Problem Solving Agents

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

PoAI

Contents

- 3.1.1 Problem Solving in AI
- 3.1.2 Algorithm of Simple Problem Solving Agents
- 3.1.3 Related Terms
- 3.1.4 Five Items to Formulate a Problem

Problem solving in AI 人工智能中的问题求解

- The solution 解

  is a sequence of actions to reach the goal.

  是一个达到目标的动作序列。
- The process 过程

  look for the sequence of actions, which is called search.

  寻找该动作序列，称其为搜索。
- Problem formulation 问题形式化

  given a goal, decide what actions and states to consider.

  给定一个目标，决定要考虑的动作与状态。
- Why search 为何搜索

  Some NP-complete or NP-hard problems, can be solved only by search.

  对于某些NP完或者NP难问题，只能通过搜索来解决。
- Problem-solving agent 问题求解智能体

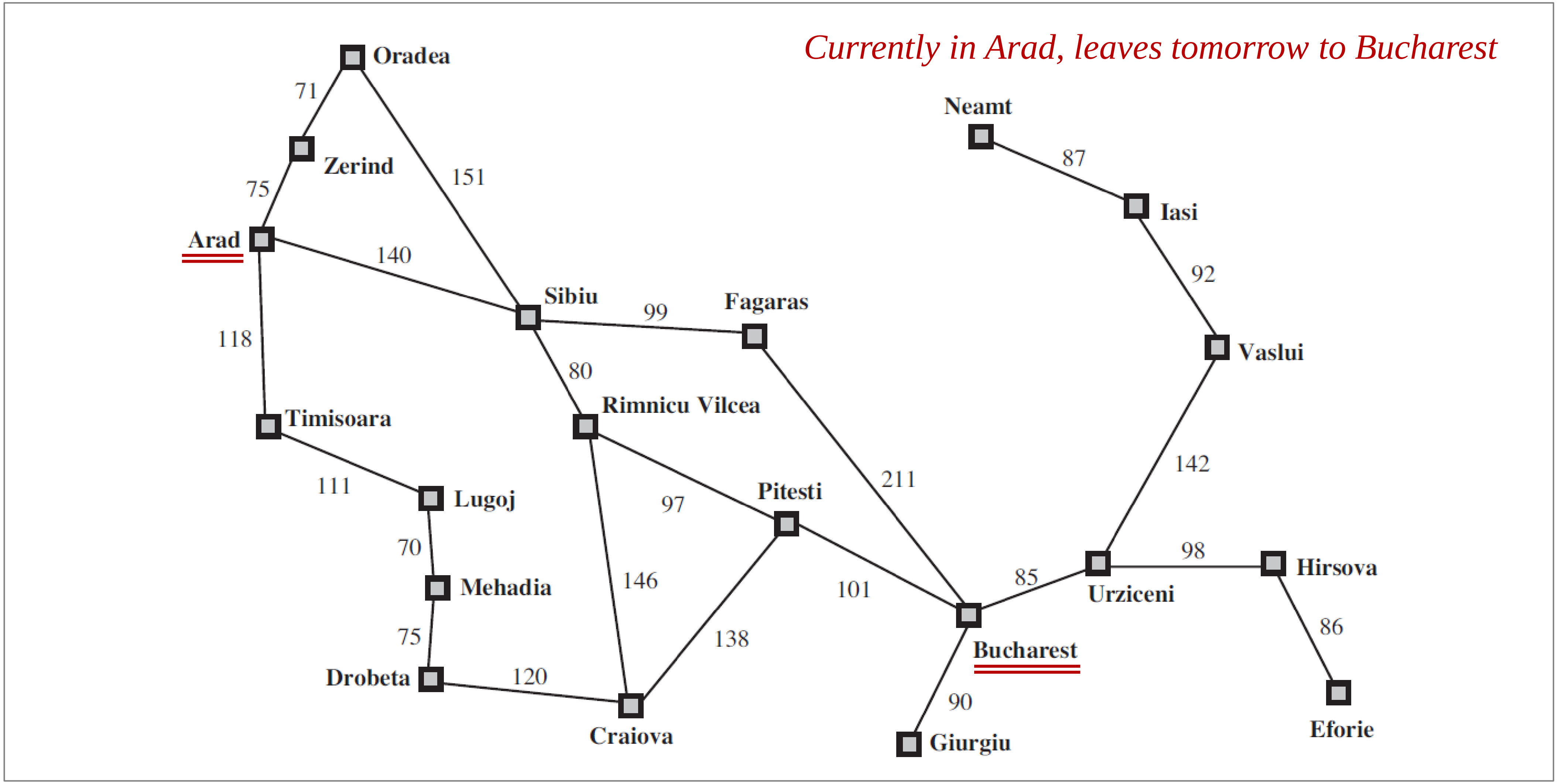
  is a kind of goal-based agent to solve problems through search.

