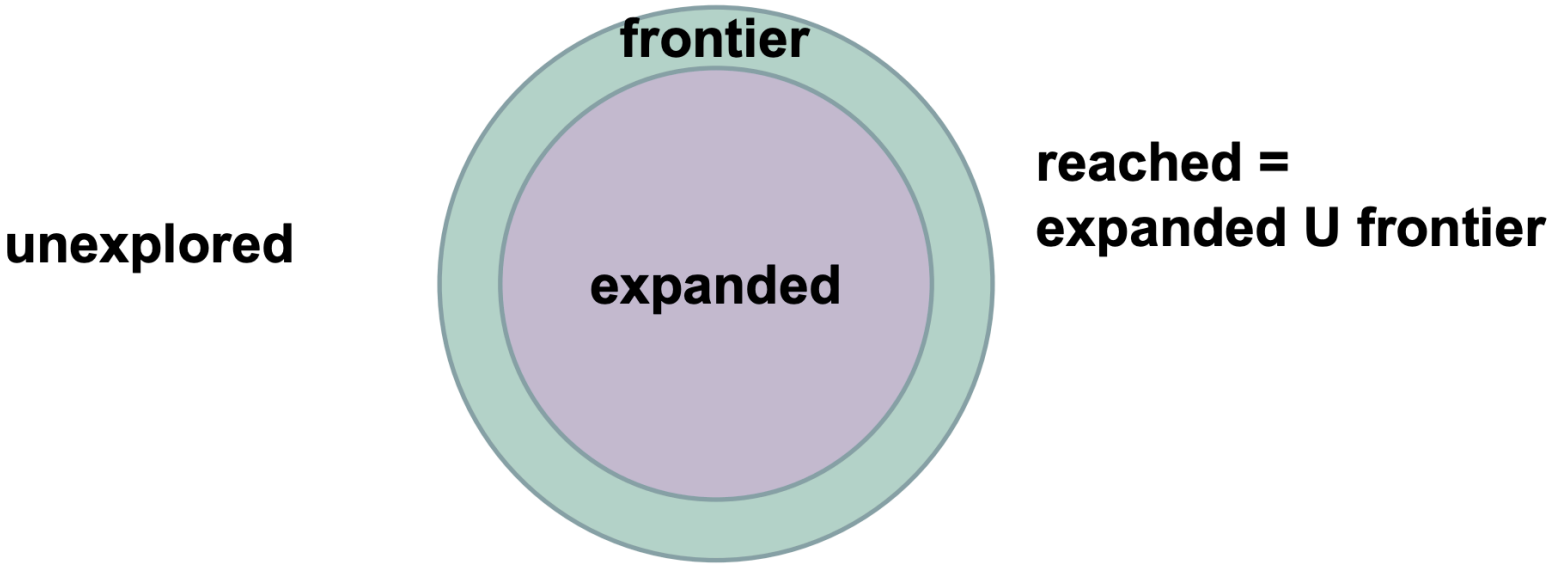
---

function GRAPH-SEARCH(*problem*) **returns** a solution, or failure
  initialize the frontier using the initial state of *problem*
  initialize the explored set to be empty
  loop do
    if the frontier is empty **then return** failure
    choose a leaf node and remove it from the frontier
    if the node contains a goal state **then return** the corresponding solution
    add the node to the explored set
    expand the chosen node, adding the resulting nodes to the frontier
      only if not in the frontier or explored set

Main variations:

