人工智能原理与算法

第3章 通过搜索进行问题求解

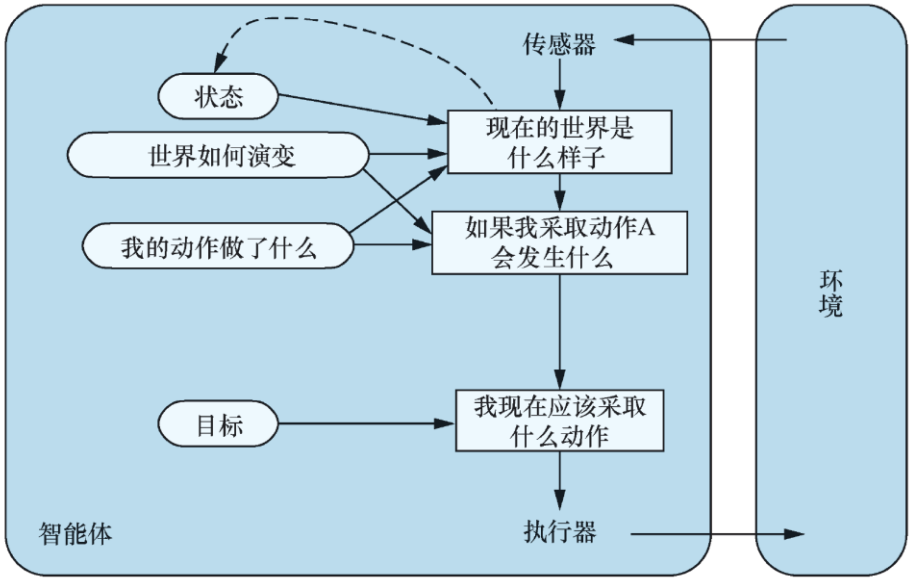
中国科学院自动化研究所
国科大人工智能学院
雷 震

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.1 概述 –回顾Agent程序

- **4种基本的Agent程序：**
  - 简单反射Agent
  - 基于模型的反射Agent
  - **基于目标的Agent**
  - 基于效用的Agent

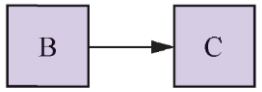


基于模型、基于目标Agent。

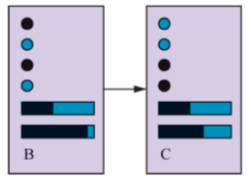
- 反射Agent *vs.* 基于目标的Agent：
  - 反射Agent存有状态-行动直接映射表，它的行为取决于这种映射。其占用存储空间大，查表耗时，不可行；
  - 基于目标的Agent考虑将要采取的行动及行动的可能后果，即与目标还有多远。

3.1 概述 –回顾Agent程序

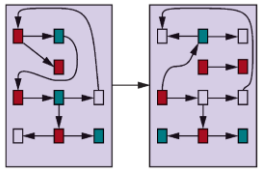
- **Agent程序如何工作（程序的组件分类）：**
  - 原子表示（atomic）：世界的每个状态是不可见的，它没有内部结构。



    - 搜索和博弈论，隐马尔科夫模型，马尔可夫决策过程。
  - 因子化表示（Factored)：将状态表示唯变量或特征的集合，每个变量或特征都可能有值。



    - 约束满足算法，命题逻辑，规划，Bayesian网，机器学习算法。
  - 结构化表示（Structured)：显示描述对象之间的关系。



    - 关系数据库，一阶逻辑，一阶概率模型，基于知识的学习，自然语言理解。

3.1 概述 -通过搜索进行问题求解

- **通过搜索进行问题求解：**
  - 当问题求解不能通过单个行动一步完成时，Agent 如何找到一组行动序列达到目标。
  - **搜索即是指从问题出发寻找解的过程。**
  - 本章讨论中基于**目标的Agent**的一种，称为**问题求解 Agent** (problem-solving Agent)
    - 问题求解 Agent 使用原子（atomic) 表示，世界的状态被视为一个整体，对问题求解算法而言没有可见的内部结构。
  - 使用更先进的要素化或结构化表示的基于目标的Agent，通常成为规划Agent。

3.1 概述 -通过搜索进行问题求解

- **通过搜索进行问题求解：**
  - 如何描述问题及其解。
  - 求解此类问题的通用的搜索算法：
    - 无信息搜索：除了问题定义本身没有任何其他信息；
    - 有信息搜索：利用给定的知识引导能够更有效地找到解。
  - 任务环境简化：
    - 限定问题的解是一组有固定顺序的行动；
    - 更一般的情况是 Agent 的行动决策将随着未来感知数据的改变而改变。

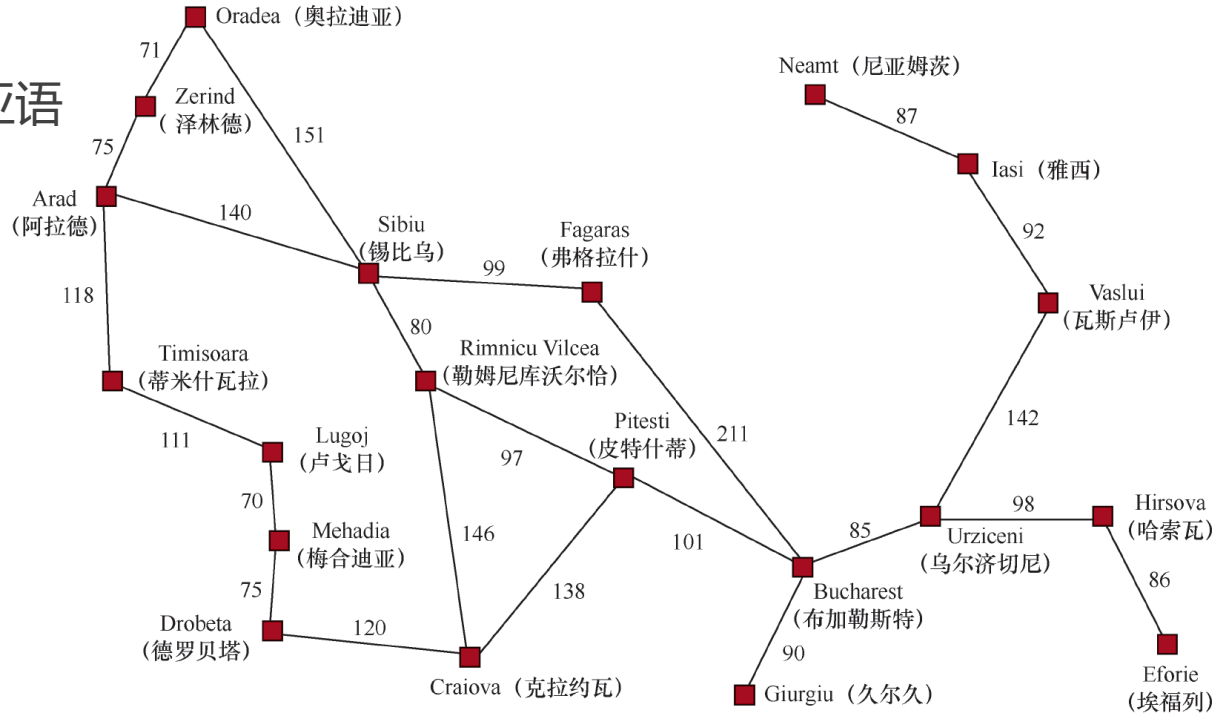
3.1 概述 -渐进复杂度和NP完全性

- 渐进复杂度：O(.)表示法：
  - 随着n趋近于无穷大时，O(n)算法要比O(n^2)好。
- NP完全性：
  - P：多项式问题，能够在O(n^k)时间内解决的问题；
  - NP：不确定多项式问题，指存在某个算法能够在多项式时间内猜测出一个该问题的解并验证这个猜测是否正确。
- 开放问题：NP是否等于P：
  - NP完全问题：NP类中最难的问题。已经证明：要么所有的NP完全问题都是P问题，要么一个也不是。

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.2 问题求解Agent

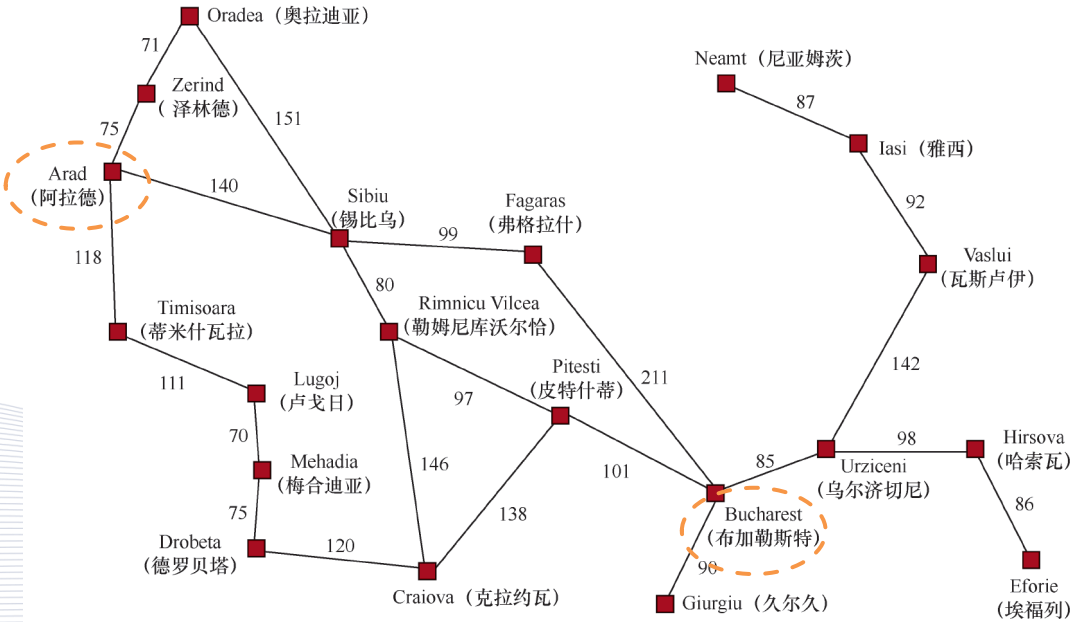
- 一个Agent去往罗马尼亚度假，“TA”或许有一些想法：
  - 想晒黑些
  - 想学习罗马尼亚语
  - 欣赏风景
  - 享受夜生活
  - 避免宿醉
  - 等等



罗马尼亚部分地区的简化道路图，道路距离单位为英里（1 英里 = 1.61 千米）

3.2 问题求解Agent –目标形式化

- 假设Agent要最大化其性能度量。Agent采纳一个目标，并试图去满足它，最大化性能度量的问题就可能会简化。
- 基于当前的情形和Agent的性能度量进行**目标形式化**，是问题求解的第一个步骤。
  - 例如目标是从Arad去往Bucharest。



罗马尼亚部分地区的简化道路图，道路距离单位为英里（1 英里 = 1.61 千米）

3.2 问题求解Agent –目标形式化

- 目标：世界的一个状态集合——目标被满足的那些状态的集合。
- Agent的任务：找出现在和未来的行动，以使它达到一个目标状态。
- 还需要什么？
  - 需要获得Agent能完成的行动种类和行动能带来的状态变化。
- **问题形式化**：给定目标下确定需要考虑哪些行动和状态的过程。

3.2 问题求解Agent –问题形式化

- **问题形式化**：Agent刻画实现目标所必需的状态和动作，进而得到这个世界中与实现目标相关的部分所构成的抽象模型。
  - 考虑从一个城市到其相邻城市的动作，这时，状态中只有“当前所在城市”会由于动作而改变。
- 两种情况：
  - 没有地图：不知道应该走哪条路，称为未知的环境。（第4章）
  - 有地图，并且地图上的每个点都可以向Agent提供信息，称为环境可观察。（本章主要讨论）
- Agent在面临多种未知值得选择时，可以首先检查那些最终导出已知价值的状态的未来行动，然后做出决策。

3.2 问题求解Agent –回顾任务环境分类

- 任务环境分类维度：
  - 完全可观察的与部分可观察的
  - 单Agent和多Agent
  - 确定的与随机的
  - 片段式的与延续式的
  - 静态的与动态的
  - 离散的与连续的
  - 已知的与未知的

3.2 问题求解Agent –搜索

- 假设：
  - 环境可观察的：Agent总是知道当前状态
  - 环境是离散的：在任一给定状态，可以选择的行动是有限的
  - 环境是已知的：Agent知道每个行动达到哪个状态
  - 环境是确定的：每个行动的结果只有一个
- 这些假设下，问题的解是一个行动的固定序列。
- 寻找这样的行动序列的过程被称为**搜索**。

3.2 问题求解Agent –执行

- 搜索算法的输入是问题，输出是问题的解，以行动序列的形式返回问题的解。
- 解一旦找到，它所建议的行动将会付诸实施。这被称为执行阶段。那么，我们就完成了对Agent 的简单设计，即“形式化、搜索、**执行**。

```
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

  state ← UPDATE-STATE(state, percept)
  if seq is empty then do
      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
```