  是一种基于目标的智能体，通过搜索来解决问题。

Algorithm of Simple Problem Solving Agents 简单的问题求解智能体算法

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    persistent: seq, an action sequence, initially empty
                state, some description of the current world state
                goal , a goal, initially null
                problem, a problem formulation
                action, the most recent action, initially none
    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
        if seq = failure then return a null action
    action ← FIRST(seq)
    seq ← REST(seq)
    return action
```

Example: A road map of part of Romania 罗马尼亚部分公路图



Related Terms 相关术语

☐ State space 状态空间

The state space of the problem is formally defined by: Initial state, actions and transition model.

问题的状态空间可以形式化地定义为：初始状态、动作和转换模型。

☐ Graph 图

State space forms a graph, in which nodes are states, and links are actions.

状态空间形成一个图，其中节点表示状态、链接表示动作。

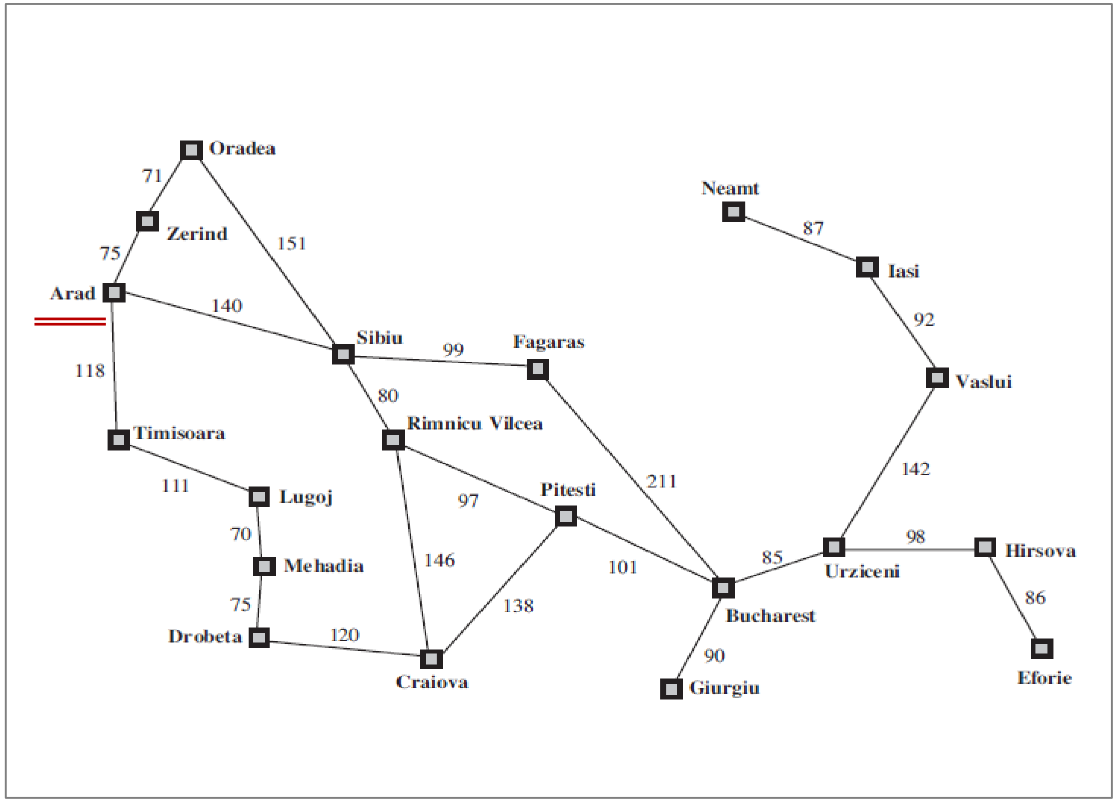
☐ Path 路径

A path in the state space is a sequence of states connected by a sequence of actions.

状态空间的一条路径是由一系列动作连接的一个状态序列。

Five Items to Formulate a Problem 问题形式化的五个要素

1) Initial state 初始状态



- The agent starts in.

  即智能体出发时的状态。

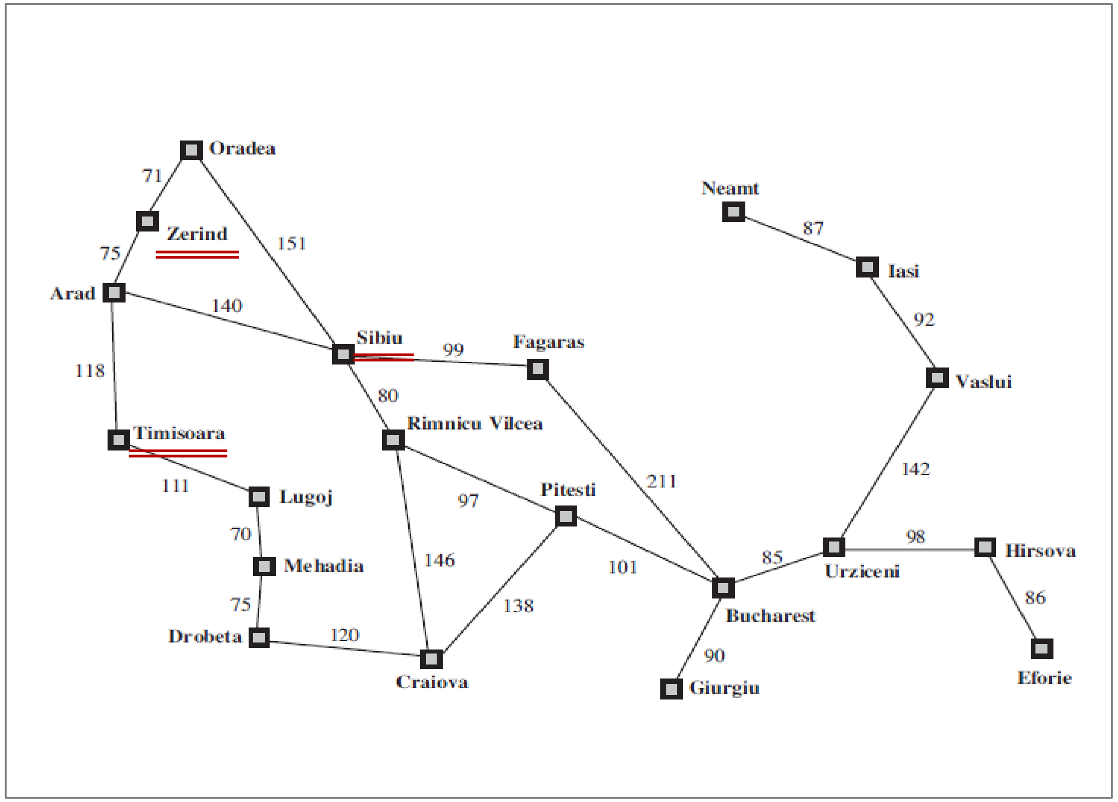
- E.g., the initial state for the agent in Arad may be described as:

  例如，该智能体位于Arad的初始状态可以记作：

$$In(Arad).$$

Five Items to Formulate a Problem:

☐ 2) Actions 动作



- A description of the possible actions available to the agent.

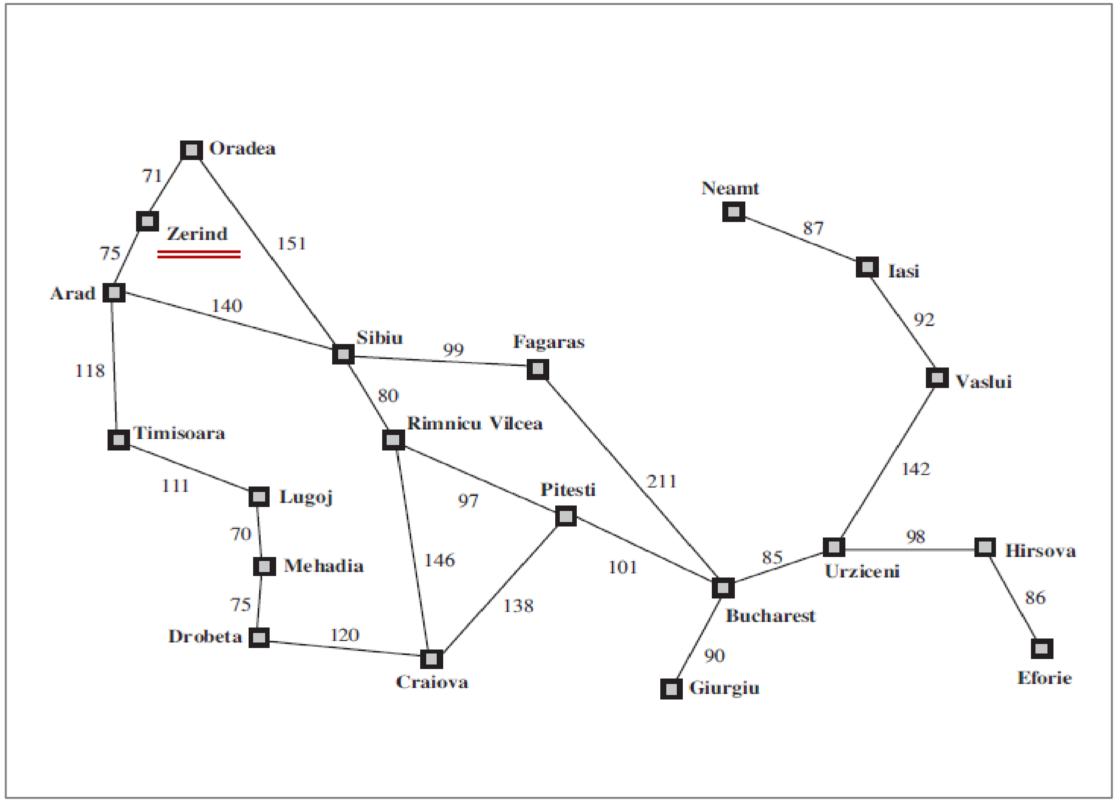
  描述该智能体可执行的动作。

- ACTION(s) returns the actions that can be executed in s. E.g., ACTION(s) 返回s状态下可执行的动作序列。例如：

$$\{Go(Zerind), Go(Sibiu), Go(Timisoara)\}.$$

Five Items to Formulate a Problem:

☐ 3) Transition model 转换模型

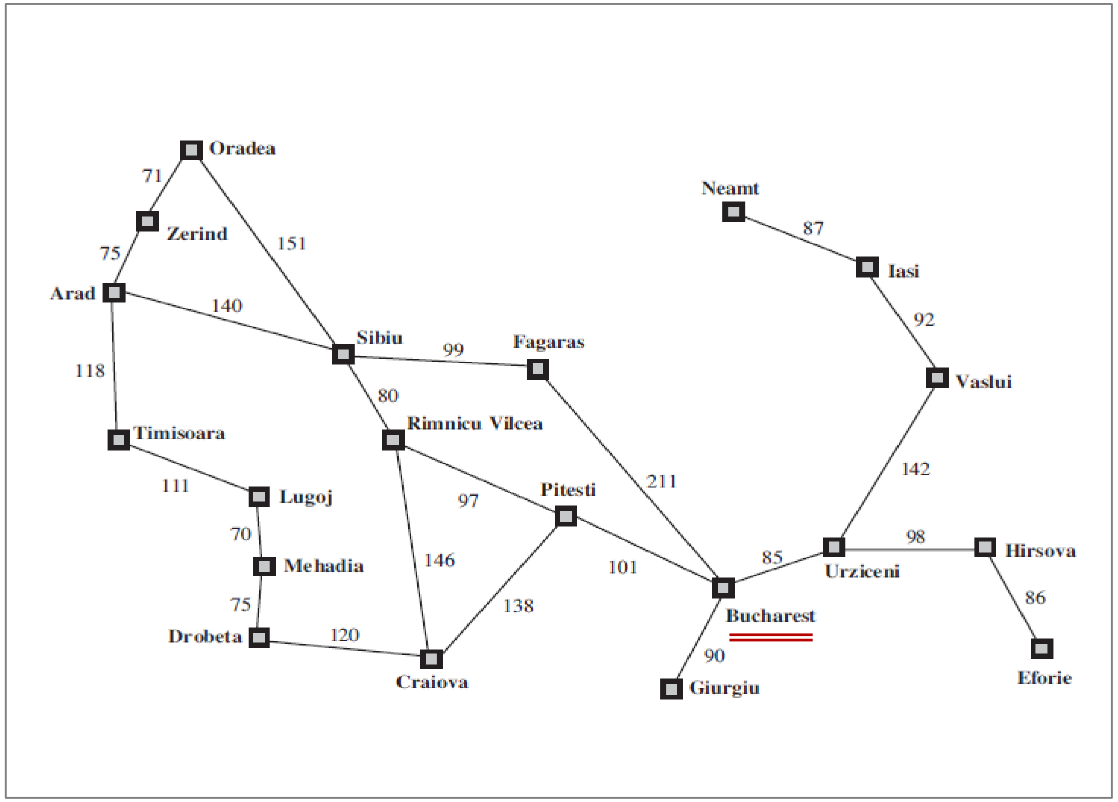


- A description of what each action does.
  描述每个动作做什么。
- $\text{RESULT}(s, a)$ returns the state from doing action a in s. E.g., $\text{RESULT}(s, a)$ 返回在s下动作a之后的状态。例如：

$$\text{RESULT}(In(Arad), Go(Zerind)) = In(Zerind)$$

Five Items to Formulate a Problem:

□ 4) Goal test 目标测试

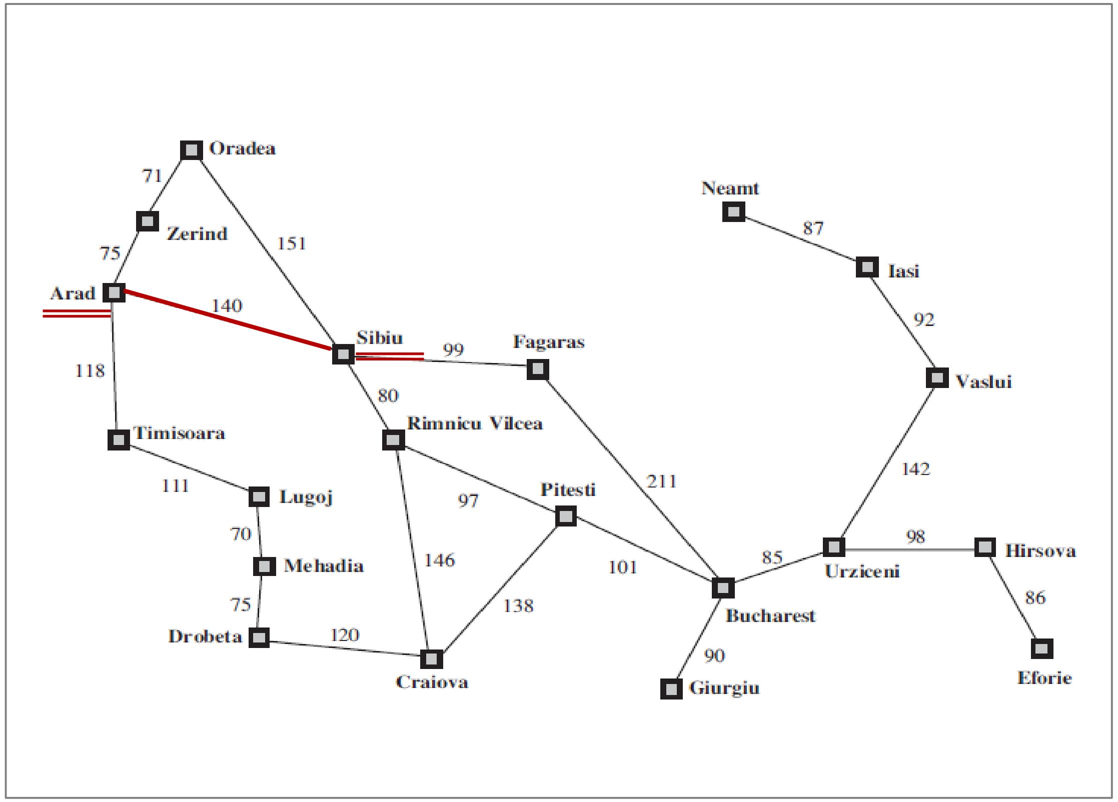


- To determine whether a given state is a goal state.
  确定一个给定的状态是否是目标状态。
- E.g., the agent's goal in Bucharest is the singleton set:
  例如：智能体在Bucharest的目标是单元素集合：

$$\{In(Bucharest)\}.$$

Five Items to Formulate a Problem:

□ 5) Path cos 路径代价



- To assign a numeric cost to each path.

  即每条路径所分配的一个数值代价。

- E.g., step cost of taking action a in state s to reach state s' is denoted by:

  例如：状态s下执行动作a到达状态s’ 的步骤代价表示为：

$$c(s, a, s').$$

Thank you for your attention!

PoAI

Example Problems

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

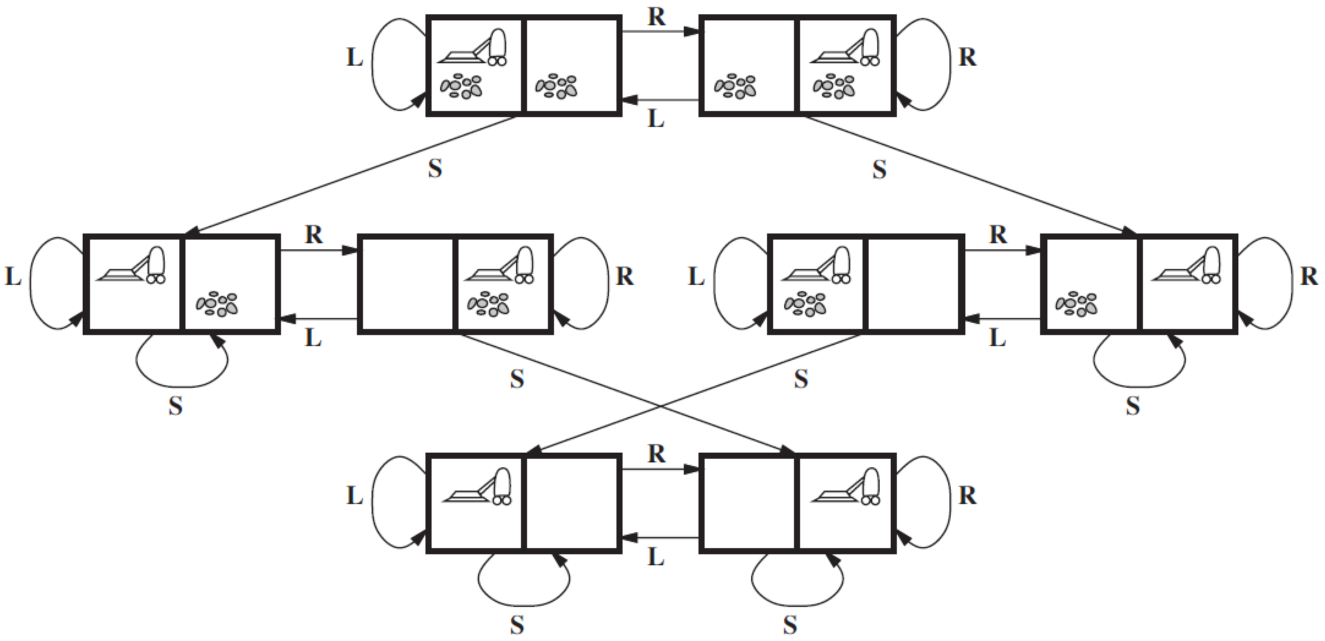
Contents

- 3.2.1 Example1: Vacuum-cleaner world
- 3.2.2 Example2: 8-puzzle
- 3.2.3 Example3: 8-queens problem

Example 1: Vacuum-cleaner world 真空吸尘器世界

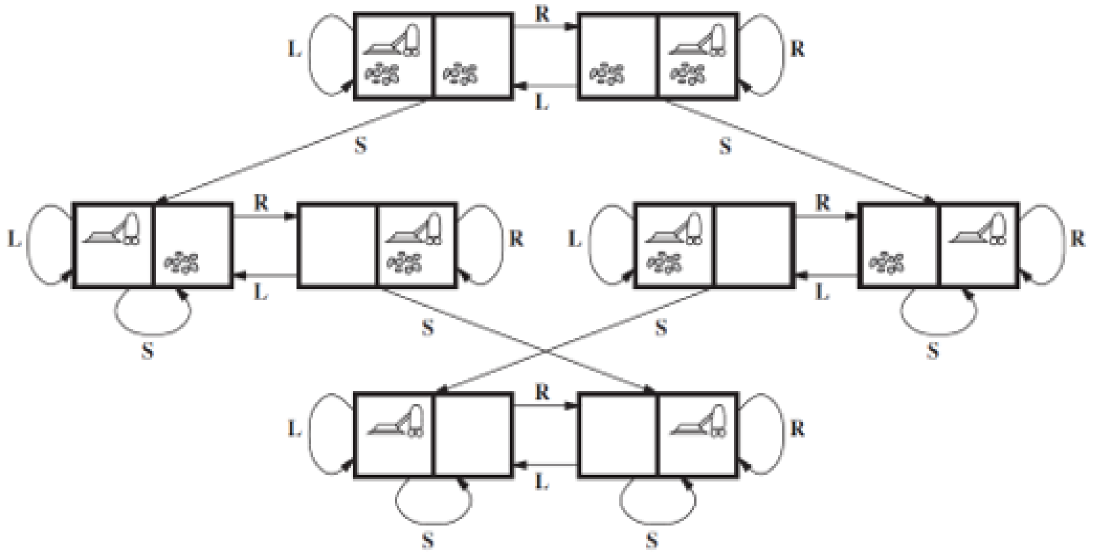
- Vacuum-cleaner world has been introduced in “2.1.6. Intelligent agent paradigm”.
  真空吸尘器世界已经在“2.1.6. Intelligent agent paradigm”中讲过。

- The states are determined by both the agent location and dirt location.
  其状态是由智能体的位置和灰尘的位置决定的。
- Links denote actions:
  L = *Left*, R = *Right*, S = *Suck*.
  链接表示动作：
  L = 左移, R = 右移, S = 吸尘。



Example 1: Vacuum-cleaner world 真空吸尘器世界

- **States 状态**
  - Agent is in one of two locations, each may contain dirt or not.
    智能体在两个地点中的一个，每个也许有灰尘或者没有。
  - Possible states, 2 locations: $2 \times 2^2 = 8$ ($n \times 2^n$).
    可能的状态，2个地点： $2 \times 2^2 = 8$（$n \times 2^n$）。
- **1) Initial state 初始状态**
  Any state can be as the initial state.
  任何状态都可以作为初始状态。
- **2) Actions 动作**
  Each state has just three actions:
  Left, *Right*, and *Suck*.
  每个状态仅有三个动作：左移，右移，以及吸尘。



Example 1: Vacuum-cleaner world 真空吸尘器世界

- 3) Transition model 转换模型

  The actions have their expected effects, except that moving:

  该动作应有的预期效果，下述动作除外：

  - *Left* in the leftmost, 在最左边进行左移
  - *Right* in the rightmost, 在最右边进行右移
  - *Suck* in a clean square. 在清洁区域进行吸尘

- 4) Goal test 目标测试

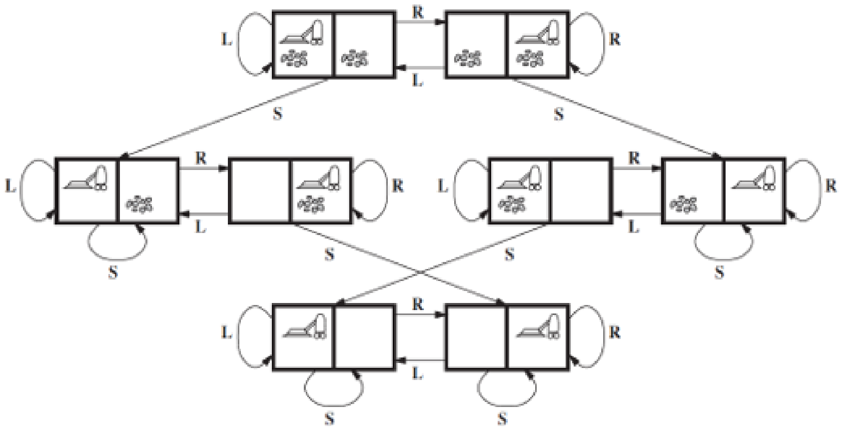
  Whether all the squares are clean.

  是否所有的区域内都干净。

- 5) Path cost 路径代价

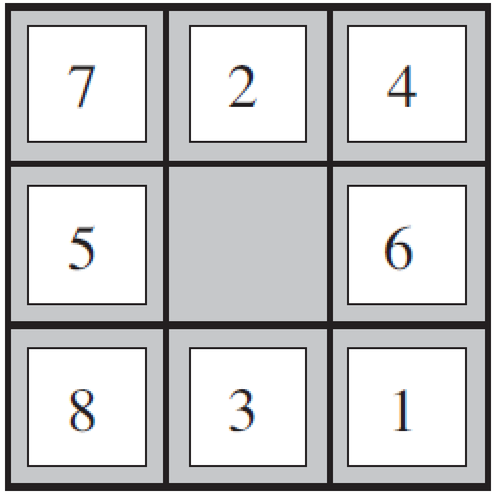
  The number of steps in the path (each step costs 1).

  等于路径的步数（每一步的代价）。

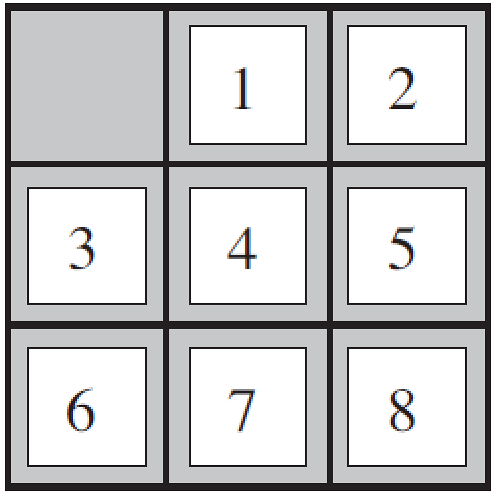


Example 2: 8-puzzle 8数码难题

- 8-puzzle: 3×3 board with 8 numbered tiles and a blank space.
  8数码难题：3×3棋盘上有8个数字棋子和一个空格。



Start state


Goal state

A tile adjacent to the blank space can slide into the space. The object is to reach a specified goal state.
与空格相邻的滑块可以移向该空格，目的是达到一个指定的目标状态。

Example 2: 8-puzzle 8数码难题

- **States** 状态