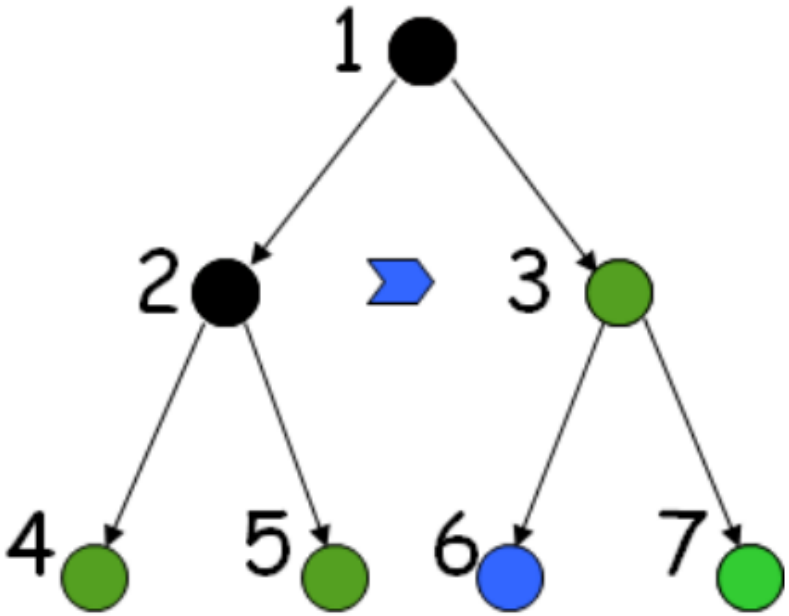
- Which leaf node to expand next
- Whether to check for repeated states
- Data structures for frontier, expanded nodes

Systematic Search for Graph Search



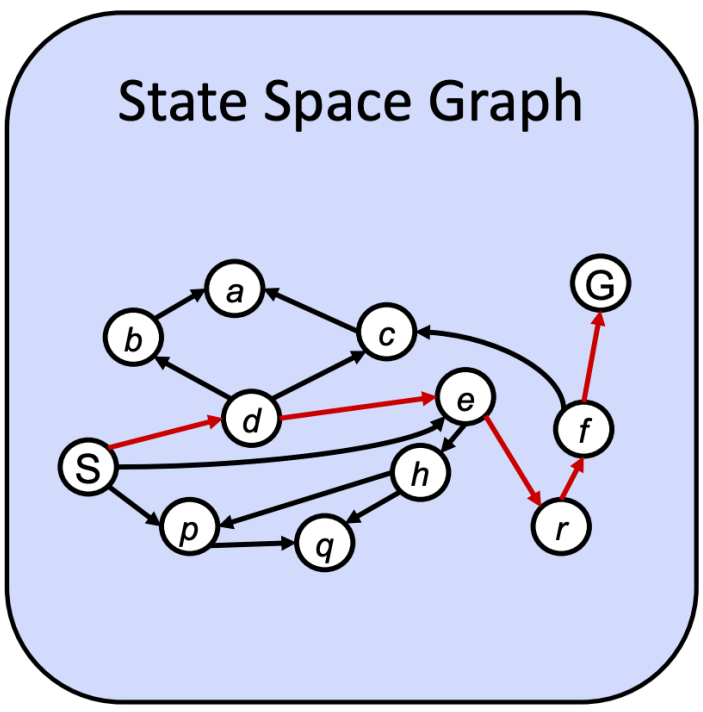
- Frontier (FRINGE) separates expanded from unexplored region of state-space graph
- Expanding a frontier node:
  1. Moves a node from frontier into expanded
  2. Adds nodes from unexplored into frontier, maintaining the first property