- **4个阶段：**
  - 目标形式化
  - 问题形式化
  - 搜索
  - 执行

3.2.1 良定义的问题及解

- **问题的形式化定义：**
  - 一个问题可以用5个组成部分形式化地描述：
    - Agent的初始状态；
    - 描述Agent的可能行动。给定一个特殊状态s，ACTIONS(s)返回在状态s下可以执行的行动集合；
    - 对每个行动的描述；正式的名称是转移模型，用函数RESULT(s, a)描述：在状态s下执行行动a后达到的状态；
    - 目标测试，确定给定的状态是不是目标状态；
    - 路径耗散函数：为每条路径赋一个耗散值，即边加权。问题求解Agent选择能反映它自己的性能度量的耗散函数。

3.2.1 良定义的问题及解

- 由上述元素即可定义一个问题，通常把它们组织在一起成为一个数据结构，并以此作为问题求解算法的输入。
- 问题的解就是从初始状态到目标状态一组行动序列。
- 解的质量由路径耗散函数度量，所有解里路径耗散值最小的解即为**最优解**。

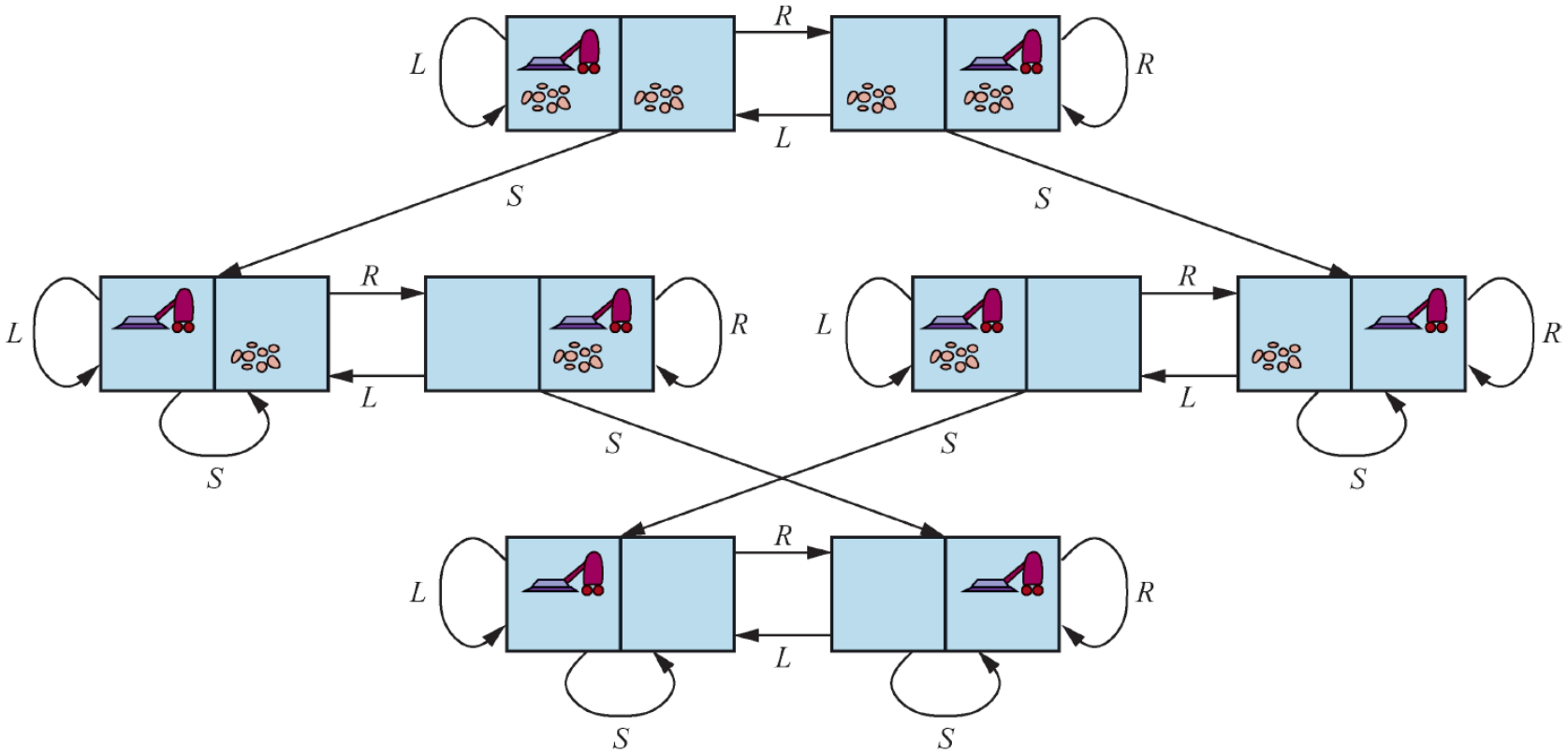
3.2.2 问题的形式化

- **一个良好的问题形式化应该具有适度的细节层次。**
  - 抽象：在表示中去除细节的过程。
  - 状态描述要抽象，行动也需要抽象。
  - 选择一个好的抽象至关重要，在保持有效抽象的前提下，去除尽可能多的细节和确保抽象的行动容易完成。否则，智能Agent将被现实世界完全淹没。
    - 不合适的抽象：如果是“右脚向前移动1 厘米”或“方向盘向左转动1 度”的抽象层次，Agent可能永远都找不到走出停车场的路，更不用说去Bucharest（目标地） 了。

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.3 问题实例 –真空吸尘器世界

- 真空吸尘器世界：



两个单元格的真空吸尘器世界的状态空间图。共有8个状态，每个状态有3种动作：L = Left（向左）、R = Right（向右）、S = Suck（吸尘）。

3.3 问题实例 –真空吸尘器世界

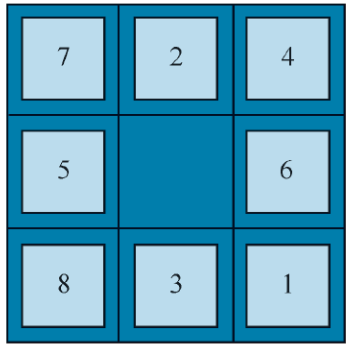
- **真空吸尘器世界：**
  - **状态**：状态：状态由Agent位置和灰尘位置确定。Agent的位置有两个，每个位置都可能有灰尘。因此，可能的世界状态有$2*2^2=8$。对于具有n个位置的大型环境而言，状态数为$n*2^n$。
  - **初始状态**：任何状态都可能被设计成初始状态。
    - 8种可能的世界状态：

3.3 问题实例 –真空吸尘器世界

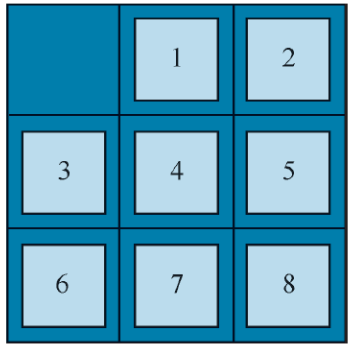
- **真空吸尘器世界：**
  - **行动**：可执行的动作只有3个：Left，Right，Suck。大型的任务环境中还可能包括Up和Down。
  - **转移模型**：行动会产生它们所期待的后果，除了在最左边位置不能Left再向左移动，在最右边位置不能再Right向右移动，在干净的位置进行Suck也没有效果。
  - **目标测试**：检测所有位置是否干净。
  - **路径消耗**：每一步耗散值为1，因此整个解路径的耗散值是路径中的步数。

3.3 问题实例 –八数码问题

- **八数码问题：**



开始状态

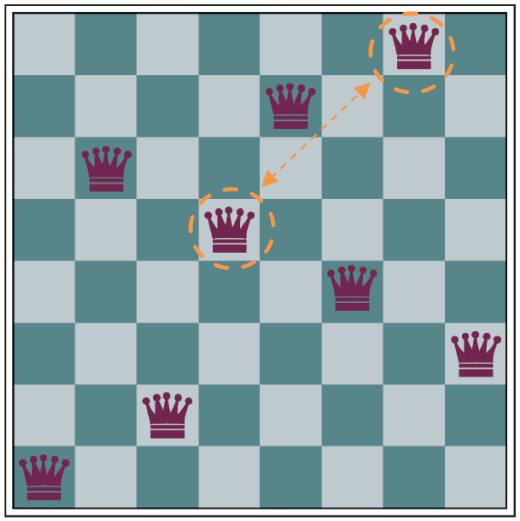


目标状态

- **状态**：状态描述指明8个棋子以及空格在棋盘9个方格上的分布。
- **初始状态**：任何状态都可能是初始状态。注意要到达任何一个给定的目标，可能的初始状态中恰好只有一半可以作为开始。
- **后继函数**：用来产生通过四个行动(把空位向Left、Right、Up或Down移动)能够达到的合法状态。
- **目标测试**：用来检测状态是否能目标布局。
- **路径耗散**：每一步耗散值为1，因此整个路径的耗散值是路径中的步数。

3.3 问题实例 –八皇后问题

- 八皇后问题：



- 8 皇后问题：在棋盘上放置8个皇后，使得它们不能互相攻击。（皇后会攻击同一行、同一列或对角线上的任何棋子）
- 左图状态非常接近于一个解，除了第4列和第7列的两个皇后会沿对角线互相攻击。

- **状态**：棋盘上0到8个皇后的任一摆放都是一个状态。
- **初始状态**：棋盘上没有皇后。
- **行动**：在任一空格增加摆放1个皇后。
- **转移模型**：返回增加了皇后的棋盘。
- **目标测试**： 8个皇后都在棋盘上，并且无法互相攻击。

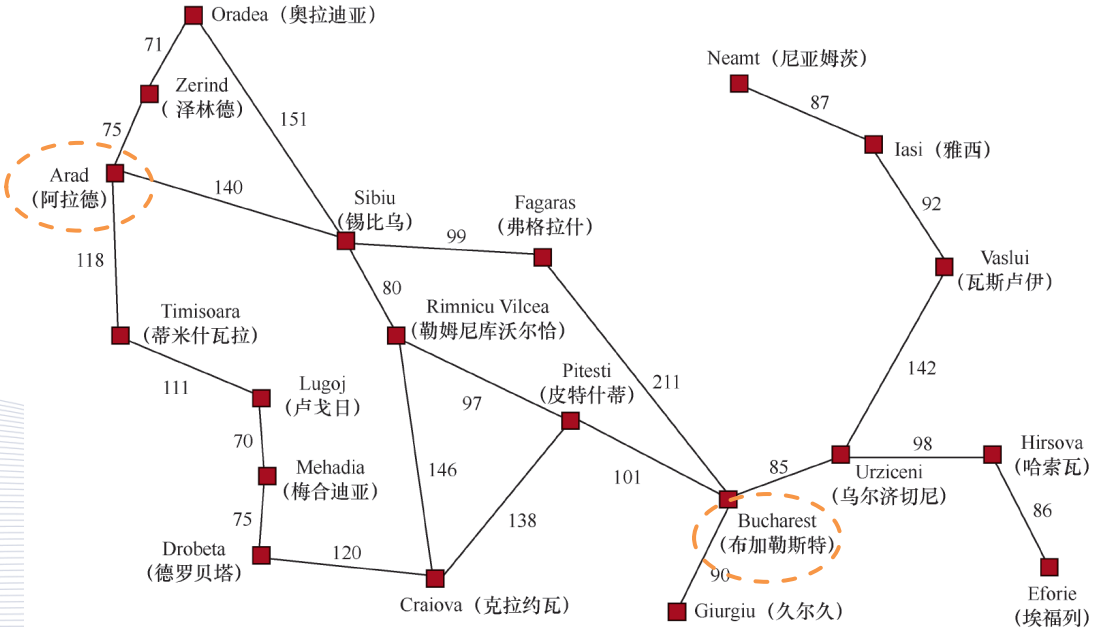
3.3 问题实例 –现实世界问题

- **现实世界问题：**
  - 寻径问题
  - 旅行问题
  - 旅行商问题
  - VLSI布线问题
  - 机器人导航问题
  - 自动装配序列问题

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.4 通过搜索求解 –Tree Search

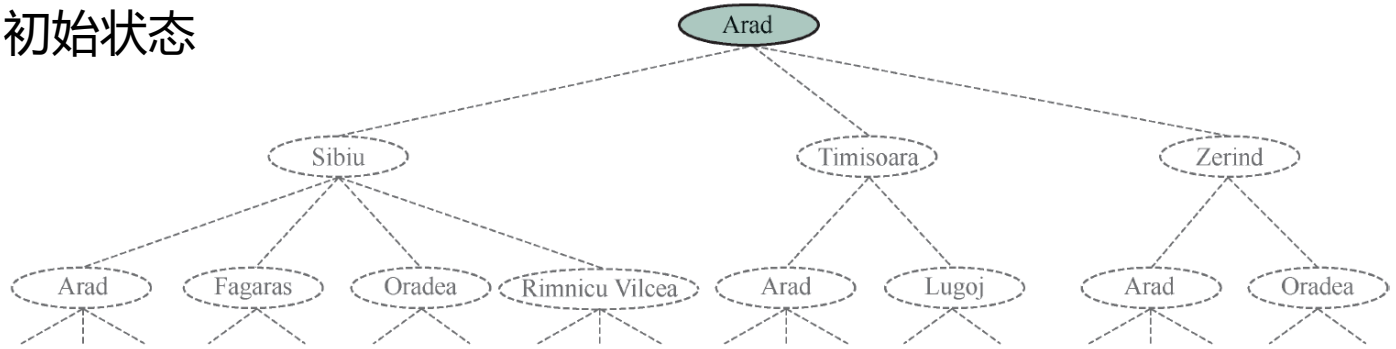
- 一个解是一个行动序列，搜索算法的工作就是考虑各种可能的行动序列：
  - 从搜索树的根节点的初始状态出发。
  - 连线表示行动。
  - 结点对应问题的状态空间中的状态。