  - Each of 8 numbered tiles in one of the 9 squares, and blank in the last square.
    8个数字滑块每个占据一个方格，而空格则位于最后一个方格。
- **1) Initial state** 初始状态
  - Any state can be the initial state.任意一个状态都可以成为初始状态。
- **2) Actions** 动作
  - Simplest formulation defines the actions as movements of the blank space: *Left*, *Right*, *Up*, or *Down*.
    最简单的形式化是将动作定义为空格的移动：左、右、上、下。
  - Different subsets are depending on where the blank is.
    不同的子集依赖于空格的位置。

7	2	4
5		6
8	3	1

Start state

Example 2: 8-puzzle 8数码难题

- 3) Transition model 转换模型

  Given a state and action, this returns the resulting state. E.g., if we apply *Left* to the start state, the resulting state has the 5 and the blank switched.

  给定状态和动作，其返回结果状态。例如，如果我们对初始状态施加左移动作，由此产生的状态则使5与空格互换。

- 4) Goal test 目标测试

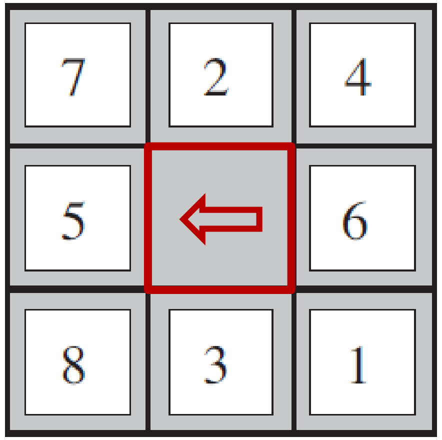
  Checks whether the state matches the goal configuration.

  即检查状态是否与目标布局相符。

- 5) Path cost 路经代价

  The number of steps in the path (each step costs 1).

  等于路径的步数（每一步的代价）。



Transition

Sliding block puzzles 滑块难题

- The 8-puzzle belongs to the family of sliding block puzzles, this family is known to be NP-complete.

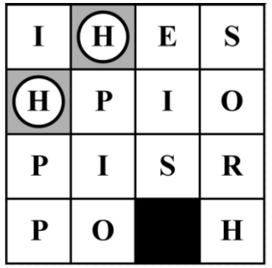
  8数码难题属于滑块难题家族，这个家族被认为是NP完的。

3x3 sliding puzzle.

7x7 sliding block puzzle

华容道

15-puzzle

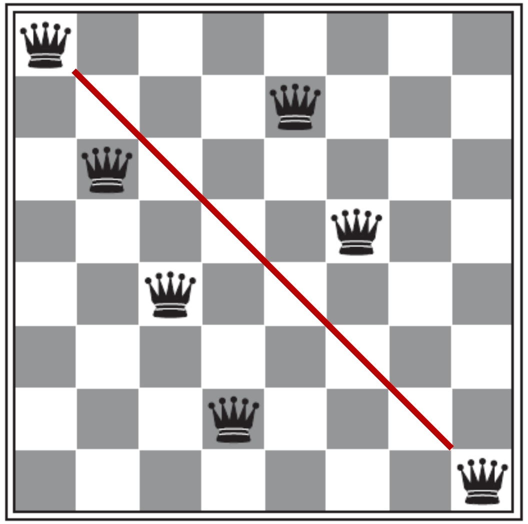


Word puzzle

Example 3: 8-queens problem 8皇后问题

- The goal is to place 8 queens on a chessboard such that no queen attacks any other. (A queen attacks any piece in the same row, column or diagonal.)
  其目标是将8个皇后摆放在国际象棋的棋盘上，使得皇后之间不发生攻击（一个皇后会攻击同一行、同一列或同一斜线上的其他皇后）。

- Two main kinds of formulation:
  两种主要类型的形式化方法：
  - Incremental formulation: starts with an empty state, then each action adds a queen to change the state.
    增量形式化：从空状态开始，然后每次添加一个皇后改变其状态。
  - Complete-state formulation: starts with all 8 queens on the board, and moves them around.
    全态形式化：初始时8个皇后都放在棋盘上，然后再将她们移开。



A queen attacks another one in the same diagonal.

Example 3: 8-queens problem 8皇后问题

The incremental formulation 增量式形式化

- States: Any arrangement of 1 to 8 queens on the board is a state.
  状态：第1至第8个皇后在棋盘上任意摆放，为一个状态。
- 1) Initial state: No queens on the board.
  初始状态：棋盘上没有皇后。
- 2) Actions: Add a queen to any empty square.
  动作：添加一个皇后至任意一个空格。
- 3) Transition model: Returns the board with a queen added to the specified square.
  转换模型：将一个皇后添加到指定空格，再返回该棋局。
- 4) Goal test: 8 queens are on the board, none attacked.
  目标测试：8个皇后都在棋盘上，并且没有攻击。
- 5) Path cost: The number of steps (each step costs 1).
  路径代价：等于步数（每步代价为1）。

Thank you for your attention!

PoAI

Searching for Solutions

School of Electronic and Computer Engineering
Peking University

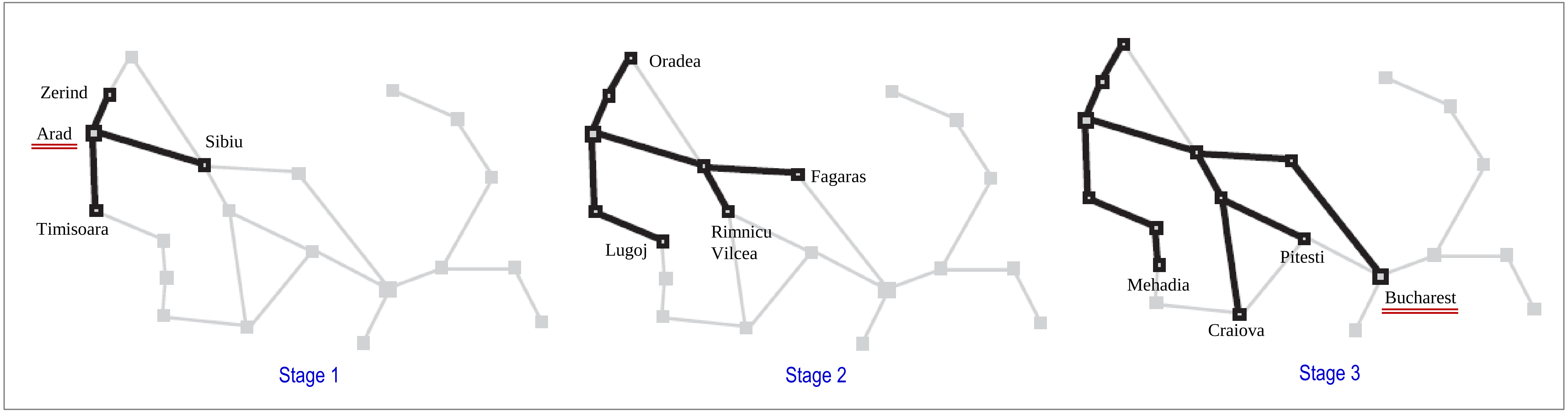
Wang Wenmin

Contents

- 3.3.1 Shortest Path Problem by Graph Search
- 3.3.2 Shortest Path Problem by Tree Search

Shortest Path Problem by Graph Search 采用图搜索的最短路径问题

- A sequence of search paths generated by a graph search on the Romania map.
  通过图搜索在该罗马尼亚地图上生成一系列搜索路径。



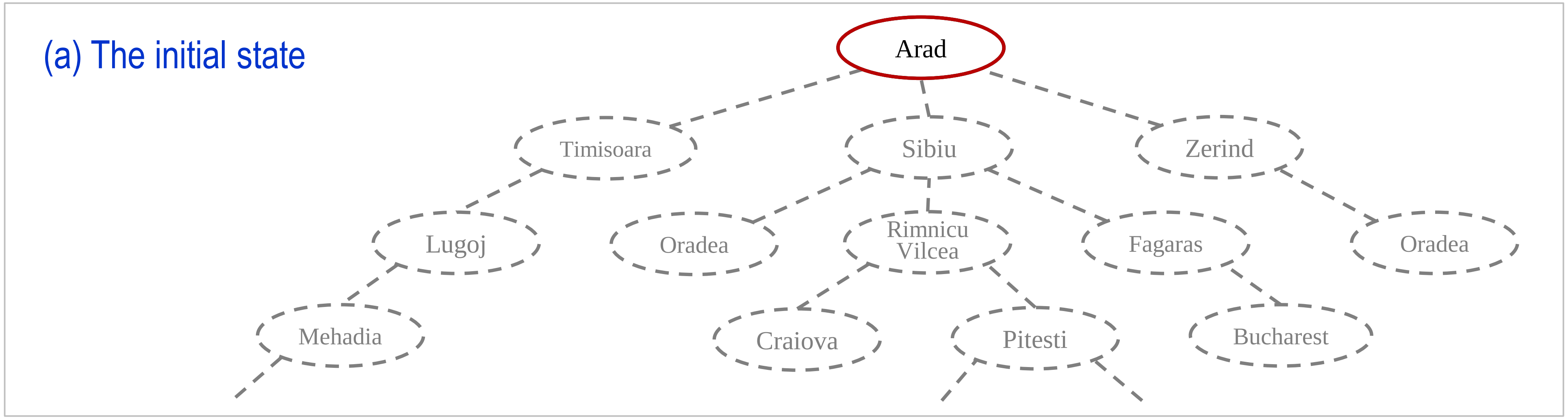
Each path has been extended at each stage by one step. Notice that at 3rd stage, the northernmost city (Oradea) has become a dead end.

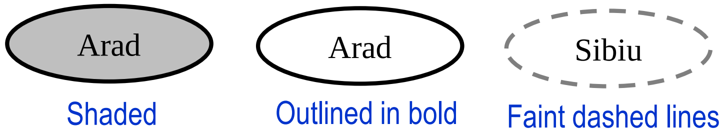
每个路径在每个阶段通过每一步加以扩展扩展。注意在第3阶段，最北部城市(Oradea)已成为死胡同。

Shortest Path Problem by Tree Search 采用树搜索的最短路径问题

- Use search trees to find a route Arad to Bucharest.

  用搜索树来寻找一条从Arad到Bucharest的路径。





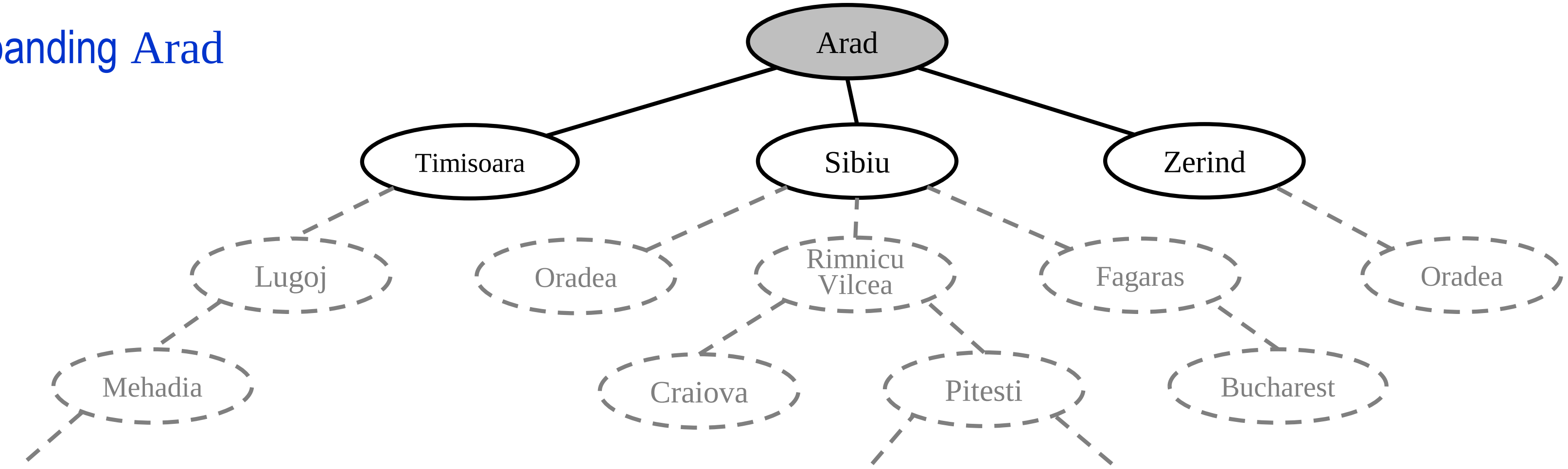
Shaded: the nodes that have been expanded.
阴影：表示该节点已被扩展。

Outlined: the nodes that have been generated but not yet expanded.
粗实线：表示该节点已被生成，但尚未扩展。

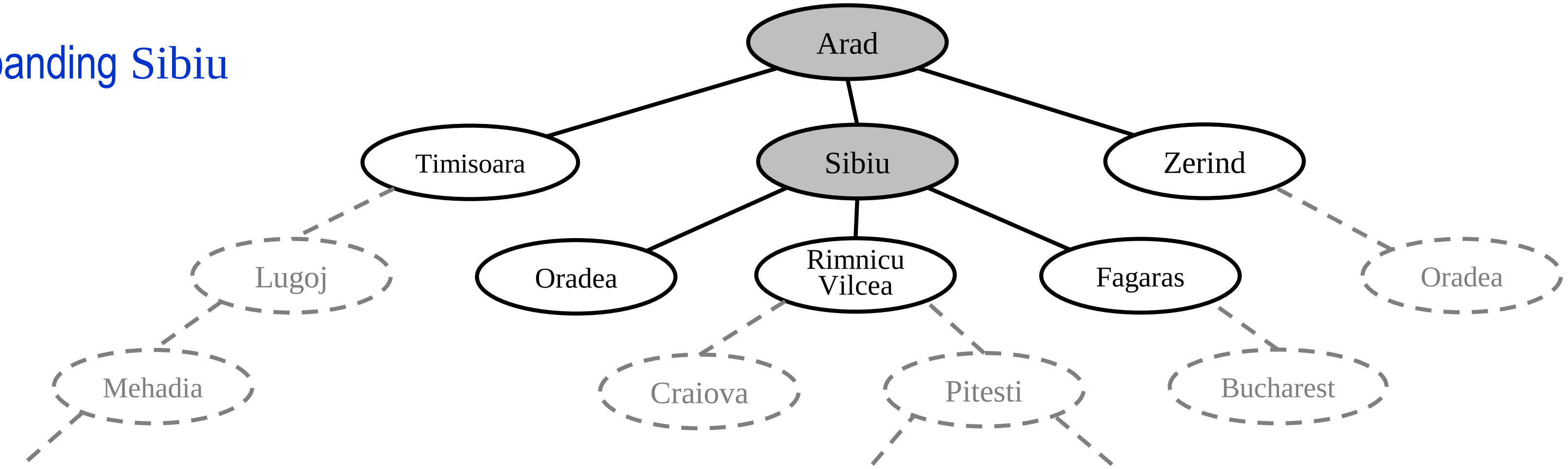
Faint dashed lines: the nodes that have not been generated.
浅虚线：表示该节点尚未生成。

Shortest Path Problem by Tree Search 采用树搜索的最短路径问题

(b) After expanding Arad

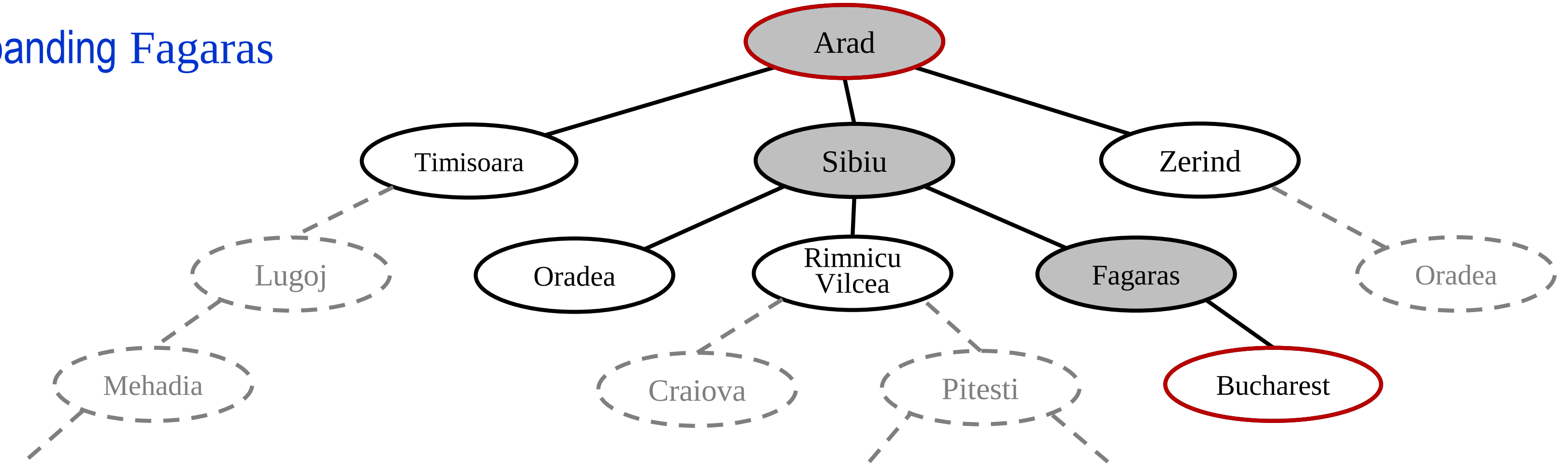


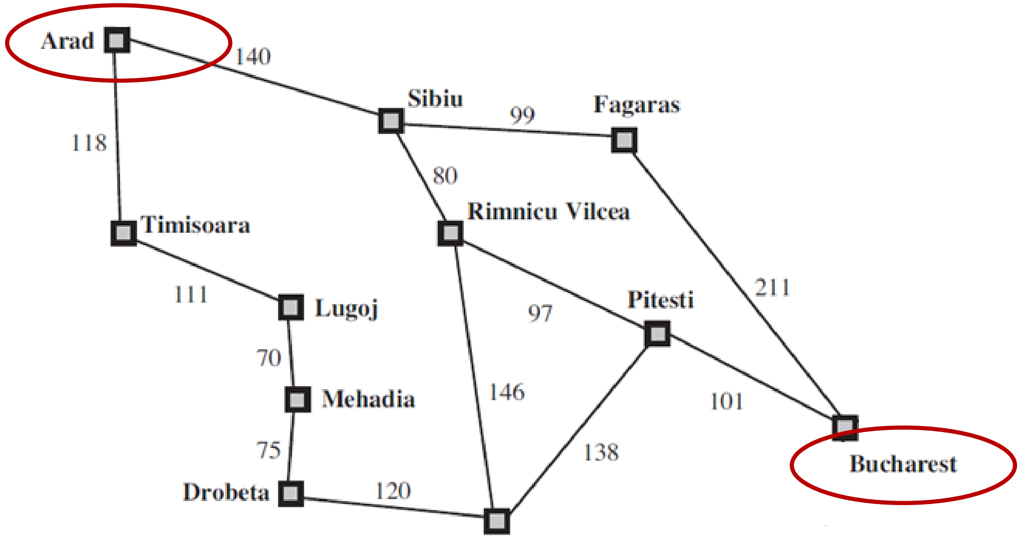
(c) After expanding Sibiu



Shortest Path Problem by Tree Search 采用树搜索的最短路径问题

(d) After expanding Fagaras





A General Tree-search Algorithm 一种通用的树搜索算法

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*

The *frontier* (also known as *open list*): an data structure, to store the set of all leaf nodes.

该*frontier*（亦称*open list*）：一种数据结构，用于存储所有的叶节点。

The process of expanding nodes on the *frontier* continues until either a solution is found or there are no more states to expand.

在*frontier*上扩展节点的过程持续进行，直到找到一个解、或没有其它状态可扩展。

A General Graph-search Algorithm 一种通用的图搜索算法

function GRAPH-SEARCH (*problem*) **returns** a solution, or failure
  initialize the *frontier* using the initial state of *problem*
  initialize the *explored* 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*
    expand the chosen node, adding the resulting nodes to the *frontier*