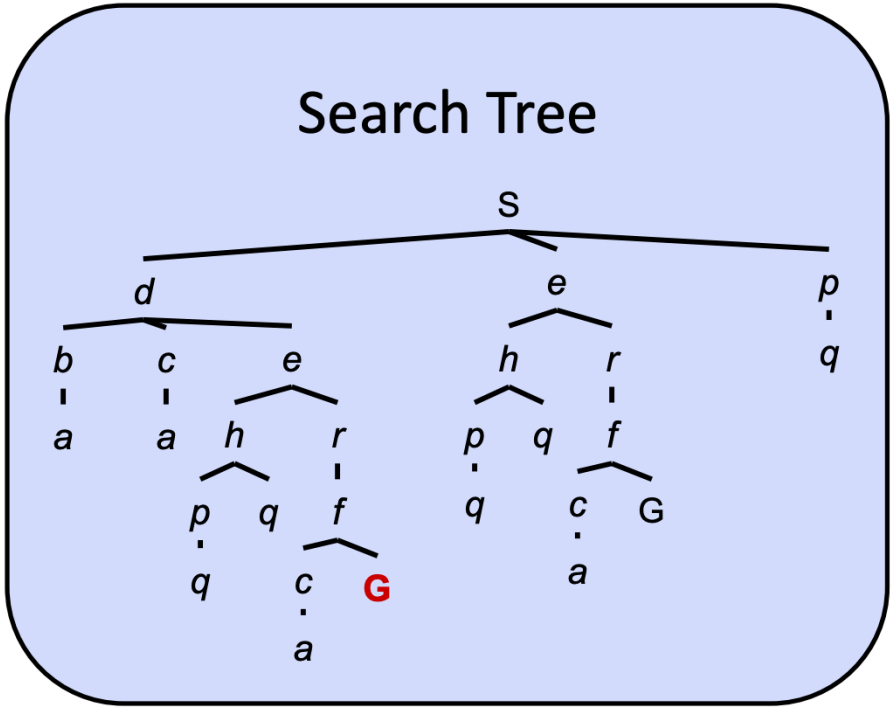
Frontier



- 如左图所示：1, 2 为已扩展节点 (expanded)，则 $\mathit{Frontier} = \{3, 4, 5\}$
- 当选择 3 扩展后，$\mathit{Frontier} = \{4, 5, 6, 7\}$

State Space Graphs vs. Search Trees

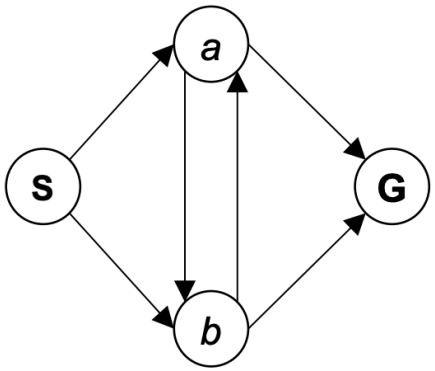




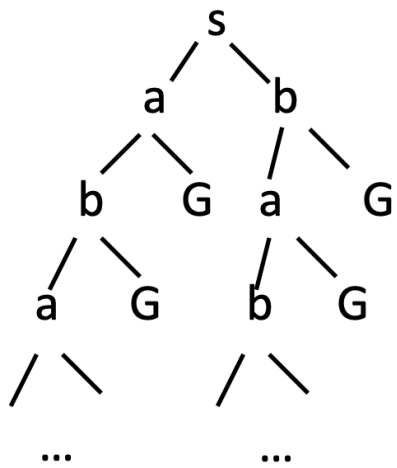
- Each NODE in the search tree is an entire PATH in the state space graph
- We construct the tree on demand and we construct as little as possible.

State Space Graphs vs. Search Trees

Consider this 4-state graph:

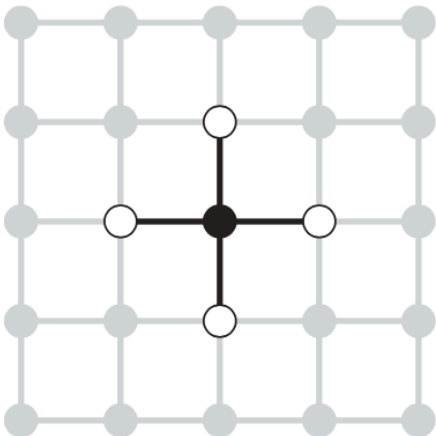
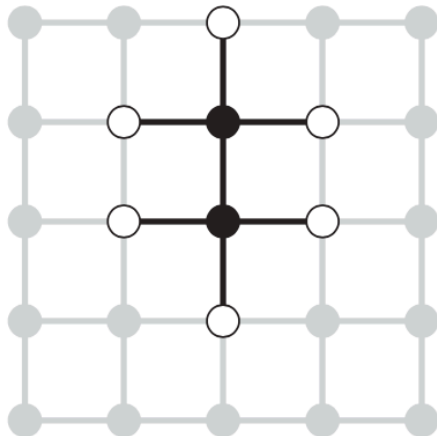
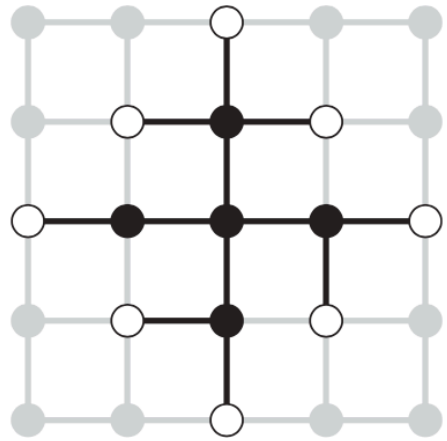