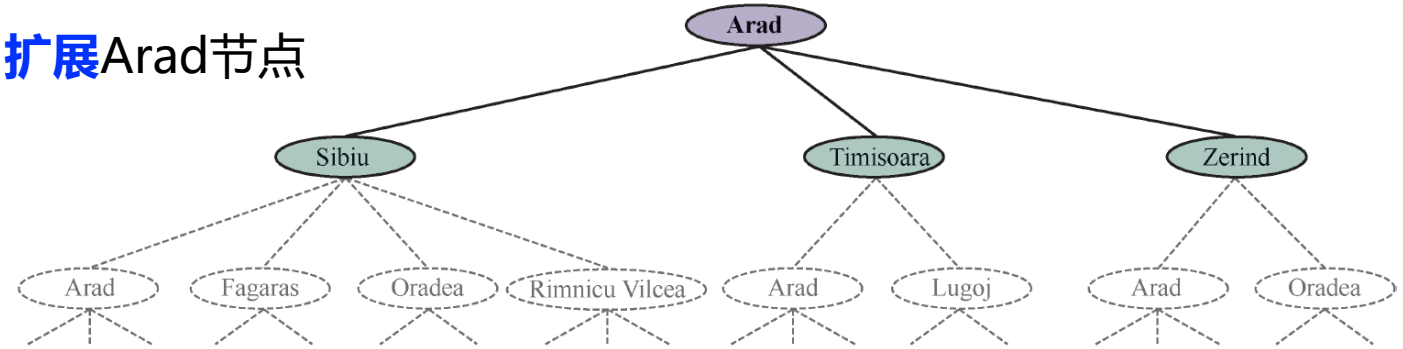
罗马尼亚部分地区的简化道路图，道路距离单位为英里（1 英里 = 1.61 千米）

3.4 通过搜索求解 –Tree Search

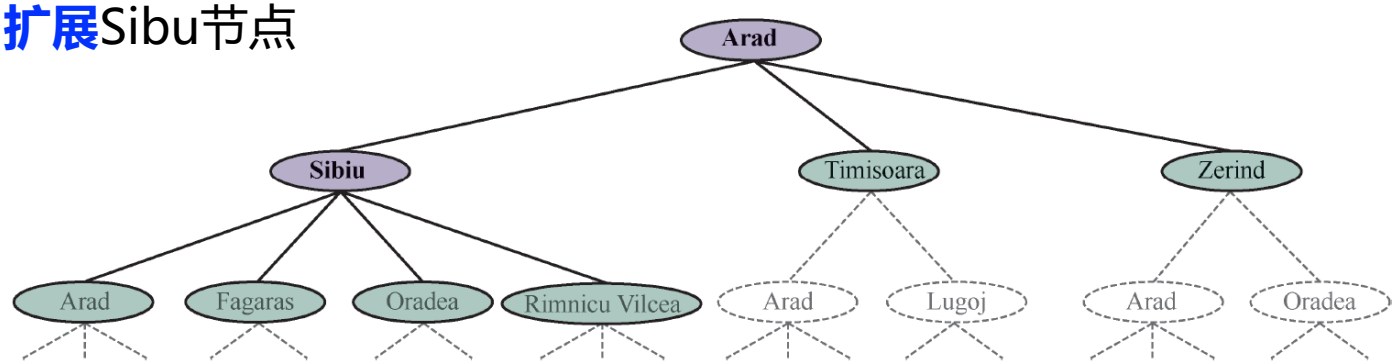
初始状态



扩展Arad节点

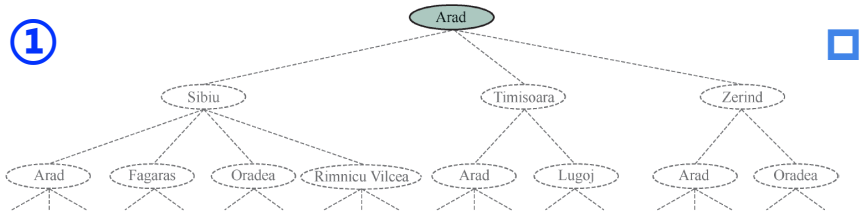


扩展Sibu节点

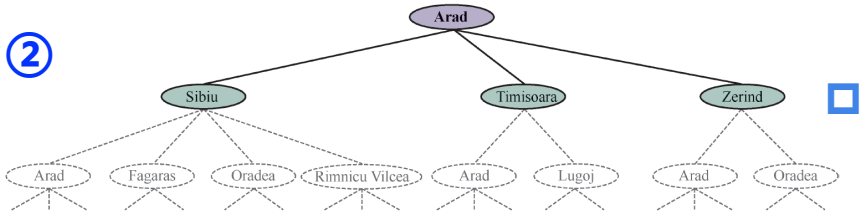


3.4 通过搜索求解 –Tree Search

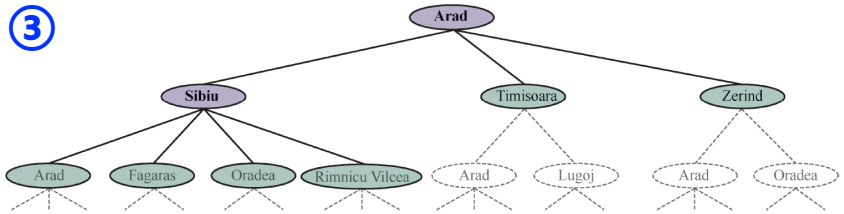
①



②



③

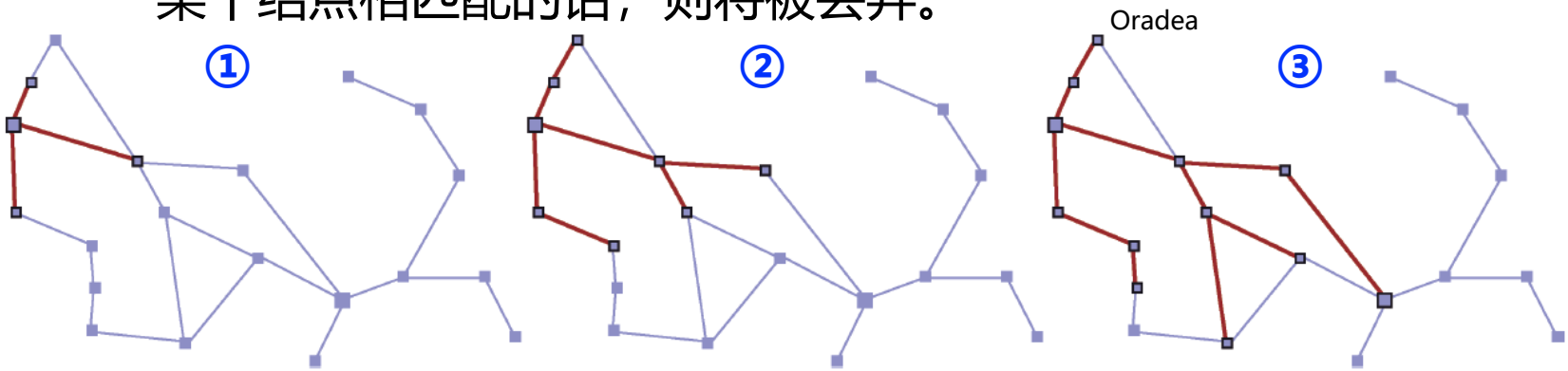


- 搜索树中的每个**节点**（node）对应于状态空间中的一个状态，每个边对应于动作。树的根对应于问题的初始状态。
- **扩展**（expand）节点：考虑该状态的可用动作，查看这些动作指向何处，并为每个结果状态**生成**（generating）一个新节点，称为**子节点**（child node）或**后继节点**（successor node）。

- ①→②中每个子节点的**父节点**（parent node）都是Arad。
- ②→③中假定先扩展Sibiu，得到了6 个未被扩展的节点（以绿色节点显示）。我们称之为搜索树的**边界**（frontier）。任何已经生成过节点的状态都称为**已达**（reached）状态（无论该节点是否被扩展）

3.4 通过搜索求解 –Graph Search

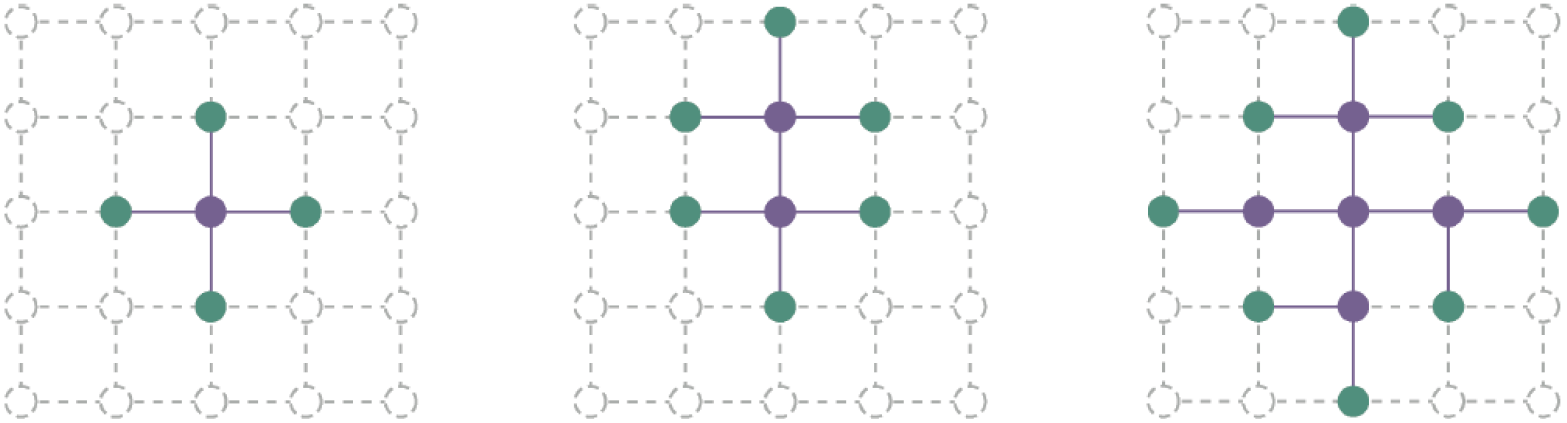
- **Graph-Search:**
  - 给Tree-search算法增加一个参数——探索集（closed 表）
  - 记录每个已扩展过的结点。新生成的结点若与已经生成的某个结点相匹配的话，则将被丢弃。



  - 在每一阶段，我们扩展边界上的每个节点，使用所有不指向已达状态的可用动作延伸每条路径。
  - 注意，在第③中，最高位置的城市（Oradea）有两个后继城市，这两个城市都已经有其他路径到达，所以没有路径可以从 Oradea 延伸。

3.4 通过搜索求解 –Graph Search

- **Graph-Search:**
  - 边界**分离**（separate）了状态空间图的两个区域，即内部区域（其中每个状态都已被扩展）和外部区域（尚未到达的状态）。



以矩形网格问题为例说明图搜索的分离性质。边界（绿色）分离了内部（淡紫色）和外部（虚线）。边界是已达但尚未扩展的节点（及相应的状态）的集合；内部是已被扩展的节点（及相应的状态）的集合；外部是尚未到达的状态的集合。在（a）中，只有根节点被扩展。在（b）中，上面的边界节点被扩展。在（c）中，按顺时针顺序扩展根节点的其他后继节点

3.4 通过搜索求解

```
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
```

- 一般的树搜索和图搜索算法的非形式化描述。

```
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
```