      only if not in the *frontier* or *explored*

The *explored* (aka *closed list*) is an data structure to remember every expanded node.

该*explored*（亦称*closed list*）：一种数据结构，用于记忆每个扩展节点。

The nodes in the *explored* or the *frontier* can be discarded.

*explored*或*frontier*中的节点可以被丢弃。

Thank you for your attention!

PoAI

Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Contents

- 3.4.1 Breadth-first Search
- 3.4.2 Uniform-cost Search
- 3.4.3 Depth-first Search
- 3.4.4 Variants of Depth-first Search
- 3.4.5 Bidirectional Search
- 3.4.6 Comparing Uninformed Search Strategies

What is Uninformed Search 什么是无信息搜索

- The uninformed search is also called blind search.

  无信息搜索也被称为盲目搜索。

- The term (uninformed, or blind) means that the search strategies have no additional information about states beyond that provided in the problem definition.

  该术语（无信息、盲目的）意味着该搜索策略没有超出问题定义提供的状态之外的附加信息。

- All they can do is to generate successors and distinguish a goal state from a non-goal state.

  所有能做的就是生成后继结点，并且从区分一个目标状态或一个非目标状态。

What is Uninformed Search 什么是无信息搜索

- All search strategies are distinguished by the *order* in which nodes are expanded.
  所有的搜索策略是由节点扩展的顺序加以区分。
- The search strategies: breadth-first, depth-first, and uniform-cost search.
  这些搜索策略是：宽度优先、深度优先、以及一致代价搜索。

Uninformed Search Strategy Evaluation 无信息搜索策略评价

- An uninformed search strategy is defined by picking the order of node expansion.
  一种无信息搜索策略是通过其选择节点扩展的顺序来定义的。
- The strategies can be evaluated along the following properties:
  其策略可按照如下特性来评价：
  - Completeness: Does it always find a solution if one exists?
    完备性：是否总能找到一个存在的解？
  - Time complexity: How long does it take to find a solution?
    时间复杂性：花费多长时间找到这个解？
  - Space complexity: How much memory is needed?
    空间复杂性：需要多少内存？
  - Optimality: Does it always find the optimal solution?
    最优性：是否总能找到最优的解？

Uninformed Search Strategy Evaluation 无信息搜索策略评价

- Time complexity and space complexity are measured in following terms:
  时间复杂性和空间复杂性用如下术语来度量：
  - b -- maximum branching factor of the search tree.
    搜索树的最大分支因子。
  - d -- depth of the shallowest solution.
    最浅解的深度。
  - m -- maximum depth of the search tree.
    搜索树的最大深度。

Breadth-first Search 宽度优先搜索

- Search Strategy 搜索策略

  Expand shallowest unexpanded node.

  扩展最浅的未扩展节点。

- Implementation 实现方法

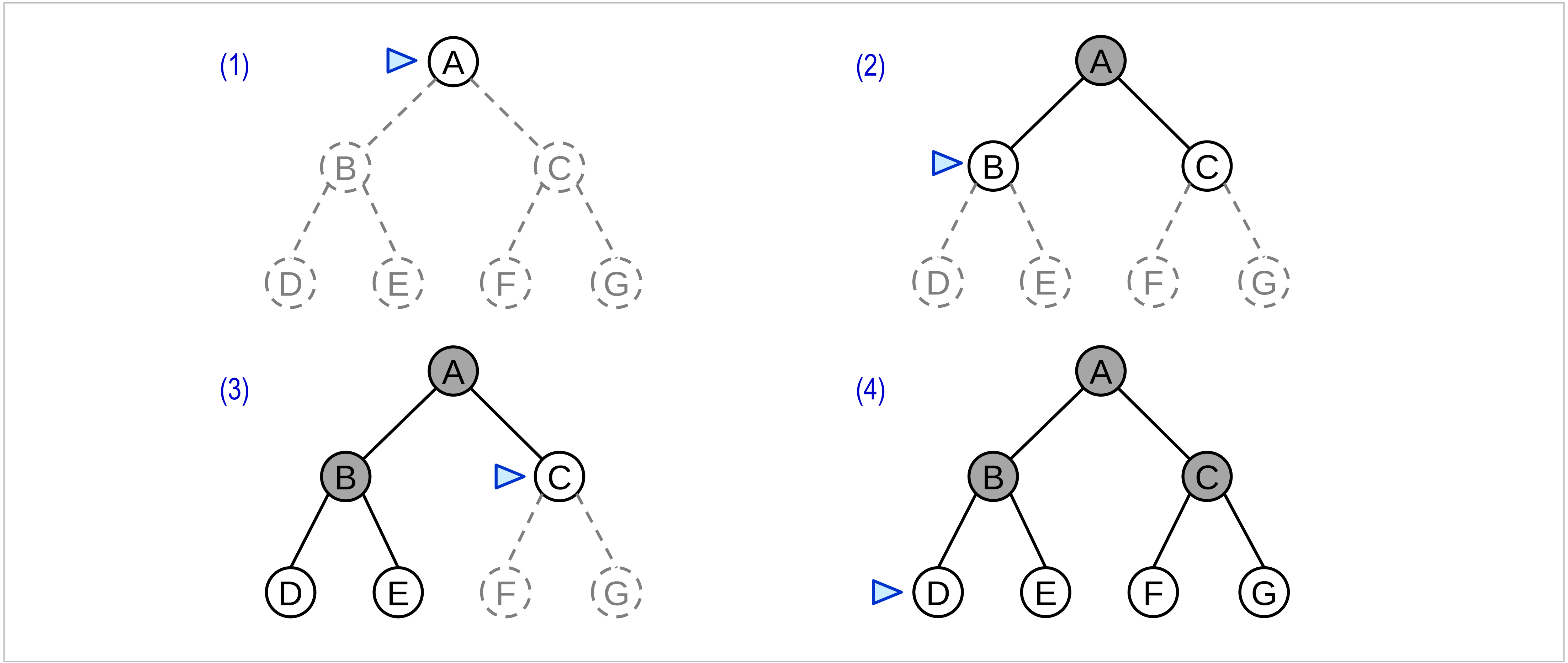
  Use FIFO (First-In First-Out) queue, i.e., new successors go at end.

  使用FIFO队列，即新的后继节点放在后面。

Breadth-first Search Algorithm on a Graph 图的宽度优先搜索算法

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE
  PATH-TEST = 0
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
    if EMPTY ? (frontier) then return failure
    node ← POP(frontier)          /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored or frontier then
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Breadth-first Search on a Simple Binary Tree 简单二叉树的宽度优先搜索



At each stage the node to be expanded next is indicated by a marker.

Properties of Breadth-first Search 宽度优先搜索的性质

- Time complexity 时间复杂性 $b + b^2 + b^3 + \ldots + b^d = O(b^d)$

- Space complexity 空间复杂性 $O(b^d)$

where

- b -- the branching factor
  分枝因子
- d -- the depth of the shallowest solution
  最浅解的深度

Time and Memory Requirements 时间和内存需求

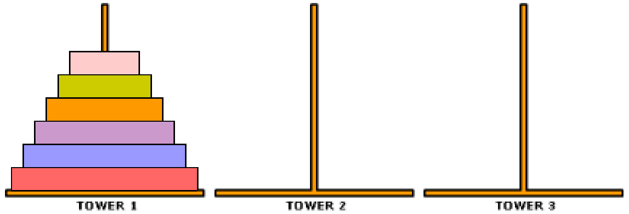
Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

Assume: branching factor b = 10; 1 million nodes/second; 1000 bytes/node.

- Memory requirements are a bigger problem, execution time is still a major factor.
  内存的需求是一个很大的问题，而执行时间仍是一个主要因素。
- Breadth-first search cannot solve exponential complexity problems but small branching factor.
  宽度优先搜索不能解决指数复杂性的问题，小的分支因子除外。

Example: Tower of Hanoi 汉诺塔问题

- It is said that there is an Indian temple which contains 3 towers by 64 golden disks.
  据说一个印度寺庙里有3个塔、塔上有64个金盘。
- Priests have been moving these disks, sample rules:
  祭司一直在移动这些金盘，规则很简单：
  - Only one disk can be moved at a time.
    每次仅能移动一个金盘。
  - A disk can only be moved if it is uppermost disk.
    仅能移动最上面的那块金盘。
  - No disk may be placed on top of a smaller disk.
    大的金盘不能放在小的金盘上面。
- According to the legend, the world will end when last move.
  据传说，当最后一次移动金盘时，世界将会毁灭。



Assume:
moving 1 disk/second;
it will take $2^{64}-1$ seconds
$\approx$ 585 billion years.

Thank you for your attention!

PoAI

Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Uniform-cost Search 一致代价搜索

□ Search Strategy 搜索策略

Expand lowest-cost unexpanded node.

扩展最低代价的未扩展节点。

□ Implementation 实现方法

Queue ordered by path cost, lowest first.

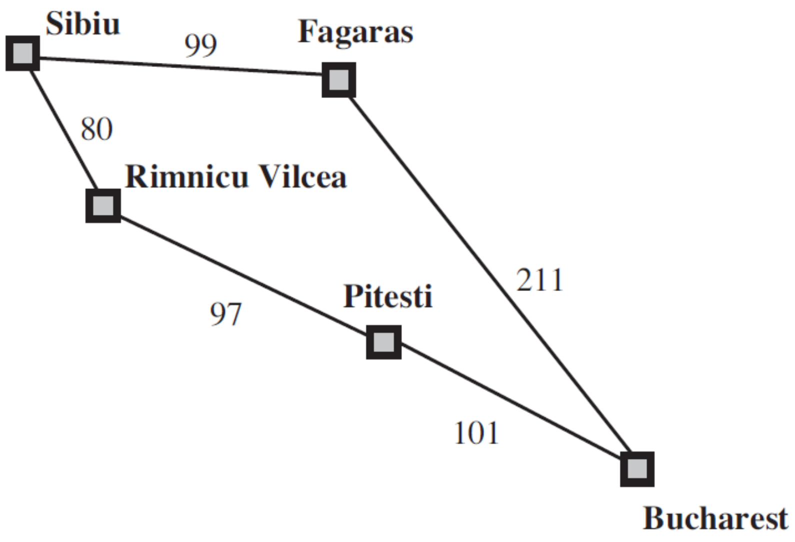
队列，按路径代价排序，最低优先。

Uniform-cost Search Algorithm 一致代价搜索算法