How big is its search tree (from S)?



- Important: Those who don't know history are doomed to repeat it!

State Space Graphs vs. Search Trees

- A search tree of depth d that includes repeated states has 4^d leaves
- There are only about $2d^2$ distinct states within d steps of any given state

Measuring Problem-solving Performance

- 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?

Search Algorithm Properties

- 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 — 目标节点最小深度
  - m: maximum depth of the state space (may be ∞) —– 最大深度

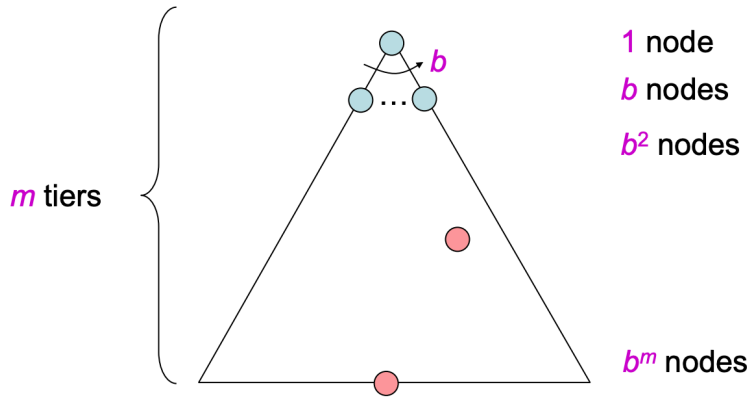


Table of Contents

Problem-solving Agents

Searching for Solutions

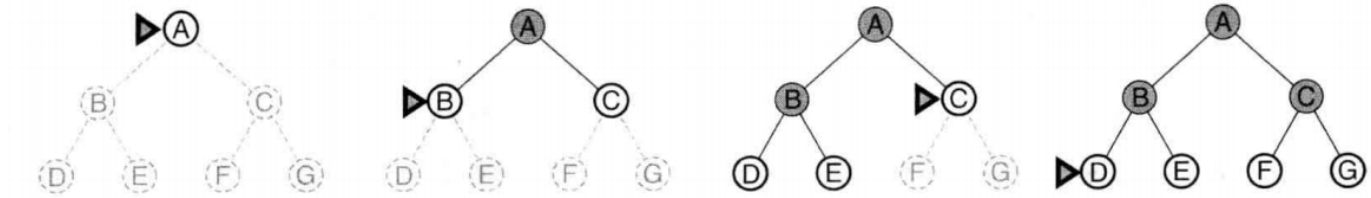
Uninformed Search Strategies

Uninformed (无信息的) Search Strategies

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

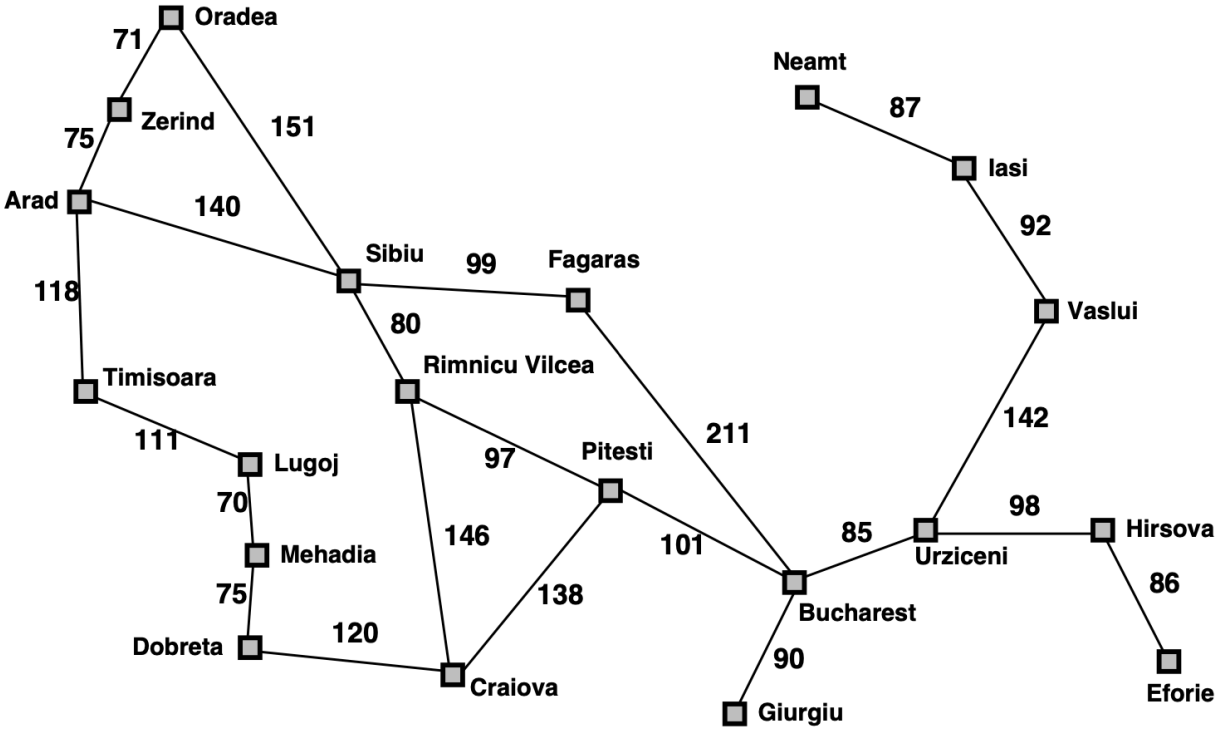