- 用粗斜体标出的图搜索算法的部分内容是指需要额外处理重复状态。

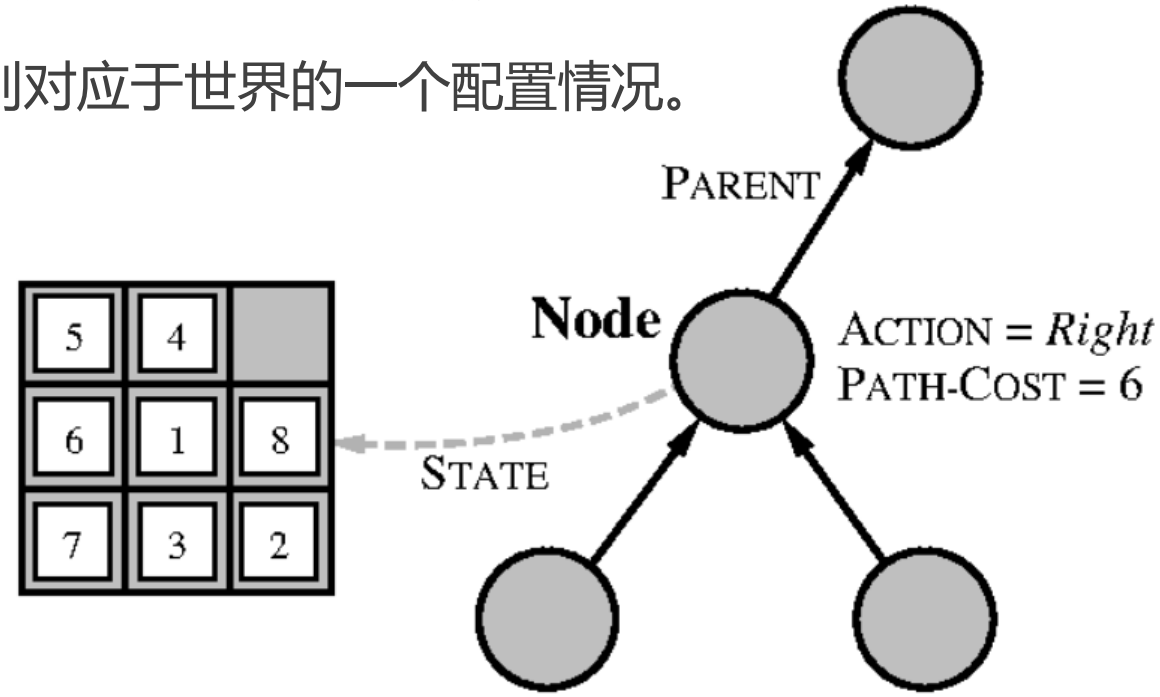
3.4.1 搜索算法基础

- **记录搜索树的数据结构：**
  - *n.STATE*：对应状态空间中的状态（也写作*node. State*，下同）
  - *n.PARENT*：搜索树中产生该结点的结点（即父结点）
  - *n.ACTION*：父结点生成该结点时所采取的行动
  - *n.PATH-COST*：代价，一般用g(n)表示，指从初始状态到达该结点的路径消耗

```
function CHILD-NODE(problem, parent, action) returns a node
    return a node with
        STATE = problem.RESULT( parent.STATE, action),
        PARENT = parent, ACTION = action,
        PATH-COST=parent. PATH-COST+problem.STEP-COST(parent.STATE, action)
```

3.4.1 搜索算法基础

- **节点：**
  - 结点是用来表示搜索树的数据结构。
  - 状态则对应于世界的一个配置情况。



结点是数据结构，由搜索树构造。每个结点都有一个父结点、一个状态和其他域。箭头由子结点指向父结点。

3.4.1 搜索算法基础

- **队列：**
  - 我们需要一个数据结构来存储边界。一个恰当的选择是某种队列（queue），因为边界上的操作有以下几个：
    - IS-EMPTY(*frontier*)：返回 true 当且仅当边界中没有节点。
    - POP(*frontier*)：返回边界中的第一个节点并将它从边界中删除。
    - TOP(*frontier*)：返回（但不删除）边界中的第一个节点。
    - ADD(*node*,*frontier*)：将节点插入队列中的适当位置。
  - **三种常见的队列形式：**
    - 先进先出（FIFO）
    - 后进先出（LIFO），又称之为栈
    - 优先级队列：队列中的元素具有根据函数计算出的优先级，总是具有最高优先级的队列出队

3.4.2 问题求解算法的性能

- **评价一个算法的性能考虑四个方面：**
  - 完备性：当问题有解时，这个算法是否能保证找到解？
  - 最优性：搜索策略是否能找到最优解？
  - 时间复杂度：找到解需要花费多长时间？
  - 空间复杂度：在执行搜索的过程中需要多少内存？

- d，最优解的深度（depth）或动作数；
- m，任意路径的最大动作数；
- b，需要考虑的节点的分支因子（branching factor）或后继节点数。

时间常常由搜索过程中产生的结点数目来度量

空间则由内存中储存的最多结点数来度量

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.5 无信息搜索策略

- **无信息搜索（盲目搜索）：**
  - 除了问题定义中提供的状态信息外没有任何附加信息。
- **有信息搜索（启发式搜索）：**
  - 知道一个非目标状态是否比其他状态“更有希望”接近目标的策略。

3.5.1 宽度优先搜索

- **宽度优先搜索（BFS）：**
  - 先扩展根结点，接着扩展根结点的所有后继，然后再扩展它们的后继，依此类推：

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  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 do
        if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
        frontier ← INSERT(child, frontier)
```

Figure 3.11 Breadth-first search on a graph.

3.5.1 宽度优先搜索

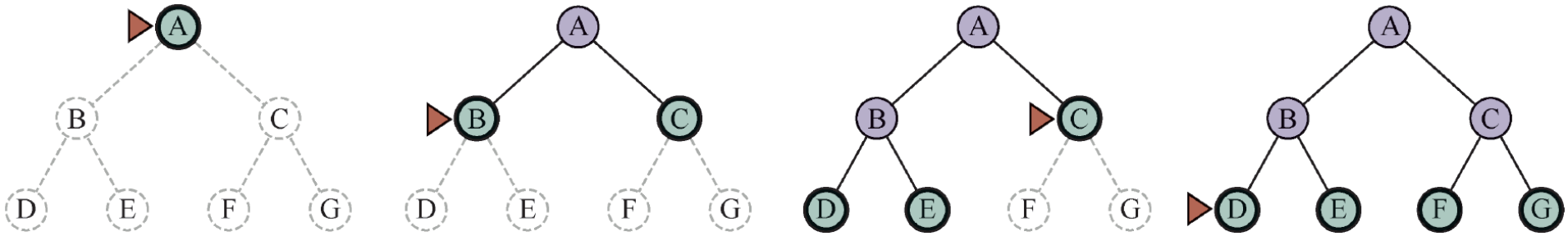


图 3-8　简单二叉树上的广度优先搜索。每个阶段接下来要扩展的节点用三角形标记表示

- **优点：**
  - 完备的
  - 目标结点一经生成，一定是最浅的目标结点。最浅的目标结点未必是最优的。
  - 如果路径代价是基于结点深度的非递减函数，则宽度优先搜索是最优的
- **缺点：**
  - 时间复杂度和空间复杂度不友好
  - 深度d，宽度b，则时间复杂度$O(b^d)$
  - 空间复杂度： $O(b^d)$

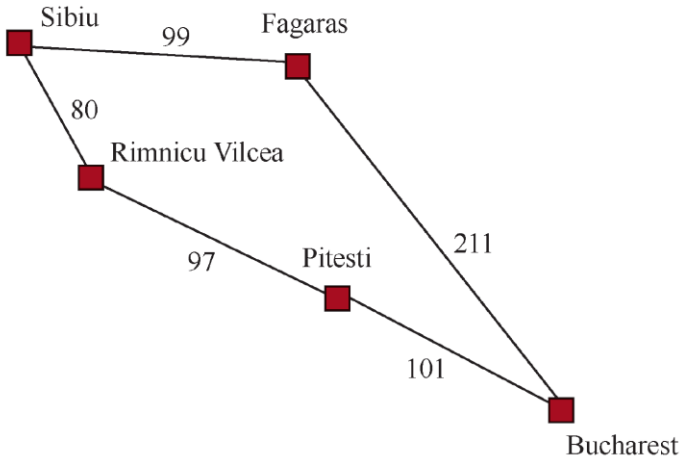
3.5.2 一致代价搜索

- **一致代价搜索（Dijkstra算法）：**
  - 当每一步的行动代价都相等时宽度优先搜索是最优的。
  - 一致代价搜索：对任何单步代价函数都是最优的算法。扩展的是路径消耗g(n)最小的结点n。

```
function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 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
```

3.5.2 一致代价搜索

- **除了按路径代价对队列进行排序外，一致代价和宽度优先算法有两个显著不同：**
  - 目标检测应用于结点被选择扩展时，而不是在结点生成的时候进行。
  - 如果边缘中的结点有更好的路径达到该结点那么会引入一个测试。



- ① Sibiu->Rimnicu Vilcea: 80
- ② Sibiu->Fagaras: 99
- ③ Sibiu->Rimnicu Vilcea->Pitesti: 80+97=177
- ④ Sibiu->Fagaras->**Bucharest**: 99 + 211 = 310

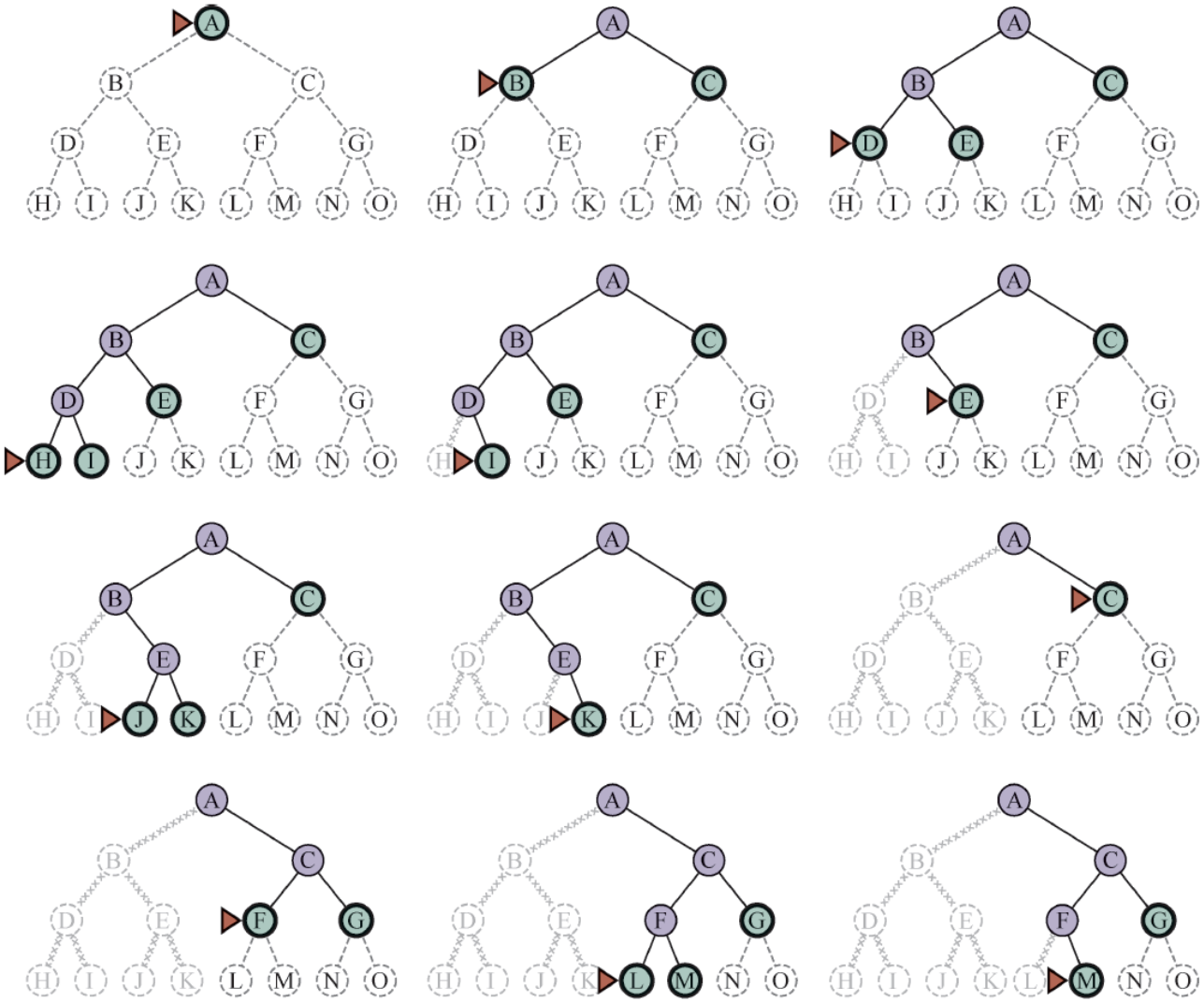
（目标结点已经生成，但仍要继续）

- ⑤ Sibiu->Rimnicu Vilcea->Pitesti->**Bucharest**: 80+97+101=278 **（这条路才是最近的）**

3.5.3 深度优先搜索

- **深度优先搜索（DFS）：**
  - 总是扩展搜索树的当前边缘结点集中最深的结点。
  - 深度优先：LIFO（后进先出）队列。
  - 宽度优先：FIFO（先进先出）队列。
  - 当所有单步耗散都相等时，宽度优先搜索在找到解时终止，一致代价搜索会检查目标深度的所有结点看谁的代价最小，在深度d无意义地做了更多的工作。

3.5.3 深度优先搜索



- 二叉树的DFS，从起始A到目标M，共12步(从左到右，从上到下)。
- 边界节点为绿色，用三角形表示下一步要扩展的节点。已扩展的节点为淡紫色，潜在的未来节点用模糊的虚线表示。边界中没有后继的已扩展节点(用非常模糊的线表示)可以丢弃。

3.5.3 深度优先搜索

- 时间复杂度：
  - $O(b^m)$，m指任一结点的最大深度；
  - 和宽度优先搜索相比没有优势。
- 空间复杂度：
  - O(bm)；
  - **空间复杂度低**。对图搜索而言，深度优先搜索只需要存储一条从根结点到叶结点的路径，以及该路径上每个结点的所有未被扩展的兄弟结点即可。
- 回溯搜索：深度优先搜索的一个变种：
  - 空间复杂度更低，O(m)。

3.5.4 深度受限搜索

- **深度受限搜索（Depth-limited Search）：**
  - 深度为l的结点被当作没有后继对待;
  - 解决了无穷路径的问题;
  - 如果$l<d$，最浅的目标结点的深度超过了深度限制，该搜索方法是不完备的;
  - 如果$l>d$，时间复杂度是$O(b^l)$，空间复杂度是$O(bl)$;
  - 深度优先搜索可以看作是特殊的深度受限搜索，其深度$l=\infty$.

d：目标结点所在的最浅的深度（从根结点到目标状态的步数）

3.5.4 深度受限搜索

- 深度受限树搜索的递归实现:

```
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)
  else if limit = 0 then return cutoff
  else
      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 else return failure
```

3.5.5 迭代加深的深度优先搜索

- 不断增大深度限制——首先为0，接着为1，然后为2，以此类推——直到找到目标；

```
function ITERATIVE-DEEPENING-SEARCH(problem) returns 一个解节点或failure
  for depth = 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result

function DEPTH-LIMITED-SEARCH(problem, ℓ) returns一个解节点或failure或cutoff
  frontier ← 一个LIFO队列（栈），其中一个元素为NODE(problem.INITIAL)
  result ← failure
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    if DEPTH(node) > ℓ then
      result ← cutoff
    else if not IS-CYCLE(node) do
      for each child in EXPAND(problem, node) do
        将child添加到frontier
  return result
```

3.5.5 迭代加深的深度优先搜索

- 不断增大深度限制——首先为0，接着为1，然后为2，以此类推——直到找到目标；
- 结合了深度优先搜索和宽度优先搜索的优点（时间换空间）；
- 空间复杂度：$O(bd)$；
- 时间复杂度：$O(b^d)$；
  - $(d)b^1+(d-1)b^2+(d-2)b^3+\ldots+b^d$
- 一般来讲，当搜索空间较大并且不知道解所在深度时，迭代加深的深度优先搜索是**首选**的无信息搜索方法。

3.5.6 双向搜索

- **双向搜索的思想是同时运行两个搜索：**
  - 一个从初始状态向前搜索；
  - 另一个从目标状态向后搜索；
  - 希望它们在中间某点相遇，此时搜索终止；
  - $b^{d/2}+b^{d/2} << b^{d}$ ；
  - 未必是最优解；
  - 空间和时间算法复杂度都为$O(b^{d/2})$ 。
- **从目标开始的向后搜索的“目标”是指什么?**
  - 八数码问题，罗马尼亚问题，真空吸尘器问题
  - 八皇后问题（目标状态是一种抽象描述，很难应用双向搜索）

3.5.7 无信息搜索策略对比

指标	广度优先	一致代价	深度优先	深度受限	迭代加深	双向（如适用）
完备性	是[1]	是[1,2]	否	否	是[1]	是[1,4]
代价最优	是[3]	是	否	否	是[3]	是[3,4]
时间复杂性	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
空间复杂性	$O(b^d)$	$O(b^{1+\lfloor C^*/\varepsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

注：[1] 如果 b 是有限的且状态空间要么有解要么有限，则算法是完备的。[2] 如果所有动作的代价都$\geqslant \varepsilon > 0$，算法是完备的。[3] 如果动作代价都相同，算法是代价最优的。[4] 如果两个方向均使用广度优先搜索或一致代价搜索

图 3-15　搜索算法比较。b 是分支因子；m 是搜索树的最大深度；d 是最浅层解的深度，当不存在解时为 m；ℓ 是深度界限

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.6 有信息（启发式）的搜索策略

- **有信息搜索策略：**
  - 使用问题本身的定义之外的特定知识；
  - 比无信息的搜索策略更有效地进行问题求解。
- **最佳优先搜索：**
  - 是一般树搜索和图搜索的一个实例，结点是基于评价函数f(n)被选择扩展的；
  - 评价函数是代价估计，评估值最低的结点被选择首先进行扩展；
  - 与一致代价搜索类似，最佳优先是f而不是g对优先级队列排队。

3.6 有信息（启发式）的搜索策略

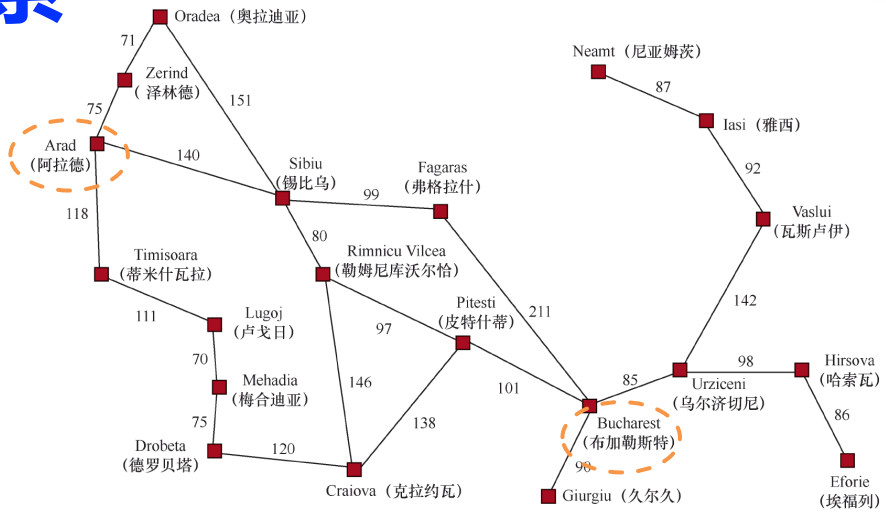
- **对f的选择决定了搜索策略，大多数最佳优先搜索算法的f由启发函数h构成：**