```
function UNIFORM-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-TEST = 0
    frontier ← a priority queue ordered by PATH-COST, with node as the only element
    explored ← an empty set
    loop do
        if EMPTY ? (frontier) then return failure
        node ← POP(frontier)          /* chooses the lowest-cost node in frontier */
        if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                frontier ← INSERT(child, frontier)
            else if  child.STATE is in frontier with higher PATH-COST then
                replace that frontier node with child
```

Example: From Sibiu to Bucharest 举例：从Sibiu到Bucharest

- From Sibiu to Bucharest, least-cost node, Rimnicu Vilcea, is expanded, next, adding Pitesti with cost 80 + 97 = 177.
  从Sibiu到Bucharest，扩展最低代价节点Rimnicu Vilcea，然后加上Pitesti的代价
- The least-cost node is now Fagaras, and adding goal node Bucharest with cost 99 + 211 = 310.
  现在最低代价节点为Fagaras，加上目标节点Bucharest的代价
- Choosing Pitesti and adding a second path to Bucharest with cost 177 + 101 = 278.
  选择Pitesti并加上第二条路径到Bucharest的代价
- This new path is better, so lowest path cost is 278.
  这条新路径较好，故最低路径代价为 278.



Properties of Uniform-cost Search 一致代价搜索的特性

- Time complexity $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

  时间复杂性

- Space complexity $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

  空间复杂性

where

- b -- the branching factor

  分支因子

- C^* -- the cost of the optimal solution

  最优解的代价

- ϵ -- every action costs at least

  至少每个动作的代价

Thank you for your attention!

Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Depth-first Search 深度优先搜索

Search Strategy 搜索策略

Expand deepest unexpanded node.

扩展最深的未扩展节点。

- Note: breadth-first-search expands shallowest unexpanded node.

  注意：宽度优先搜索扩展最浅的未扩展节点。

Implementation 实现方法

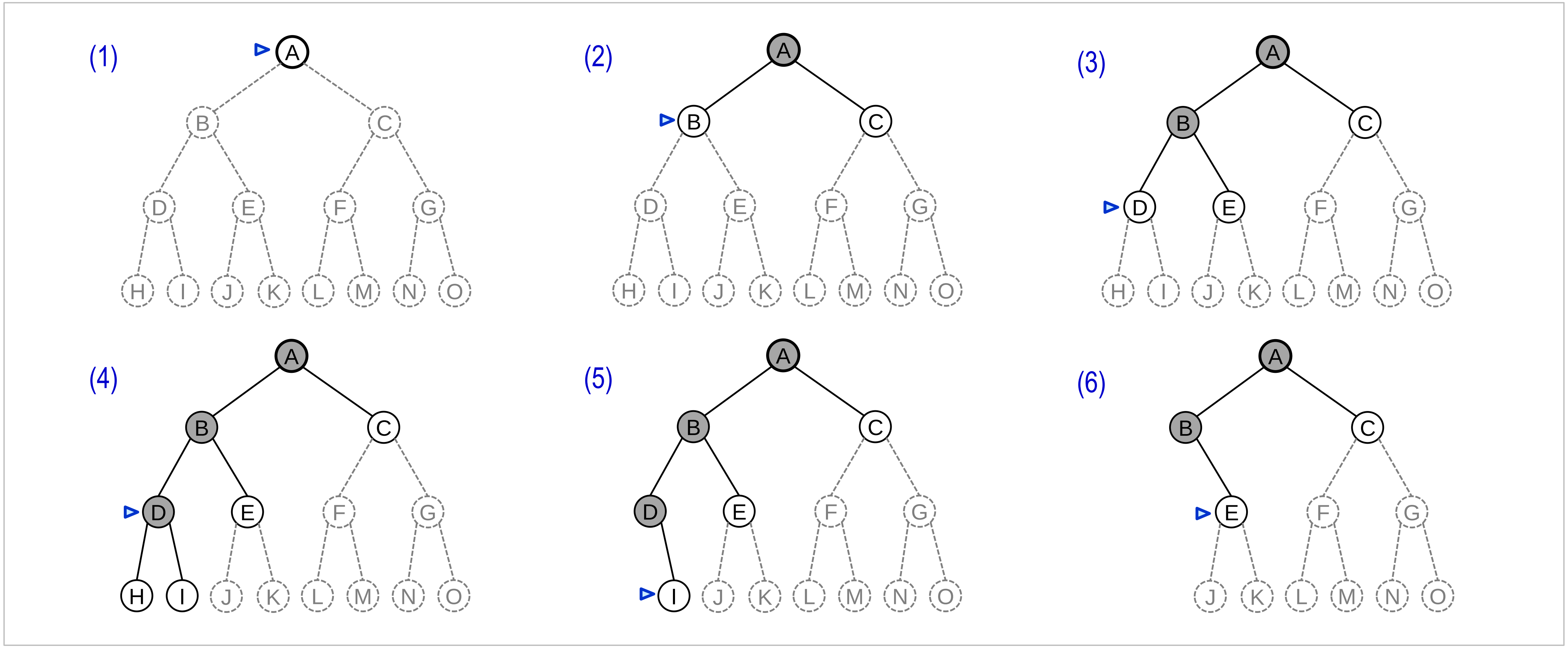
Use LIFO queue, put successors at front.

使用LIFO 队列，把后继节点放在队列的前端。

- Note: breadth-first-search uses a FIFO queue

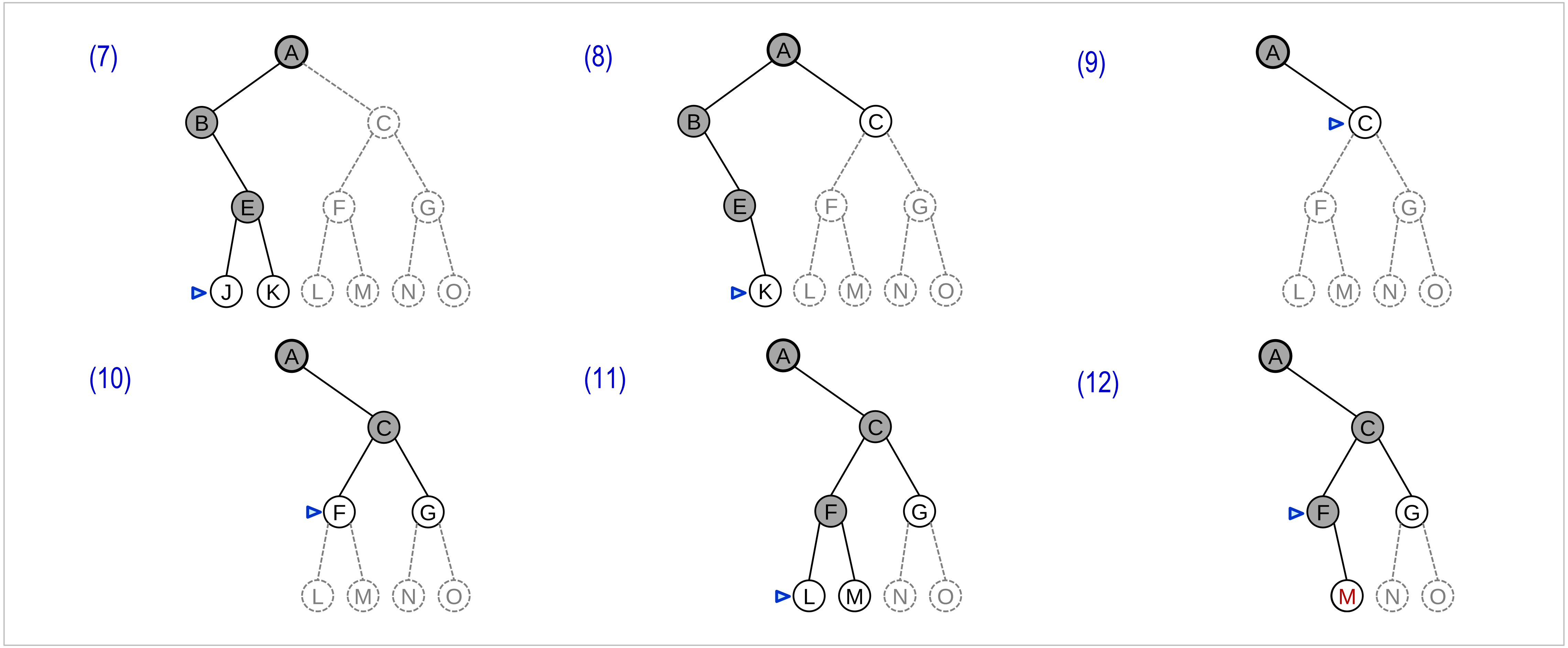
  注意：宽度优先搜索使用FIFO队列。

Depth-first Search on a Simple Binary Tree 简单二叉树的深度优先搜索



Explored nodes with no descendants are removed from memory.

Depth-first Search on a Simple Binary Tree 简单二叉树的深度优先搜索



Nodes at depth 3 have no successors, M is the only goal node.

Properties of Depth-first Search 深度优先搜索的特性

- Time complexity $O(b^m)$

  时间复杂性

- Space complexity $O(bm)$

  空间复杂性

where

- b -- the branching factor

  分支因子

- m -- the maximum depth of any node

  任一节点的最大深度

Thank you for your attention!

PoAI

Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Contents

- 3.4.4 Depth-limited Search
- 3.4.4 Iterative Deepening Search

1) Depth-limited Search 深度受限搜索

- The failure of depth-first search will be happened if in infinite state spaces.
  若状态空间无限，深度优先搜索就会发生失败。
- This problem can be solved with a predetermined depth limit l, i.e. nodes at depth l are treated as if they have no successors.
  这个问题可以用一个预定的深度限制 l得到解决，即：深度 l以外的节点被视为没有后继节点。
- Disadvantages
  缺点
  - It will introduces an additional source of incompleteness if we choose $l < d$, that is, the shallowest goal is beyond the depth limit.
    如果我们选择 $l < d$，即最浅的目标在深度限制之外，这种方法就会出现额外的不完备性。
  - Depth-limited search will also be non-optimal if we choose $l > d$.
    如果我们选择 $l > d$，深度受限搜索也将是非最优的。

Depth-limited Search Algorithm 深度受限搜索算法

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    if limit = 0 then return cutoff     /* no solution */
    cutoff_occurred ? ← false
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        result ← RECURSIVE-DLS(child, problem, limit − 1)
        if result = cutoff then cutoff_occurred ? ← true
        else if result ≠ failure then return result
    if cutoff_occurred ? then return cutoff  /* no solution */
    else return failure
```