- 定义：无信息搜索是指除了问题定义中提供的状态信息外没有任何附加信息。与之相对应的是有信息搜索（启发式搜索），可以知道一个非目标状态是否比其他状态更有希望接近目标状态。
- 这里我们介绍以下 6 种无信息搜索策略：
  - Breadth-first search 广度优先搜索
  - Uniform-cost search 一致代价搜索
  - Depth-first search 深度优先搜索
  - Depth-limited search 深度受限搜索
  - Iterative deepening search 迭代加深的深度优先搜索
  - Bidirectional search 双向搜索

Breadth-first Search 广度优先搜索



- Frontier 选用 FIFO (first-in, first-out) 队列
- 完备性：完备 (if d is finite)
- 最优性：若每条边 cost 一致 (if cost=1 per step)，则一定返回最优解；否则不一定
- 时间复杂度:
  - 访问节点数 $\leq 1 + b + b^2 + b^3 + \cdots + b^d = O(b^d)$
  - If the algorithm were to apply the goal test to nodes when selected for expansion, rather than when generated, the whole layer of nodes at depth d would be expanded before the goal was detected and the time complexity would be $\leq 1 + b + b^2 + b^3 + \cdots + b^d + b(b^d - 1) = O(b^{d+1})$
- 空间复杂度：$O(b^d)$ or $O(b^{d+1})$ (扩展节点做 goal test), 所有节点均被存储