$h(n)$ = 从节点 n 的状态到目标状态的最小代价路径的代价估计值

  - 贪婪最佳优先搜索；
  - A*搜索；
  - 存储受限的启发式搜索。

3.6.1 贪婪最佳优先搜索

- **贪婪最佳优先搜索：**
  - 试图扩展离目标最近结点，这样可能可以很快找到解。



罗马尼亚部分地区的简化道路图，道路距离单位为英里（1 英里 = 1.61 千米）

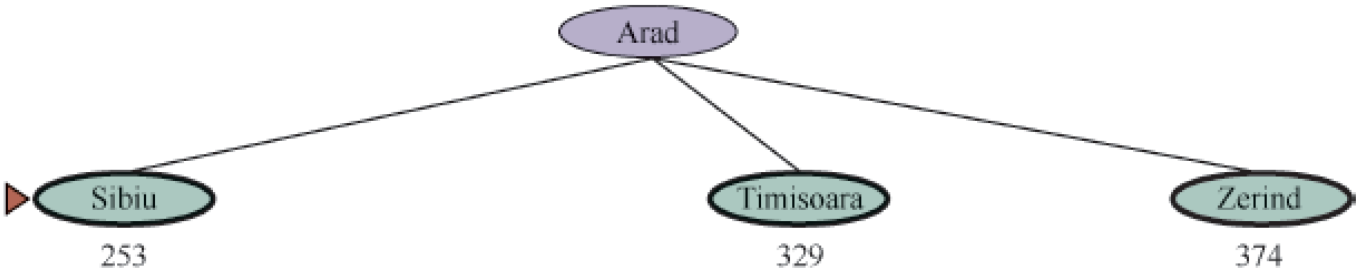
Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

h_{SLD}（到 Bucharest 的直线距离）的值

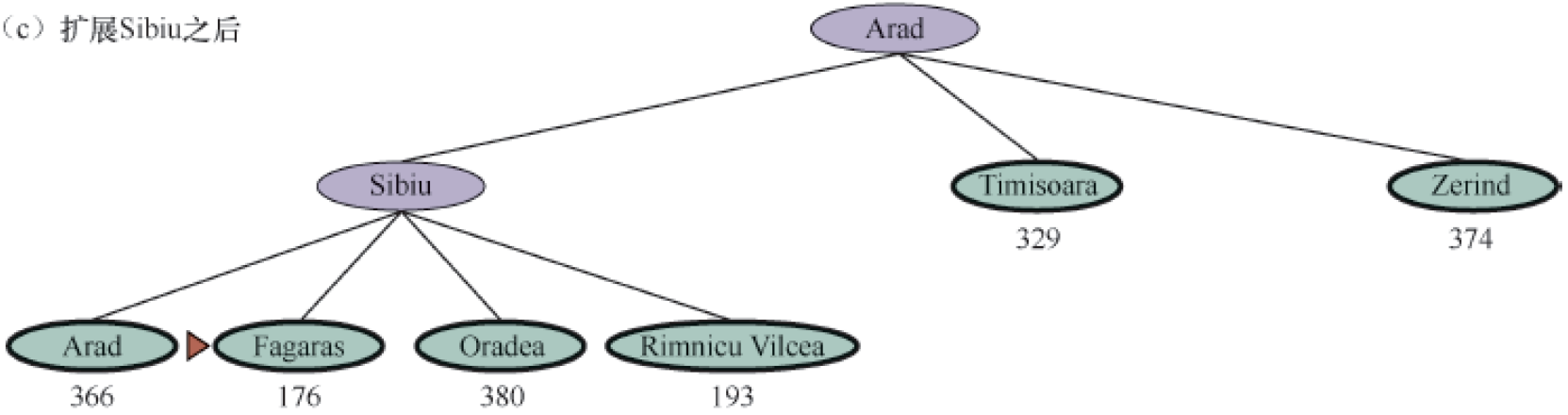
3.6.1 贪婪最佳优先搜索

（a）初始状态

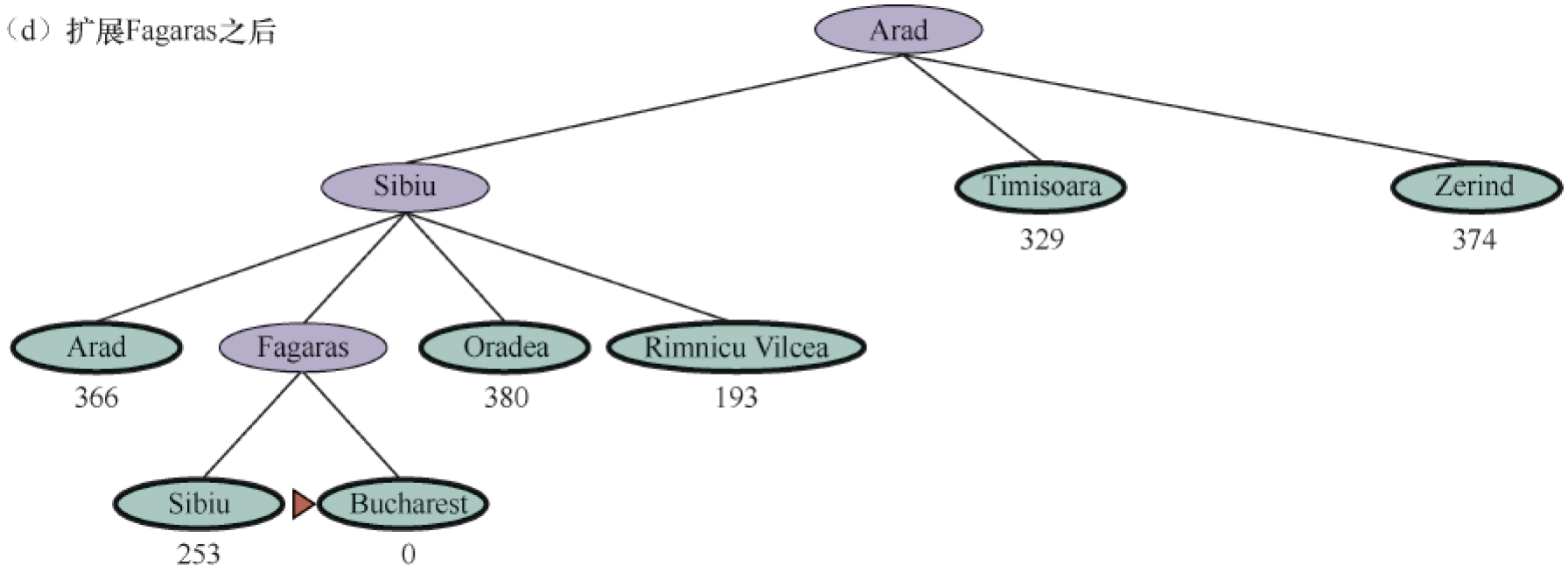
（b）扩展Arad之后



（c）扩展Sibiu之后

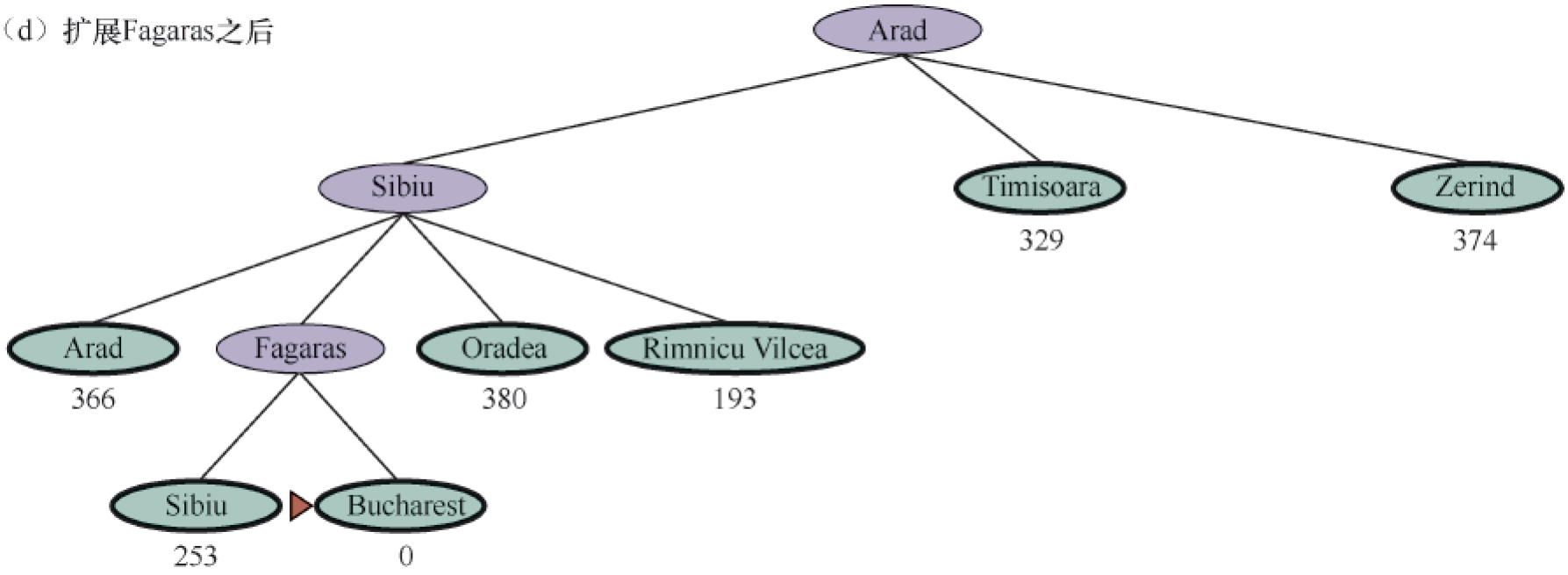


3.6.1 贪婪最佳优先搜索



- 找到的解不一定是代价最优的：
  - 经由Sibiu和Fagaras到达Bucharest路径比经过Rimnicu Vilcea和Pitesti的路径长32英里

3.6.1 贪婪最佳优先搜索



- 在有限状态空间中是完备的，在无限状态空间中是不完备的
- 最坏情况下：算法的时间复杂度和空间复杂度都是O(|V|)
- 复杂度降低的幅度取决于特定的问题和启发式函数的质量。对于一个好的启发式函数，时间复杂度和空间复杂度有可能降到O(bm)

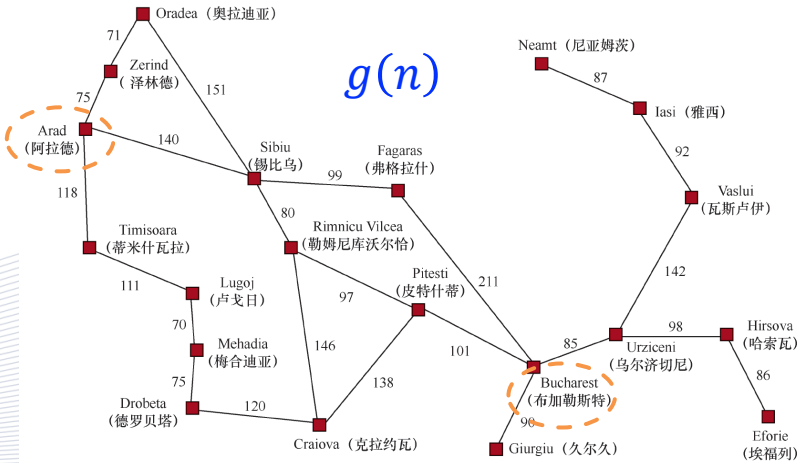
3.6.2 A*搜索

- **A*搜索（A-star search，A星搜索）：**
  - 是一种最佳优先搜索(可使用反证法证明)，评价函数为:

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

  - 其中$g(n)$是从初始状态到节点n 的路径代价， $h(n)$是从节点n 到目标状态的最短路径的代价***估计值***，因此我们有:

$f(n) =$ 经过n 到一个目标状态的最优路径的代价估计值



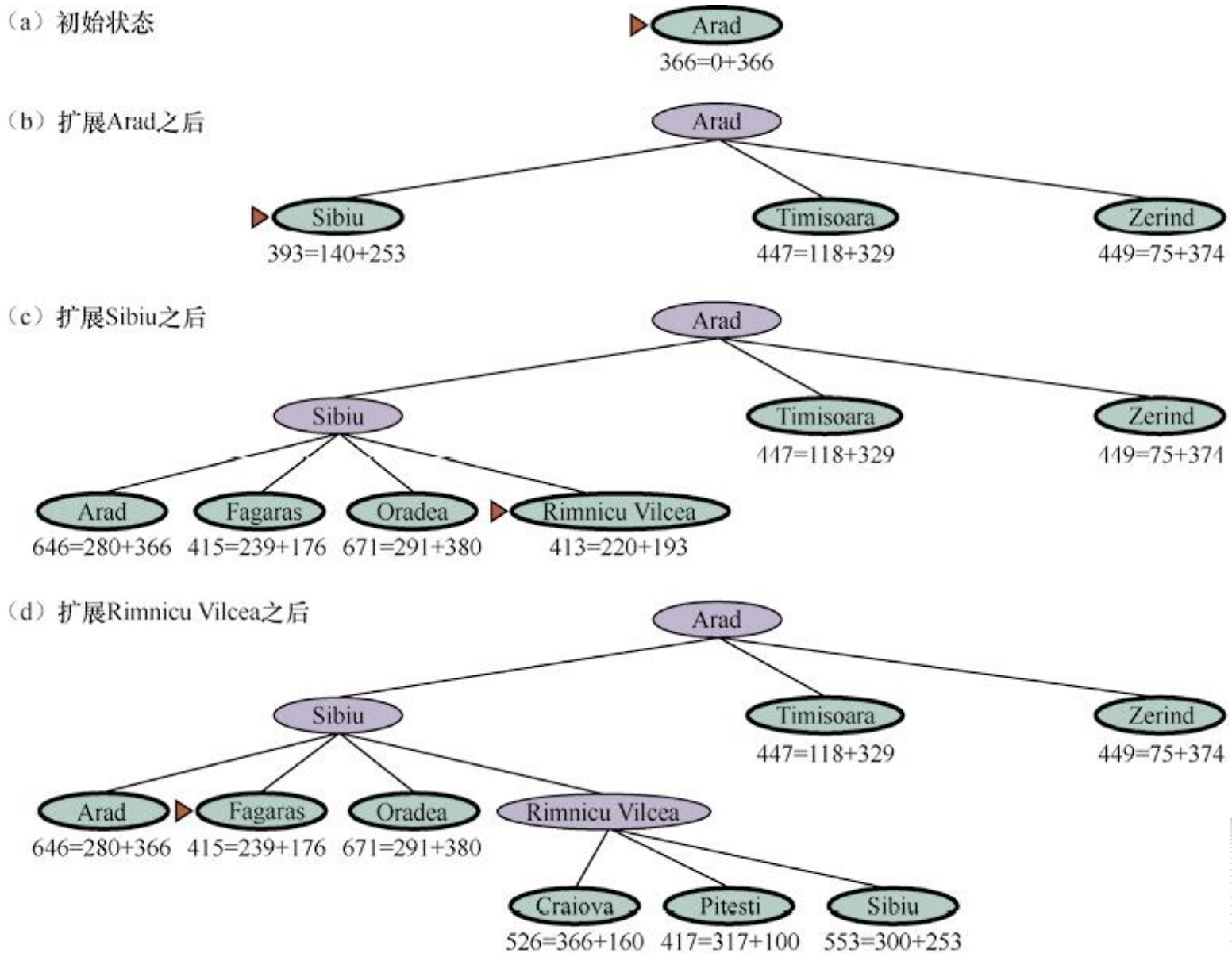
罗马尼亚部分地区的简化道路图，道路距离单位为英里（1 英里 = 1.61 千米）

$h(n)$

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

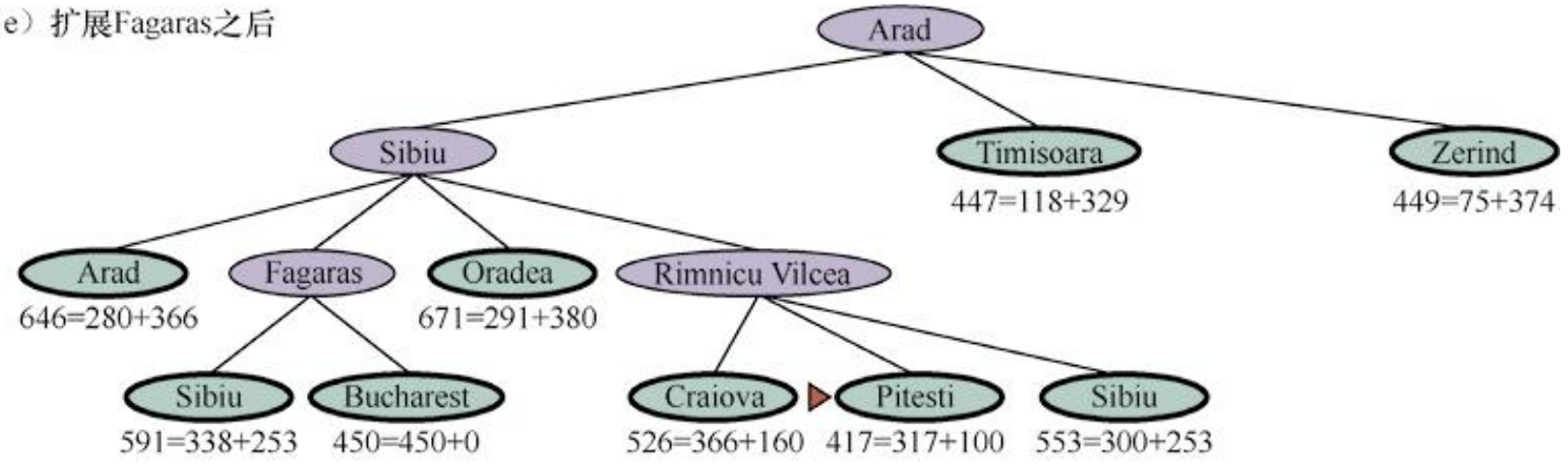
h_{SLD}（到 Bucharest 的直线距离）的值

3.6.2 A*搜索

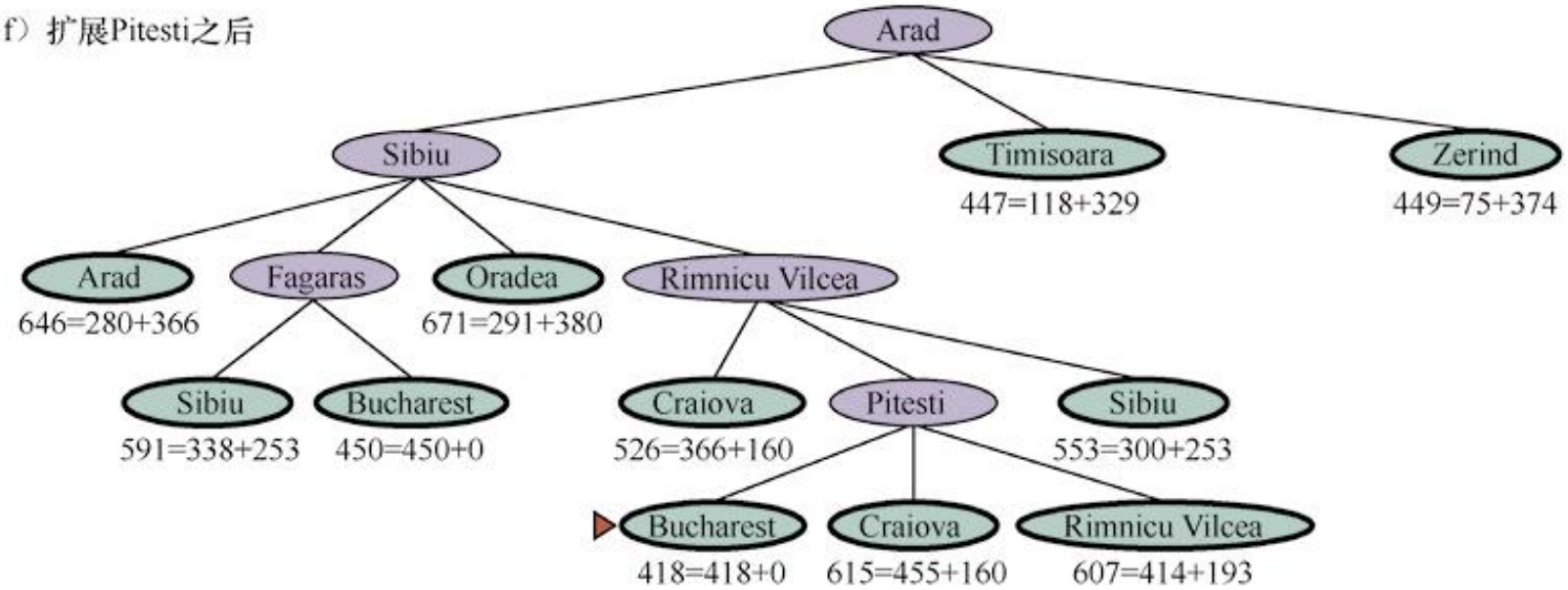

（a）初始状态
Arad
366=0+366
（b）扩展Arad之后
Arad
Sibiu
393=140+253
Timisoara
447=118+329
Zerind
449=75+374
（c）扩展Sibiu之后
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
413=220+193
（d）扩展Rimnicu Vilcea之后
Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
415=239+176
Oradea
671=291+380
Rimnicu Vilcea
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

3.6.2 A*搜索

（e）扩展Fagaras之后



（f）扩展Pitesti之后



3.6.2 A*搜索

- **A*搜素是完备的**
- **保证最优性的条件：可采纳性和一致性。**
- **可采纳启发式（可容许性）：**
  - 从不会过高估计达到目标的代价； $h(n) \leqslant h^*(n)$
  - 直线距离是可采纳的启发式，两点之间直线最短，所以用直线距离肯定不会高估。
- **一致性（单调性）：**
  - 如果对于每一个结点n和通过任一行动a生成的n的每个后继结点n'，从结点n达到目标的估计代价不大于从n到n'的单步代价与从n'到达目标的估计代价之和：

$$h(n) \leq c(n, a, n') + h(n')$$

3.6.2 A*搜索

- **一致的启发式函数都是可容许的，反之不成立。**
- **证明：**
  - 假设最佳路径上有k个结点，考虑k=1的情况： n' 就是目标结点，则有