A recursive implementation of depth-limited tree search

2) Iterative Deepening Search 迭代加深搜索

- It combines the benefits of depth-first and breadth-first search, running repeatedly with gradually increasing depth limits until the goal is found.
  它将深度优先和宽度优先的优势相结合，逐步增加深度限制反复运行直到找到目标。
- It visits the nodes in the search tree in the same order as depth-first search, but the cumulative order in which nodes are first visited is effectively breadth-first.
  它以深度优先搜索相同的顺序访问搜索树的节点，但先访问节点的累积顺序实际是宽度优先。

function ITERATIVE-DEEPENING-SEARCH (*problem*) **returns** a solution, or failure
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ cutoff **then return** *result*

It repeatedly applies *depth* limited search with increasing limits, in which it calls DEPTH-LIMITED-SEARCH algorithm.

Thank you for your attention!

PoAI

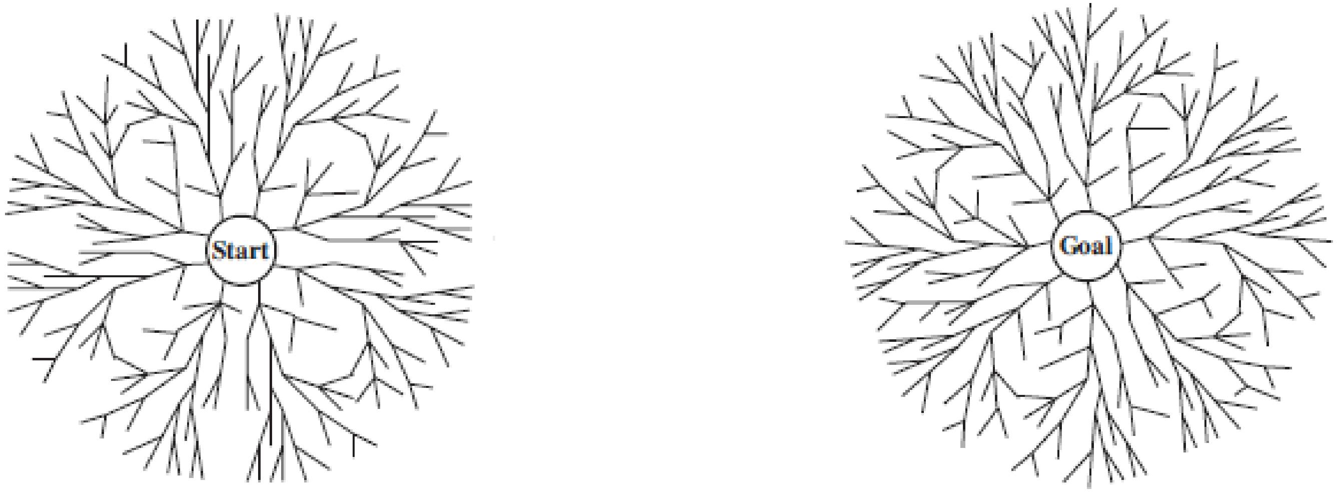
Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Bidirectional Search 双向搜索

- It runs two simultaneous searches: one forward from the *initial state*, and another backward from the *goal*. It stops when the two meet in the middle.
  它同时进行两个搜索：一个是从初始状态向前搜索，而另一个则从目标向后搜索。当两者在中间相遇时停止。



- This method can be guided by a heuristic estimate of the remaining distance.
  该方法可以通过一种剩余距离的启发式估计来导向。

Thank you for your attention!

PoAI

Uninformed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Evaluation of Uninformed Tree-search Strategies

无信息树搜索策略评价

Criterion	Breadth First	Uniform Cost	Depth First	Depth Limited	Iterative Deepening	Bidirectional
Complete	Yes^a	$\text{Yes}^{a,b}$	No	No	Yes^a	$\text{Yes}^{a,d}$
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	$\text{Yes}^{c,d}$

Where

- b -- maximum branching factor of the tree
- d -- depth of the shallowest solution
- m -- maximum depth of the tree
- l -- the depth limit

- a -- complete if b is finite
- b -- complete if step costs ϵ for positive
- c -- optimal if step costs are all identical
- d -- if both directions use breadth-first search

Thank you for your attention!

PoAI

Informed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

What is Informed Search 什么是有信息搜索

- Also known as Heuristic Search.

  亦被称为启发式搜索。

- The strategies use problem-specific knowledge beyond the definition of the problem itself, so that can find solutions more efficiently than can an uninformed strategy.

  这类策略采用超出问题本身定义的、问题特有的知识，因此能够找到比无信息搜索更有效的解。

- The general approaches use one or both of following functions:

  一般方法使用如下函数中的一个或两者：

  - An evaluation function, denoted $f(n)$, used to select a node for expansion.

    评价函数，记作 $f(n)$，用于选择一个节点进行扩展。

  - A heuristic function, denoted $h(n)$, as a component of f.

    启发式函数，记作 $h(n)$，作为 f 的一个组成部分。

Contents

- 3.5.1 Best-first Search
- 3.5.2 Greedy Search
- 3.5.3 A* Search
- 3.5.4 Iterative Deepening A* Search

Best-first Search 最佳优先搜索

☐ Search Strategy 搜索策略

- A node is selected for expansion based on an evaluation function, $f(n)$.
  搜索策略：一个节点被选择进行扩展是基于一个评价函数，$f(n)$。
- Most best-first algorithms also include a heuristic function, $h(n)$.
  大多数的最佳优先算法还包含一个启发式函数，$h(n)$。

☐ Implementation 实现方法

- Identical to that for uniform-cost search.
  实现方法：与一致代价搜索相同。
- However best-first search uses of $f(n)$ instead of $g(n)$ to order the priority queue.
  然而，最佳优先搜索使用 $f(n)$ 代替 $g(n)$ 来整理优先队列。

Best-first Search 最佳优先搜索

- Heuristic function 启发式函数

  $h(n)$ = estimated cost of the cheapest path from the state at node n to a goal state.

  $h(n)$ = 从节点n到目标状态的最低路径估计代价。

- Special cases 特例
  - Greedy Search

    贪婪搜索
  - A* search

    A*搜索

Greedy Search 贪婪搜索

Search Strategy 搜索策略

- Try to expand the node that is closest to the goal.
  试图扩展最接近目标的节点。

Evaluation function 评价函数

$$f(n) = h(n)$$

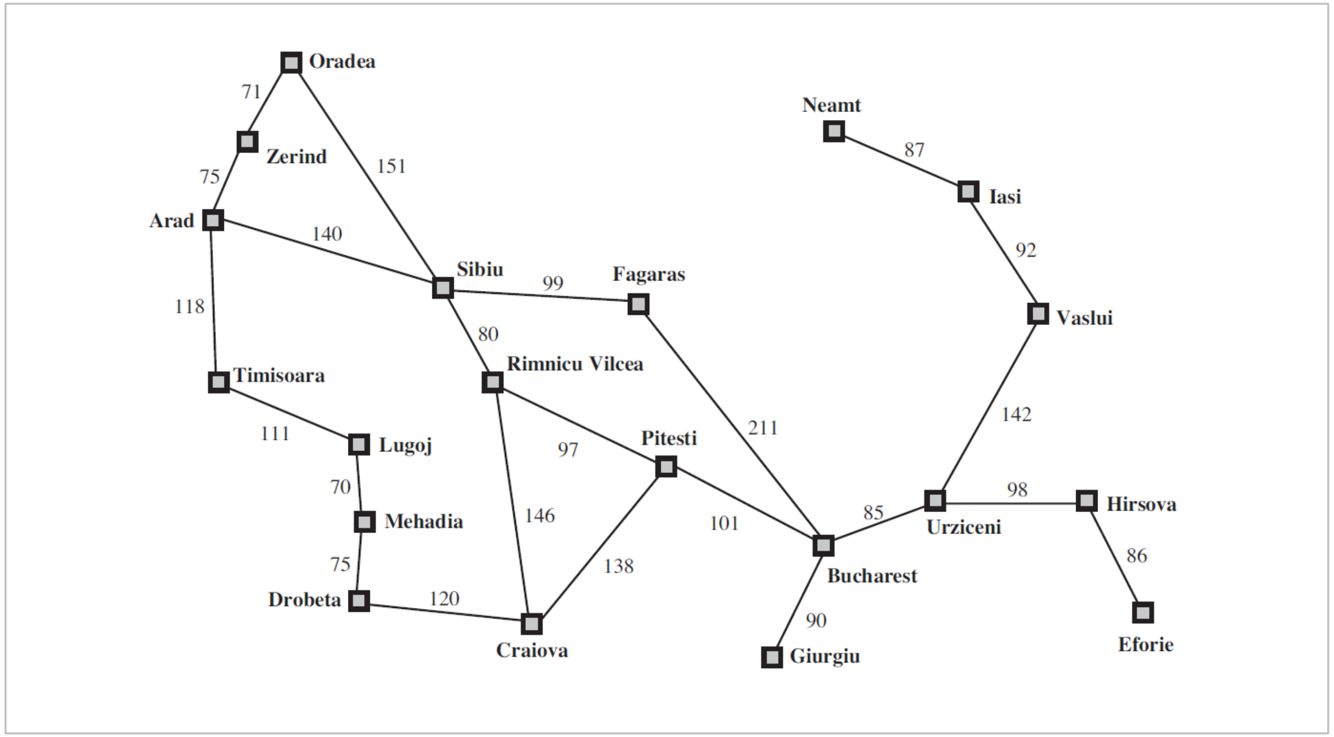
- It evaluates nodes by using just the heuristic function.
  它仅使用启发式函数对节点进行评价。
- $h(n)$ -- estimated cost from n to the closest goal.
  $h(n)$ -- 从n到最接近目标的估计代价。

Why call “greedy” 为什么称为“贪婪”

- at each step it tries to get as close to the goal as it can.
  每一步它都试图得到能够最接近目标的节点。

Example: from Arad to Bucharest 举例：从Arad到Bucharest

- h_{SLD} : straight-line distance 直线距离



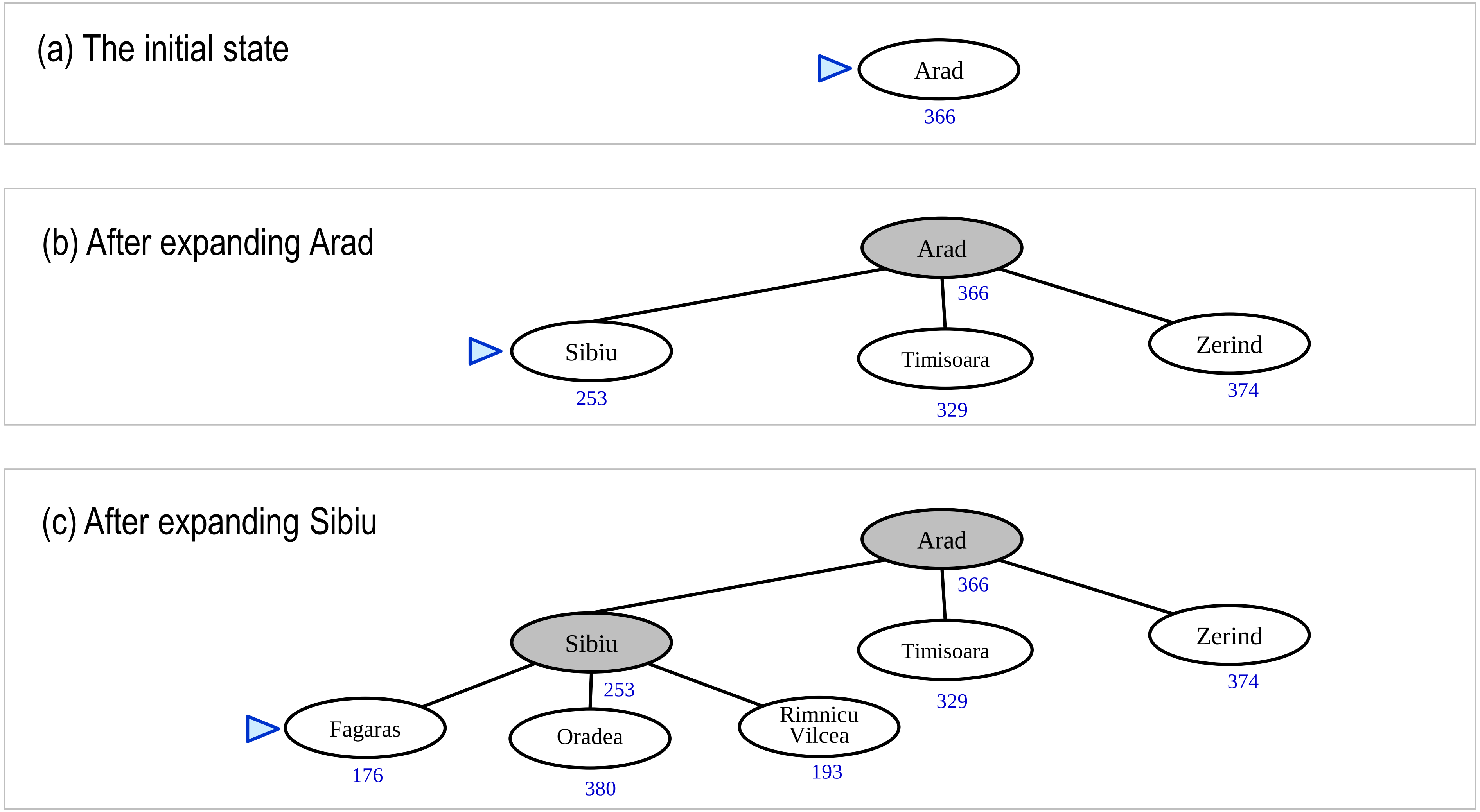
h_{SLD} Values

Arad	366	Mehadia	241
Bucharest	0	Neamt	234
Craiova	160	Oradea	380
Drobeta	242	Pitesti	100
Eforie	161	Rimnicu Vilcea	193
Fagaras	176	Sibiu	253
Giurgiu	77	Timisoara	329
Hirsova	151	Urziceni	80
Iasi	226	Vaslui	199
Lugoj	244	Zerind	374

Notice: the values of h_{SLD} cannot be computed from the problem description itself. Moreover, it takes a certain amount of experience to know that h_{SLD} is correlated with actual road distances and therefore is a useful heuristic.