Romania with Step Costs in km

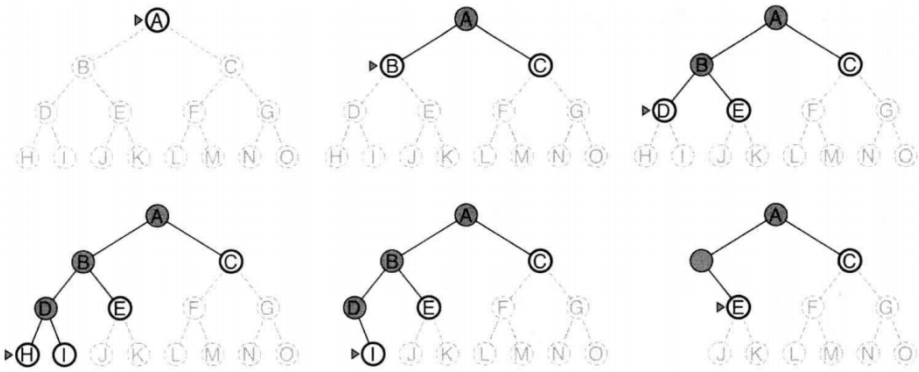


Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Uniform-cost Search 一致代价搜索

- Expand least-cost unexpanded node
- Implementation: Frontier = queue ordered by path cost
- Frontier 选用优先级队列，先扩展 Path-Cost 小的节点；若所有路径 cost 一致，则一致代价搜索等于广度优先搜索
- 完备性：完备，若单步 cost 有下界 ϵ (if step cost $\geq \epsilon$)
- 最优性：最优, nodes expanded in increasing order of $g(n)$
- 时间复杂度：# of nodes with g $\leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, 其中 C^* 代表最优解所需的代价
- 空间复杂度：# of nodes with g $\leq$ cost of optimal solution, $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

Depth-first Search 深度优先搜索



- Frontier 选用 LIFO (last-in, first-out) 队列
- 完备性：No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path ⇒ complete in finite spaces
- 最优性：不保证最优
- 时间复杂度：访问节点数 $\leq 1 + b + b^2 + b^3 + \cdots + b^m = O(b^m)$
  - terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
- 空间复杂度：$O(bm)$, *i.e.*, linear space! 只存储一条根到叶节点的路径，以及该路径上节点未被扩展的兄弟节点

Depth-limited Search 深度受限搜索

= depth-first search with depth limit l,

i.e., nodes at depth l have no successors

- 目的为了避免深度优先搜索一条路走到黑的尴尬
- 设置一个深度界限 l，若节点深度大于 l 时，不则再扩展
- 返回的结果：
  - 有解
  - 无解
  - 在 l 范围内无解
- 完备性：不是
- 最优性：不保证
- 时间复杂度：$O(b^l)$
- 空间复杂度：$O(bl)$
- Solves infinte-depth path problem
- if $l < d$, possibly incomplete
- If $l > d$, not optimal