$$h(n) \le c(n, a, n')$$

  - 更一般的情况，假设n'在最优路径上，离目标结点还有k步，并且$h(n')$ 是可容许的，则有

$$h(n) \le c(n, a, n') + h(n') \le c(n, a, n') + h^*(n') = h^*(n)$$

3.6.2 A*搜索 –最优性证明

- **A*算法的最优性，**
  - 如果h(n)是可采纳的，那么A*的树搜索版本是最优的
  - 如果h(n)是一致的，那么图搜索的A*算法是最优的
- **证明：**
  - ① 如果h(n)是一致的，那么沿着任何路径的f(n)值都是非递减的。
  - ② 如A*选择扩展结点n时，就已经找到到达结点n的最优路径。

3.6.2 A*搜索 –最优性证明

- ① 如果h(n)是一致的，那么沿着任何路径的f(n)值都是非递减的。

  - 假设n'是结点n的后继，那么对于行动a，有$g(n') = g(n) + c(n, a, n')$，可得:

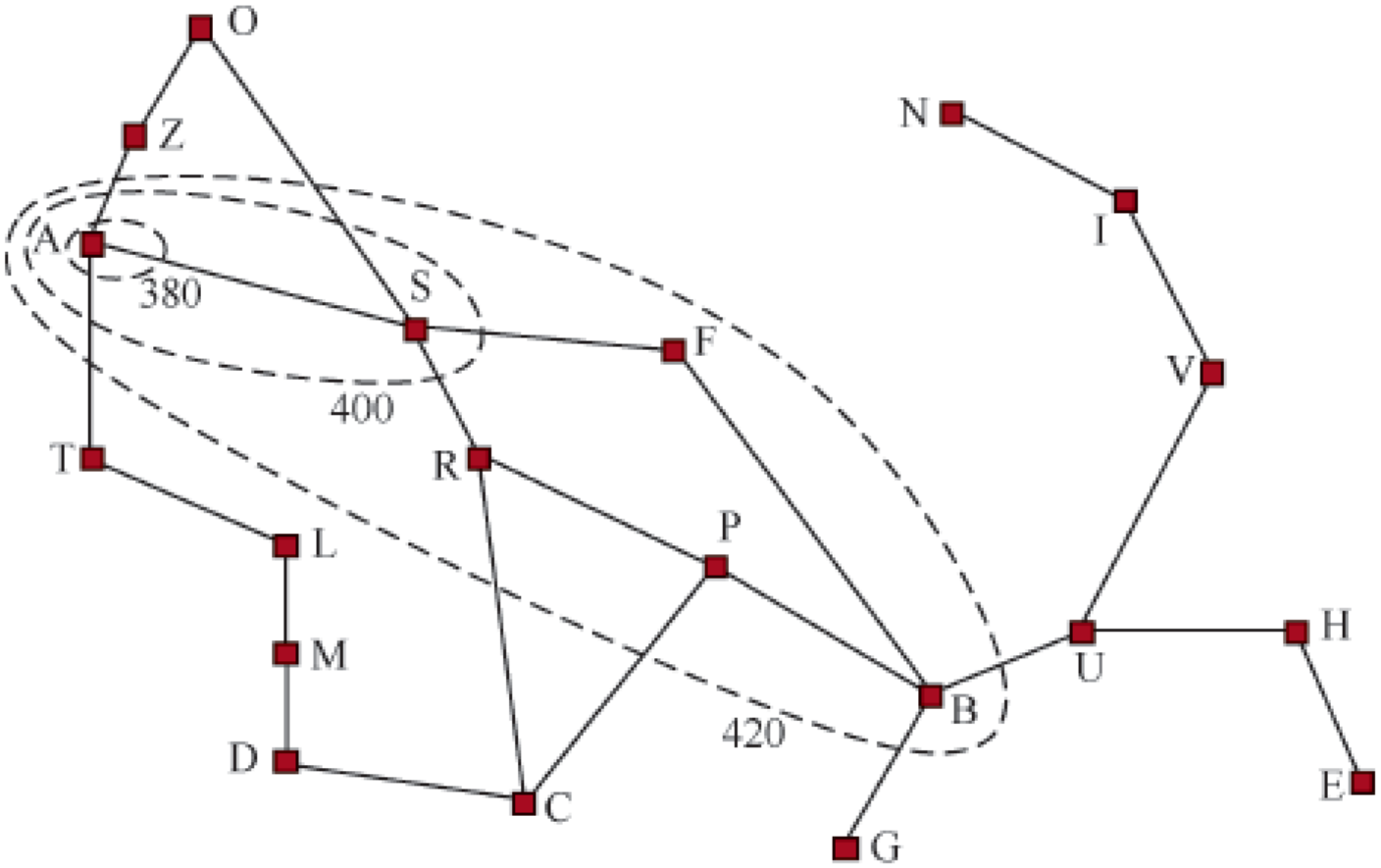
$$
\begin{aligned}
f(n') &= g(n') + h(n') \\
&= g(n) + c(n, a, n') + h(n') \\
&\geq g(n) + h(n) \\
&= f(n)
\end{aligned}
$$

3.6.2 A*搜索 –最优性证明

- ②如A*选择扩展结点n时，就已经找到到达结点n的最优路径。
  - 如果不成立，则在到达n的最优路径上就会存在另一个边缘结点n'。因为f在任何路径上都是非递减的，n'的f代价比n小，会被先选择。

3.6.2 A*搜索

- 罗马尼亚地图，其中等值线为f=380、f=400和f= 420，初始状态为 Arad。给定等值线内的节点的代价f=g+h小于或等于等值线值。



3.6.2 A*搜索

- 如果C*是最优解路径的代价值，则A*算法不会扩展f(n)>C*的结点。这样的结点被剪枝——无需检验就直接把它们从考虑中排除——在AI的很多领域中都是很重要的。
- 对从根节点开始扩展搜索解路径的算法中，A*算法对于任何给定的一致的启发式函数都是效率最优的。
- A*算法不一定就是我们需要的答案（**不一定合适**）。
  - 空间复杂度与时间复杂度；
  - 许多问题所扩展的节点数可能是解路径长度的**指数级**。考虑一个具有超强吸力的真空吸尘器世界，它以单位代价清理任一方格却不需要访问该方格。在这种情况下，可以按任何顺序清理方格。如果开始时有N个脏的方格，则有2^N种状态，其中某个子集已被清理；所有这些状态都在最优解路径上，因此满足f(n) <C*，所以所有这些状态都会被A*搜索访问。

3.6.3 存储受限的启发式搜索

- **迭代加深A*算法（IDA*）：**
  - 截断值用 f 代价（g+h）代替搜索深度。
- **递归最佳优先搜索（RBFS）：**
  - 从当前结点的祖先可得到最佳可选路径的 f 值。如果当前结点超过了这个限制，递归将回到可选路径上；
  - 空间复杂度是最佳解路径所在深度的**线性关系**。

3.6.3 存储受限的启发式搜索

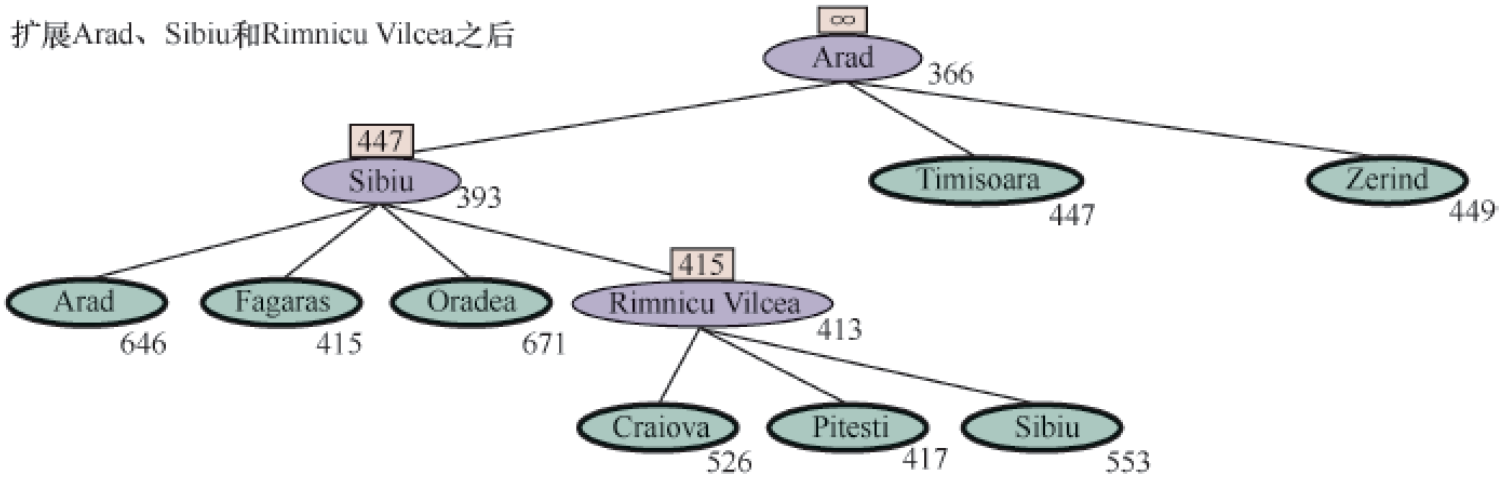
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns 一个解或者failure
    solution, fvalue ← RBFS(problem, NODE(problem.INITIAL), ∞)
  return solution
function RBFS(problem, node, f_limit) returns 一个解或failure，以及一个新的f代价限制
  if problem.IS-GOAL(node.STATE) then return node
  successors ← LIST(EXPAND(node))
  if successors为空 then return failure, ∞
  for each s in successors do          //用前一次搜索中的值更新f
      s.f ← max(s.PATH-COST + h(s), node.f)
  while true do
      best ← successors 中f值最低的节点
      if best.f > f_limit then return failure, best.f
      alternative ← successors中第二低的f值
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result, best.f
```