注意：h_{SLD}的值无法从问题描述本身来计算。此外，它要积累一定的经验才能知道，h_{SLD}与实际道路的距离相关，因此是一个有用的启发。

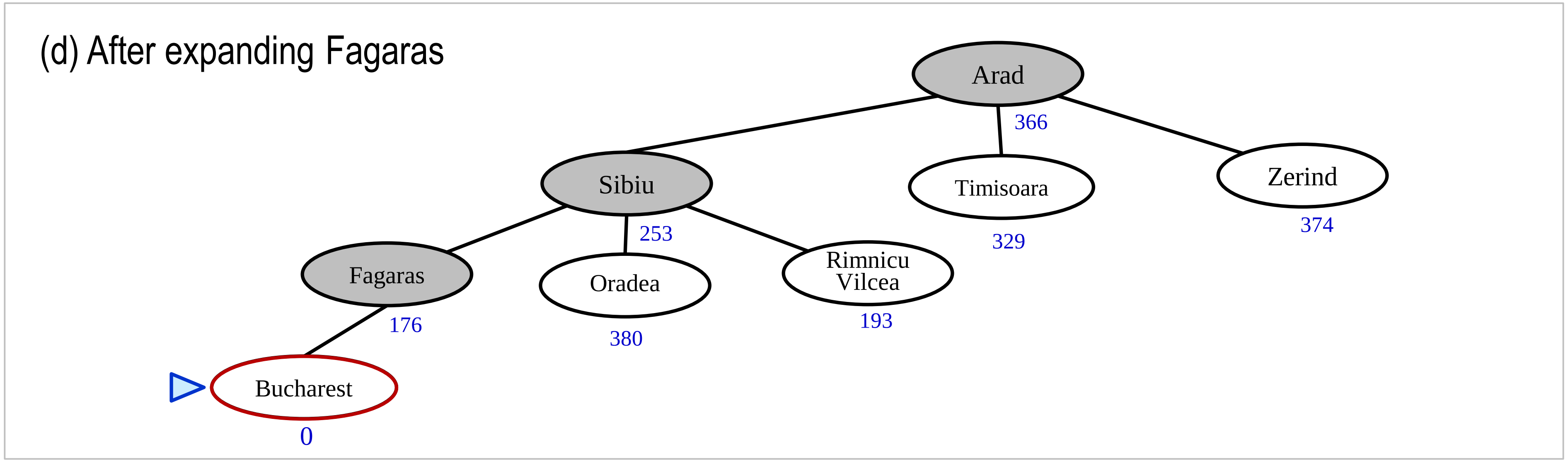
Example: from Arad to Bucharest 举例：从Arad到Bucharest



h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Example: from Arad to Bucharest 举例：从Arad到Bucharest

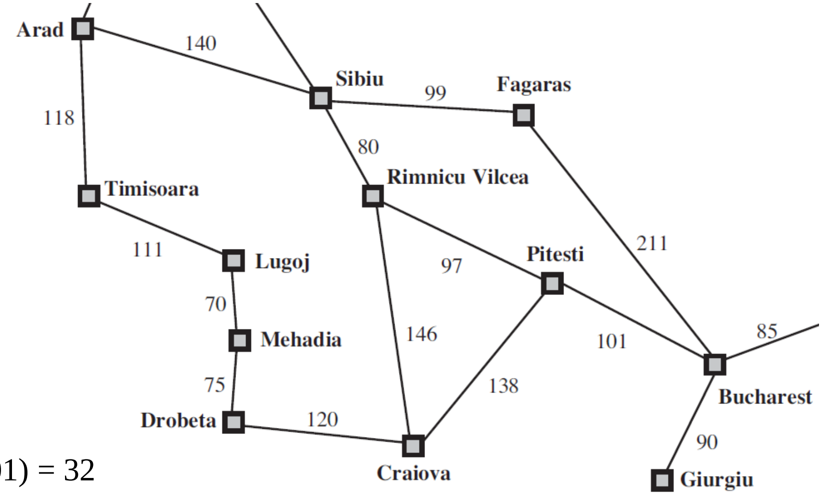


h_{SLD} Values

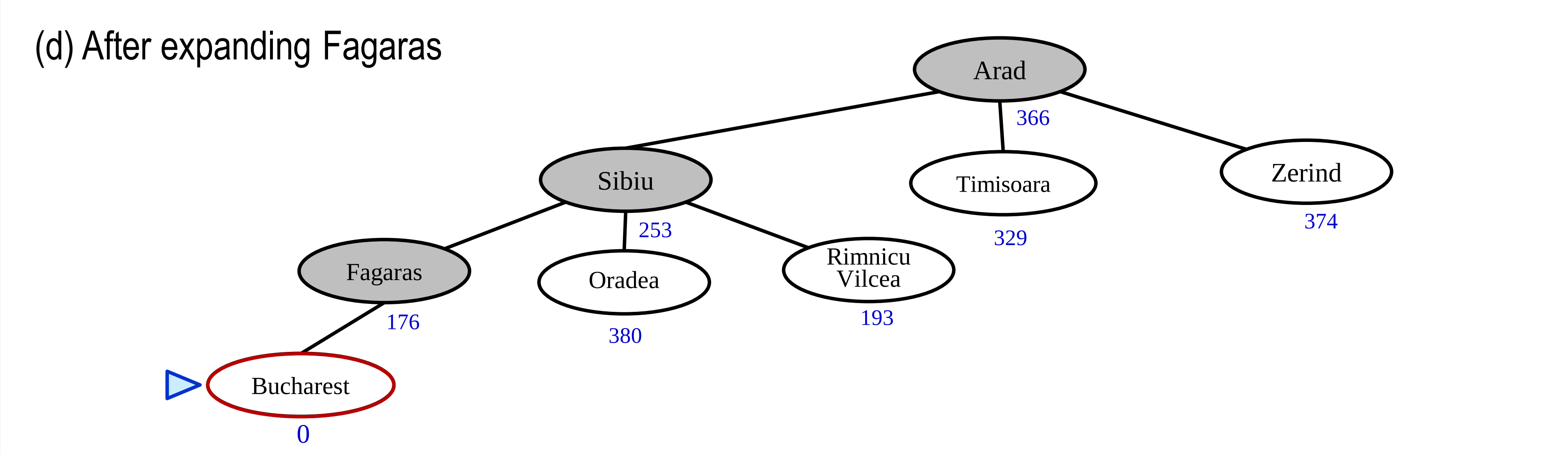
Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Notice: For this particular problem, it uses h_{SLD} to find a solution, hence its search cost is minimal. However it is not optimal: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

(140+99+211) – (140+80+97+101) = 32



Example: from Arad to Bucharest 举例：从Arad到Bucharest



h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Notice: For this particular problem, it uses h_{SLD} to find a solution, hence its search cost is minimal. However it is not optimal: the path via Sibiu and Fagaras to Bucharest is 32 kilometers longer than the path through Rimnicu Vilcea and Pitesti.

注意：对这个具体问题，它采用h_{SLD}找到解，因此搜索代价是最小的。然而它不是最优的：如果计算路径代价的话，这条经由Sibiu和Fagaras到Bucharest的路径比经过Rimnicu Vilcea 和Pitesti远32公里。

(140+99+211) – (140+80+97+101) = 32

Properties of Greedy Tree Search 贪婪树搜索的特性

- Worst-case time: $O(b^m)$
  最差情况下的时间
- Space complexity: $O(b^m)$
  空间复杂性

where

- b -- the branching factor
  分支因子
- m -- the maximum depth of the search space
  搜索空间的最大深度

Thank you for your attention!

PoAI

Informed Search Strategies

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Contents

- 3.5.1 Best-first Search
- 3.5.2 Greedy Search
- 3.5.3 A* Search
- 3.5.4 Iterative Deepening A* Search

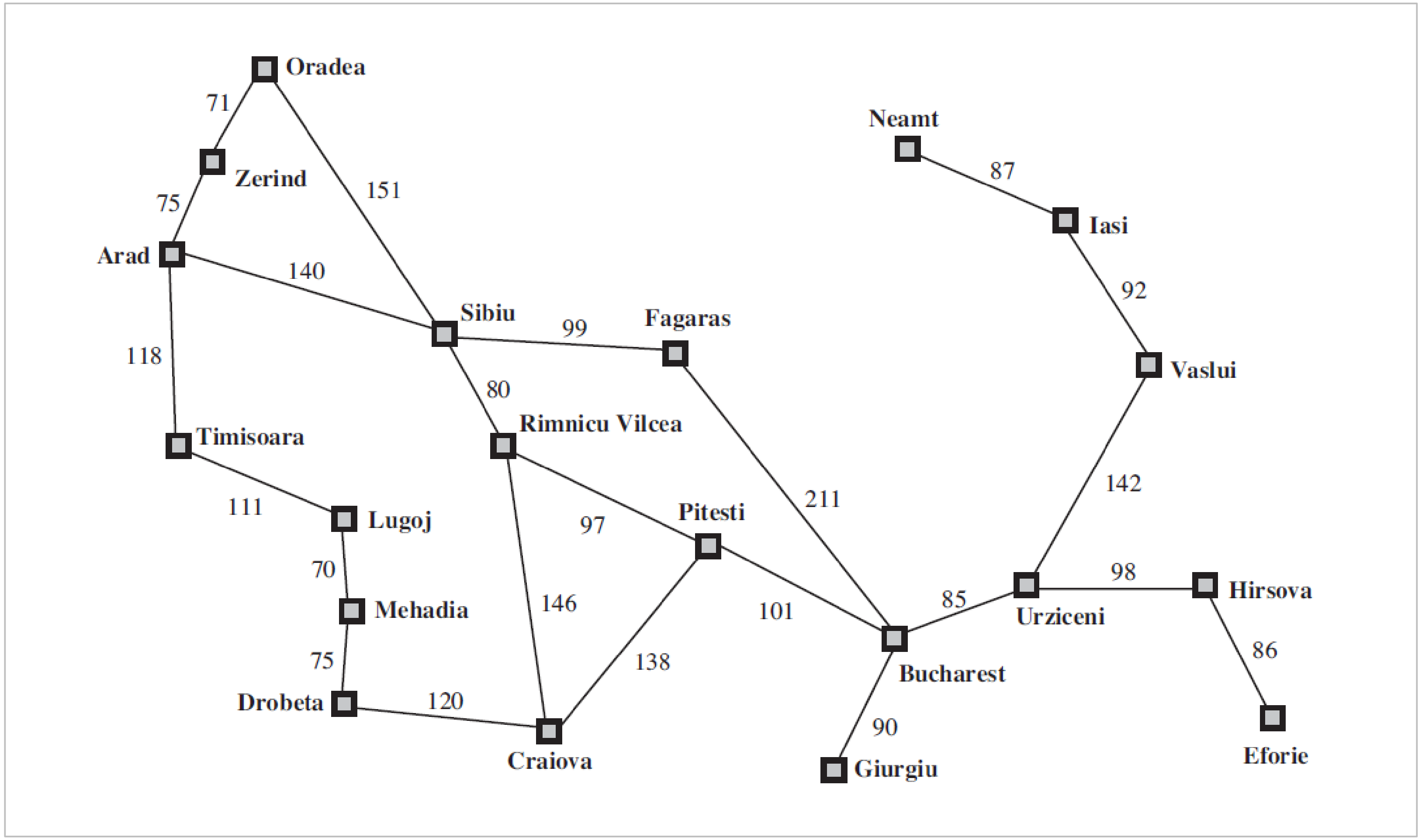
A* Search A*搜索

- Search Strategy 搜索策略
  - avoid expanding expensive paths, minimizing the total estimated solution cost.
    避免扩展代价高的路径，使总的估计求解代价最小化。
- Evaluation function 评价函数
  - $g(n)$ -- cost to reach the node
    到达该节点的代价
  - $h(n)$ -- estimated cost to get from the node to the goal
    从该节点到目标的估计代价

$$f(n) = g(n) + h(n)$$

- Theorem: A* search is optimal
  定理：A*搜索是最优的

Example: Form Arad to Bucharest 举例：从Arad到Bucharest



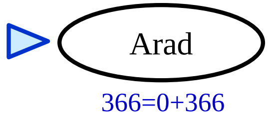
h_{SLD} Values

Arad	366
Bucharest	0
Craiova	160
Drobeta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	100
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