Iterative Deepening Search 迭代加深的深度优先搜索

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 ∞ do
            result DEPTH-LIMITED-SEARCH( problem, depth)
            if result ≠ cutoff then return result
    end
```

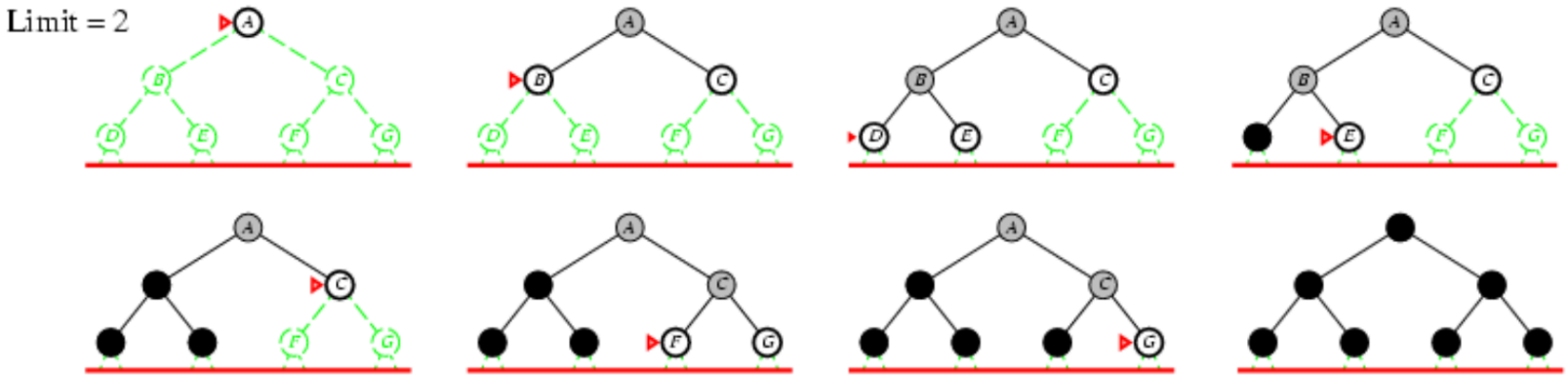
- 使用不同的深度界限 $k = 0, 1, 2, \ldots$ 不断重复执行“深度受限搜索”
- 结合了广度优先搜索和深度优先搜索的优点
- 完备性：完备
- 最优性：若每条边 cost 一致 (if step cost $= 1$)，则一定返回最优解；否则不一定
- 时间复杂度：$db + (d-1)b^2 + (d-2)b^3 \cdots + b^d = O(b^d)$
- 空间复杂度：$O(bd)$, 深度界限达到 d

Iterative Deepening Search $l = 0$

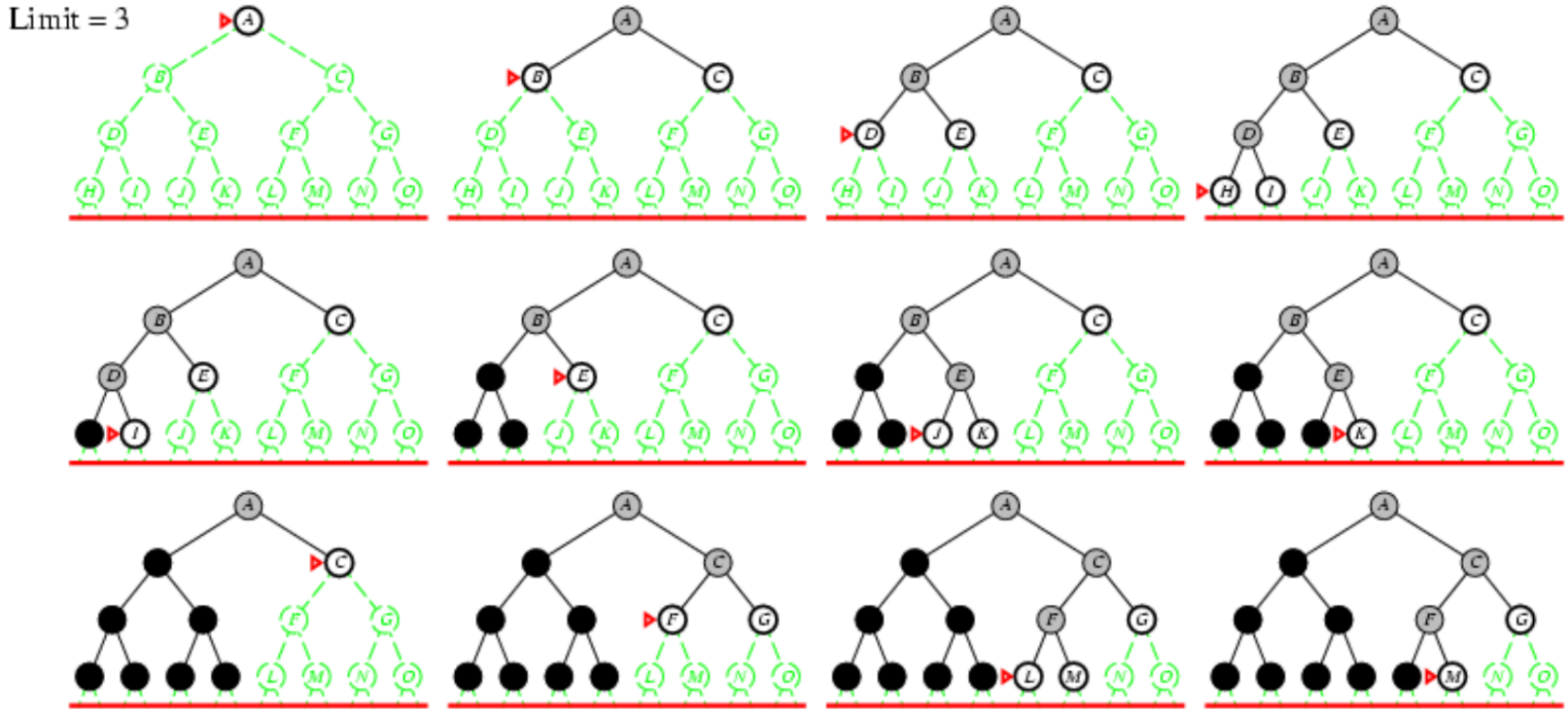
Limit = 0

Iterative Deepening Search $l = 1$

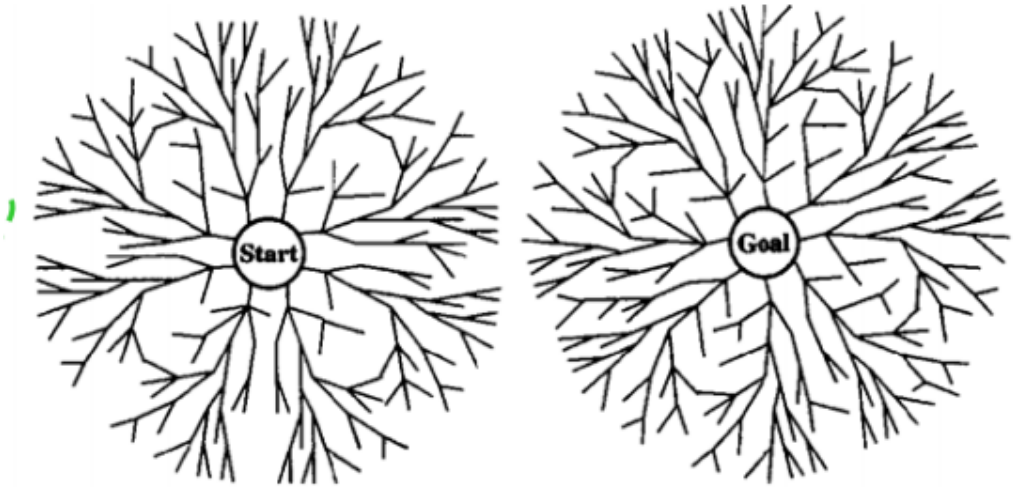
Iterative Deepening Search $l = 2$



Iterative Deepening Search $l = 3$



Bidirectional Search 双向搜索



- 从初态和终态同时进行广度优先搜索，但其中的难点是：终态可能不好描述（比如 n 皇后问题的终态）；由终态返回的函数可能不好描述
- 完备性：完备
- 最优性：若每条边 cost 一致，则一定返回最优解；否则不一定
- 时间复杂度：$O(b^{d/2})$
- 空间复杂度：$O(b^{d/2})$

搜索策略总结

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes[c]	Yes	No	No	Yes[c]	Yes[c,d]

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: [a] complete if b is finite; [b] complete if step costs $\geq \epsilon$ for positive ϵ; [c] optimal if step costs are all identical; [d] if both directions use breadth-first search.

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

- $N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450$
- $N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 = 111,100$

There is some extra cost for generating the upper levels multiple times, but it is not large.

In general, iterative deepening is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Summary

- 如何定义一个搜索问题
  - States S, initial state s_0, actions $A(s)$, transition model $Result(s, a)$, goal test $G(s)$, path cost
- 无信息搜索策略
  - 广度优先搜索
  - 一致代价搜索
  - 深度优先搜索
  - 深度受限搜索
  - 迭代加深的深度优先搜索
  - 双向搜索
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

作业

第二版书：

- 3.7 (a, b, d)
- 3.9