图 3-22　递归最佳优先搜索算法

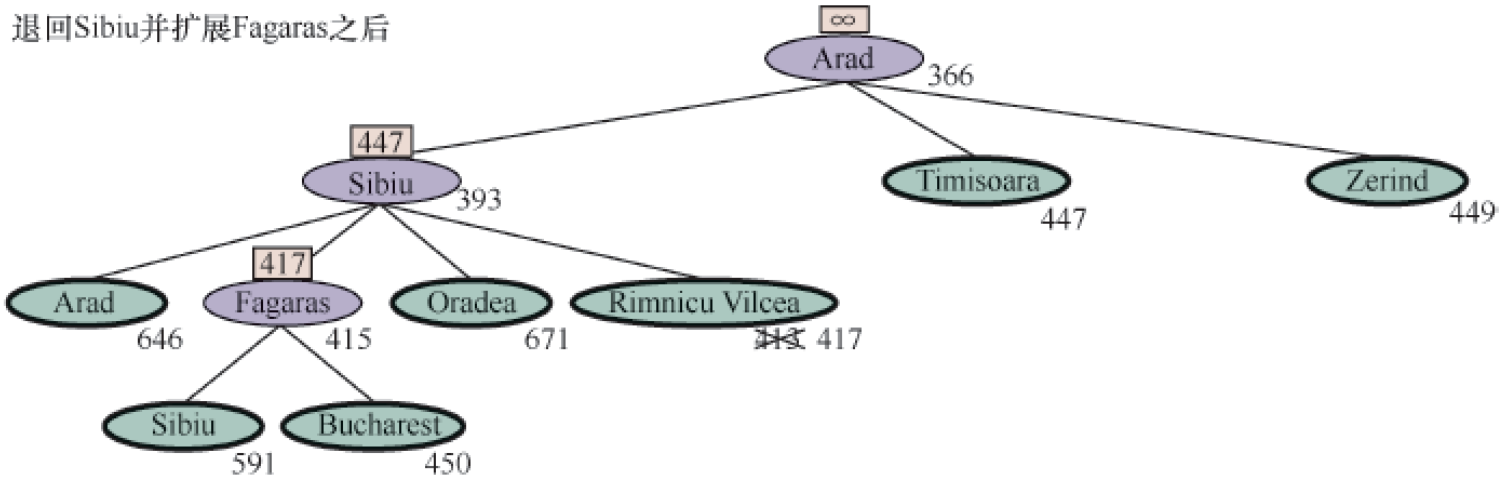
3.6.3 存储受限的启发式搜索

- 递归最佳优先搜索（RBFS）：

（a）扩展Arad、Sibiu和Rimnicu Vilcea之后

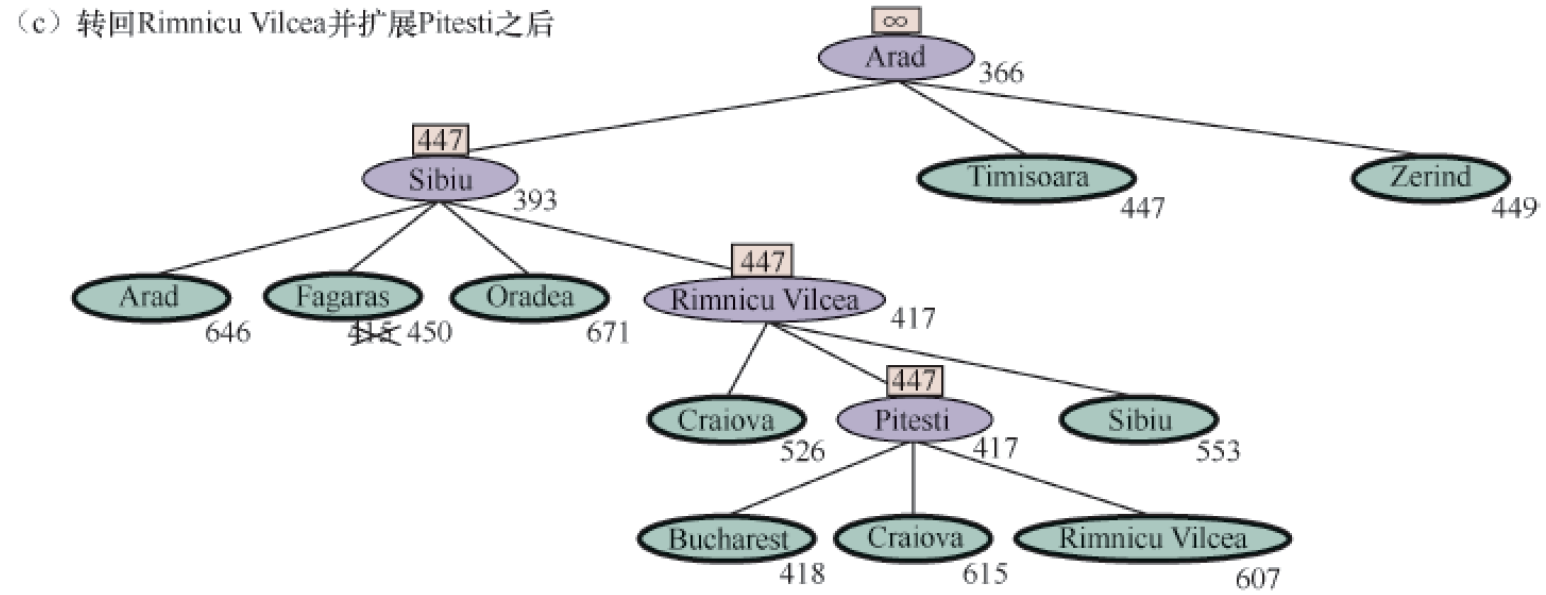


（b）退回Sibiu并扩展Fagaras之后



3.6.3 存储受限的启发式搜索

- 递归最佳优先搜索（RBFS）：



3.6.4 学习以促搜索

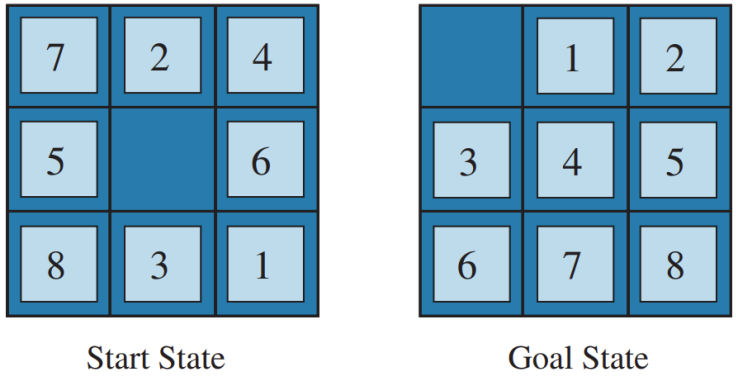
- **固定的搜索策略：**
  - 宽度优先；
  - 贪婪最佳优先。

- **有其他更好的搜索策略吗？**
  - 从经验中学习，学习已有实例，尝试构造函数h，这些方法中的大多数学习到的都是启发式函数的一个不完美的近似，因此存在启发式函数不可容许的风险。这必然导致算法需要在学习时间、搜索运行时间和解的代价之间进行权衡；
  - 例如可变的系数c_1与c_2： $h(n) = c_1 x_1(n) + c_2 x_2(n)$

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.7 启发式函数

- **八数码问题：**
  - 一个随机产生的八数码问题的平均解步数是22步；
  - 分支因子约为3；
  - 深度为22的穷举搜索树考虑大约$3^{22} \approx 3.1*10^{10}$个状态；
  - **两个常用的启发式函数：**
    - h_1=不在位的棋子数；
    - h_2=所有棋子到其目标位置的距离和。



典型的八数码问题实例；它的解路径为26步。

3.7 启发式函数 –启发式的精确度对性能的影响

有效分支因子b*:

对于某一个问题，如果A*算法生成的总结点数为n，解的深度为d，那么b*就是深度为d的标准搜索树为了能够包括n+1个结点所必须的分支因子。即:

$$n+1 = 1 + b^* + (b^*)^2 + \cdots + (b^*)^d$$

- 如果A* 用52 个节点在第5 层上找到了一个解，那么有效分支因子是1.92。

3.7 启发式函数 –启发式的精确度对性能的影响

- ITERATIVE-DEEPENING-SEARCH以及使用h_1和h_2的A*算法的搜索代价和有效分支因子的比较。图中的数据是通过八数码问题实例计算的平均值，对于解路径的不同深度d，每种深度上都选取100个问题实例：

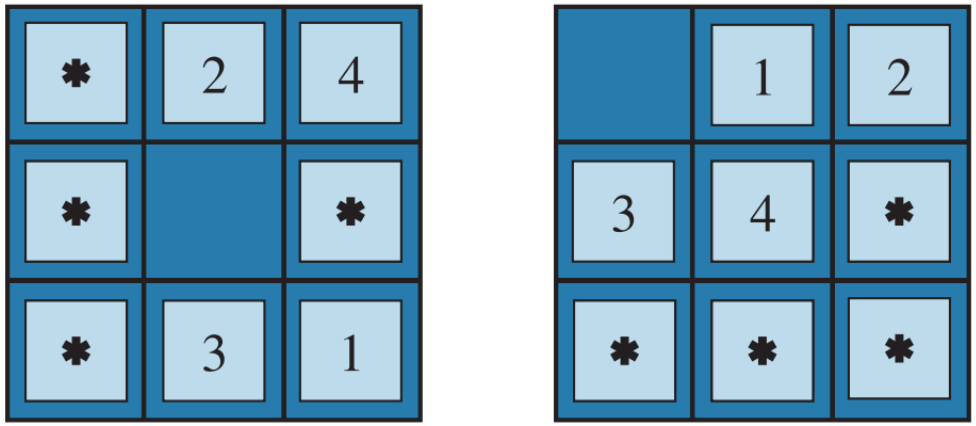
	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

3.7 启发式函数 –从松弛问题出发设计可采纳的启发式

- h_1:
  - 如果游戏的规则改变为每个棋子可以随便移动，而不是只能移动到与其相邻的空位上。
- h_2:
  - 如果一个棋子可以向任意方向移动一步，甚至可以移到已经被其他棋子占据的位置上。
- 松弛问题：减少了行动限制的问题；
- 松弛问题的状态空间图是原有状态空间的超图；
- 松弛问题的最优解代价是原问题的**可采纳的启发式**；
- 如果可采纳启发式的集合$h_1 \ldots h_m$对问题是有效的，那么$h(n)=\max\{h_1(n),\ldots h_m(n)\}$是一个更好的新的启发式。

3.7 启发式函数 –从子问题出发设计可采纳的启发式

- 模式数据库：对每个可能的子问题实例存储解代价：



Start State　　Goal State

如图是八数码问题的一个子问题。任务是将棋子1、2、3和4移到正确位置上，而不考虑其他棋子的情况.

3.7 启发式函数 –从经验中学习启发式

- 通过学习算法构造h(n):
  - 神经网络
  - 决策树
  - 等等

- 概述
- 问题求解Agent
- 问题实例
- 通过搜索求解
- 无信息搜索策略
- 有信息（启发式）的搜索策略
- 启发式函数
- 本章小结

3.8 本章小结

- 确定性的、可观察的、静态的和完全可知的环境下，Agent可以用来选择行动的方法；
- Agent可以构造行动序列以达到目标，这个过程称为搜索；
- 无信息搜索：
  - 宽度优先、一致代价、深度优先、迭代加深、双向搜索……
- 有信息搜索：
  - 启发式函数、A*搜索、迭代加深A*搜索……
- 启发式函数构建。

3.8 本章小结 –课程作业

- 你的目标是让机器人走出迷宫。机器人面朝北，开始位置在迷宫中间。你可以让机器人转向面朝东、南、西或北。你可以让机器人向前走一段距离，在撞墙之前它会停步。
  - 将问题形式化。状态空间有多大?
  - 在迷宫中游走，在两条路或更多路交叉的路口可以转弯。重新形式化这个问题。现在状态空间有多大?
  - 从迷宫的任一点出发，我们可以朝四个方向中的任一方向前进直到可以转弯的地方，而且我们只需要这样做。重新对这个问题进行形式化。我们需要额外记录机器人的朝向吗?
  - 在我们对问题的最初描述中已经对现实世界进行了抽象，限制了机器人的行动并移除了细节。列出三个我们做的简化。

3.8 本章小结 –课程作业

- 传教士和野人问题：

三个传教士和三个野人在河的一岸，有一条能载一个人或者两个人的船。请设法使所有人都渡到河的另一岸，要求在任何地方野人数都不能多于传教士的人数（传教士可以为0）。

- 请对该问题进行详细形式化，只描述确保该问题求解所必需的特性。画出完整的状态空间图
- 应用合适的搜索算法求出该问题的最优解，描绘出搜索过程。对于这个问题检查重复状态是个好主意吗？

THANKS