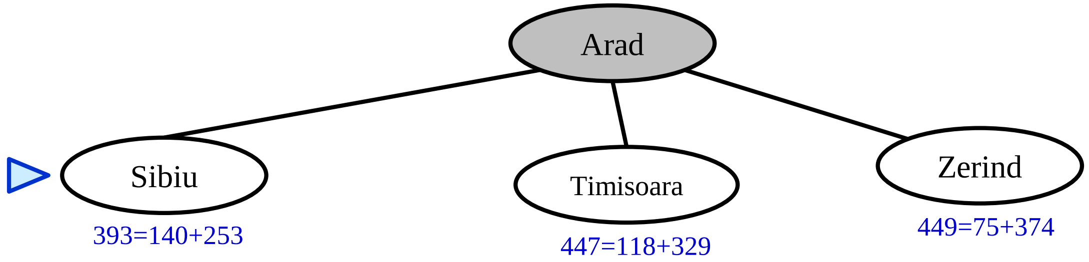
$f(n) = g(n) + h(n)$, which $g(n)$ = path cost, $h(n) = h_{SLD}$

Example: Form Arad to Bucharest 举例：从Arad到Bucharest

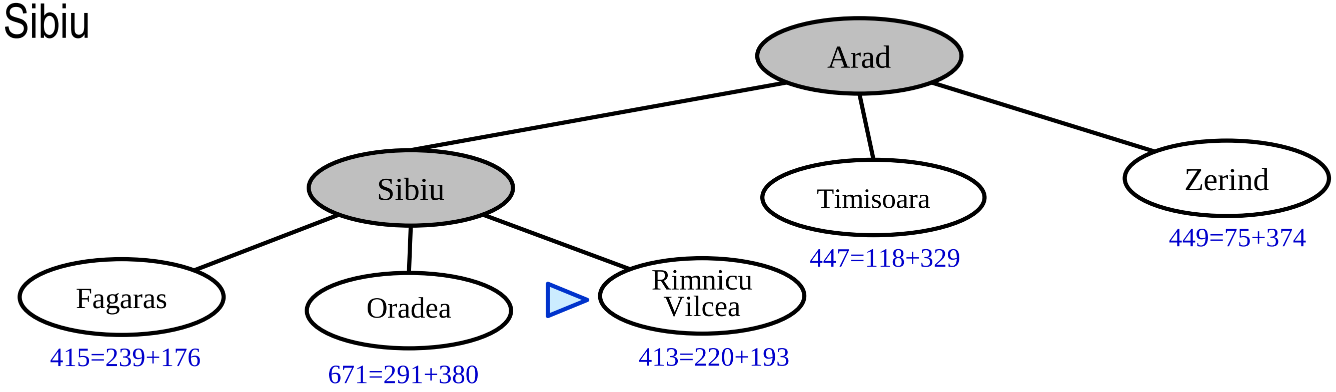
(a) The initial state



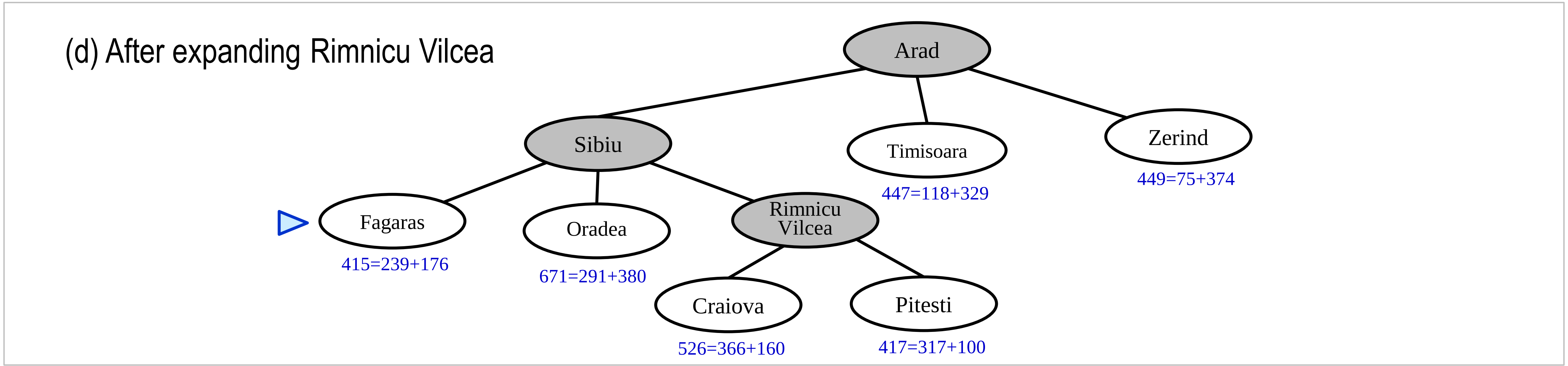
(b) After expanding Arad

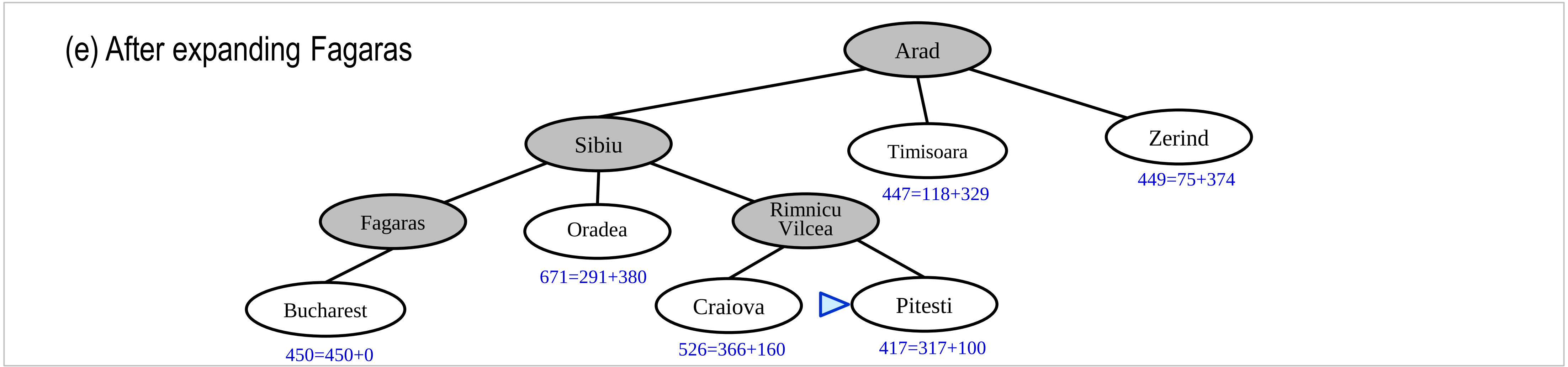


(c) After expanding Sibiu



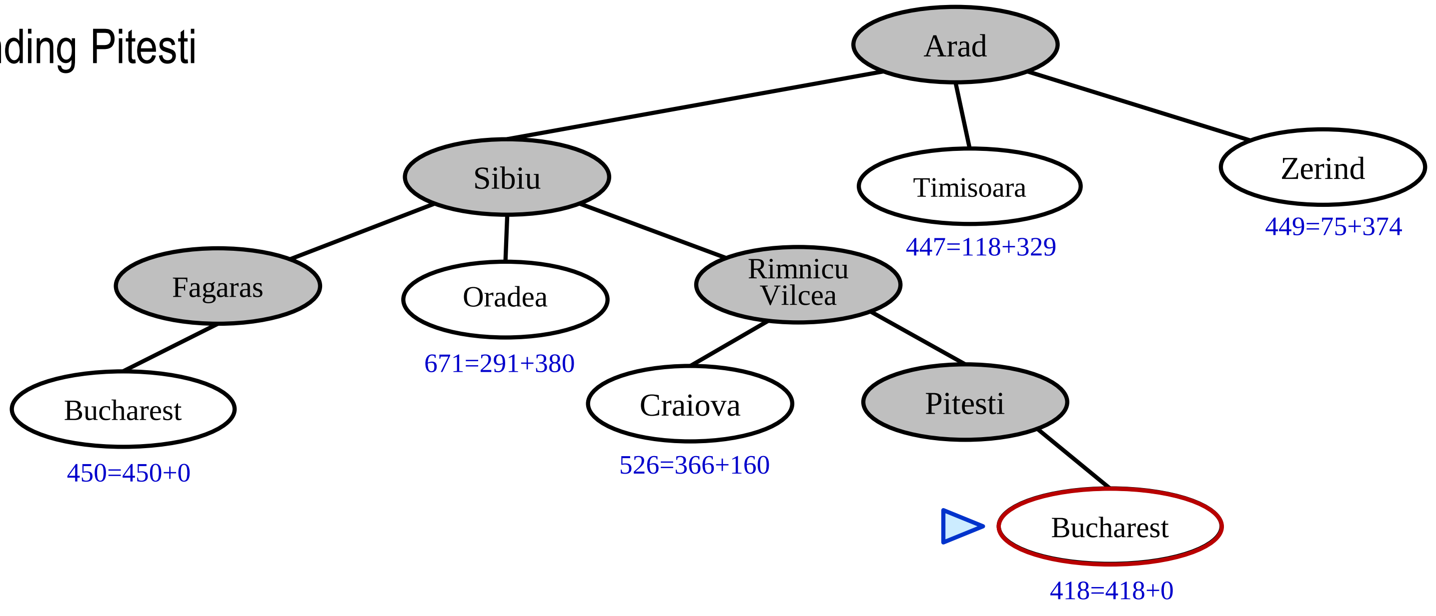
Example: Form Arad to Bucharest 举例：从Arad到Bucharest





Example: Form Arad to Bucharest 举例：从Arad到Bucharest

(f) After expanding Pitesti



Iterative Deepening A* Search 迭代加深A*搜索

- It is a variant of iterative deepening depth-first search
  它是迭代加深深度优先搜索的变种
  - that borrows the idea to use a heuristic function to evaluate the remaining cost to get to the goal from the A* search algorithm.
    从A*搜索算法借鉴了这一思想，即使用启发式函数来评价到达目标的剩余代价。
- Since it is a depth-first search algorithm, its memory usage is lower than in A*
  因为它是一种深度优先搜索算法，内存使用率低于A*算法
  - but unlike standard iterative deepening search, it concentrates on exploring the most promising nodes and thus doesn’t go to the same depth everywhere in the search tree.
    但是，不同于标准的迭代加深搜索，它集中于探索最有希望的节点，因此不会去搜索树任何处的同样深度。

Comparing Iterative Deepening Search 迭代加深搜索之比较

- Iterative deepening depth-first search
  - uses search depth as the cutoff for each iteration.
    迭代加深深度优先搜索：使用搜索深度作为每次迭代的截止值。
- Iterative Deepening A* Search
  - uses the more informative evaluation function, i.e.
    迭代加深A*搜索：使用信息更丰富的评价函数，即

$$f(n) = g(n) + h(n)$$

where

- $g(n)$ -- cost to reach the node
  到达该节点的代价。
- $h(n)$ -- estimated cost to get from the node to the goal
  该节点到目标的估计代价

Thank you for your attention!

PoAI

Heuristic Functions

School of Electronic and Computer Engineering
Peking University

Wang Wenmin

Contents

- 3.6.1 Heuristics for 8-puzzle
- 3.6.2 Search Cost

Heuristics for 8-puzzle 8数码难题的启发式

- To find shortest solutions by using A* , need a heuristic function that are two commonly used candidates.

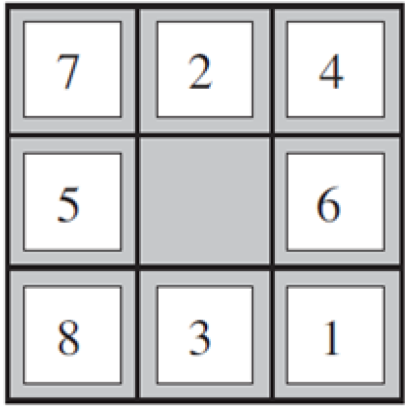
  要用A*算法找到最短距离的解，需要一个启发式函数，通常有两个候选。

h_1 = *number of misplaced tiles* = 8.

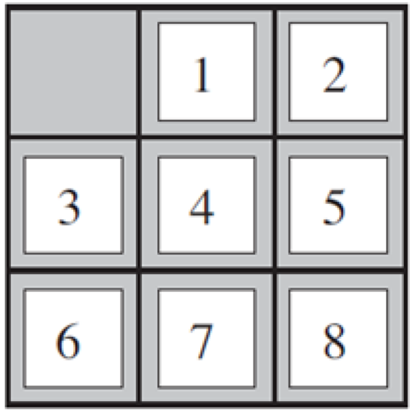
错位棋子的个数

h_2 = *total Manhattan distance* (*tiles from desired locations*) = 3+1+2+2+2+3+3+2 = 18

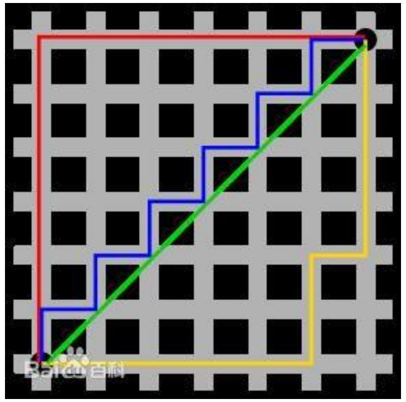
曼哈顿距离之和（每个棋子到目标位置）



Start state



Goal state



Manhattan distance

Search Cost 搜索代价

Search Cost (nodes generated)

d (depth)	Iterative Deepening Search	$A^*(h_1)$	$A^*(h_2)$
2	10	6	6
4	112	13	12
6	680	20	18
8	6384	39	25
10	47127	93	39
12	3644035	227	73
14	-	539	113
16	-	1301	211
18	-	3056	363
20	-	7276	676
22	-	18094	1219
24	-	39135	1641

- If $h_2(n) \geq h_1(n)$ for all n, then h_2 dominates h_1 and is better for search.
  若对于所有的n，$h_2(n) \geq h_1(n)$，则h_2优于h_1，因而h_2更适合搜索。

Thank you for your attention!

PoAI