人工智能——盲目搜索

饶洋辉
数据科学与计算机学院，
中山大学
raoyangh@mail.sysu.edu.cn
http://sdcs.sysu.edu.cn/node/2471

搜索

- 搜索可以解决的问题
- 盲目（无信息）搜索
- 启发式搜索

Readings: Chapter 3

*Slides based on those of Sheila McIlraith

为什么需要搜索?

- 在游戏策略中取得较好的结果
- 其他AI问题中也可以应用

搜索问题的形式化定义

我们需要考虑下面几个部分来对搜索问题进行形式化定义：

- 状态空间(state space)：表示需要进行搜索的空间。状态空间是对问题的形式化
- 动作(action)：表示从一个状态到另一个状态。动作是对真实的动作的形式化
- 初始状态(initial state)：当前状态的表示
- 目标(goal)：需要达到的目标状态的表示
- 启发方法(heuristics)：用于指挥搜索的前进方向

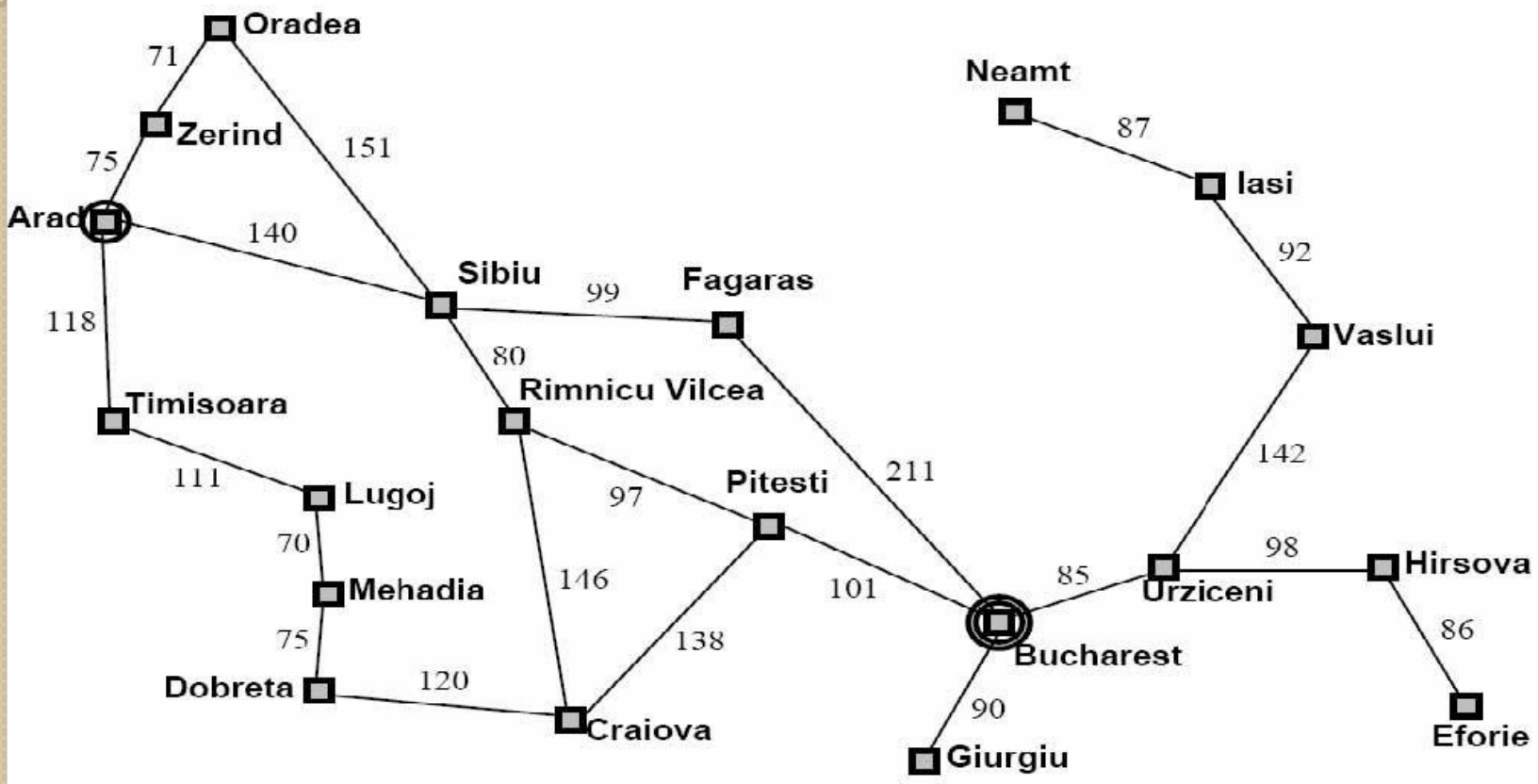
搜索问题的形式化定义

- 把问题形式化为对状态空间的搜索问题之后，就可以使用各种搜索算法解决问题。

- 问题的解是由“动作”构成的序列。序列中的动作可以将初始状态转化为目标状态。

例1：罗马尼亚旅行问题

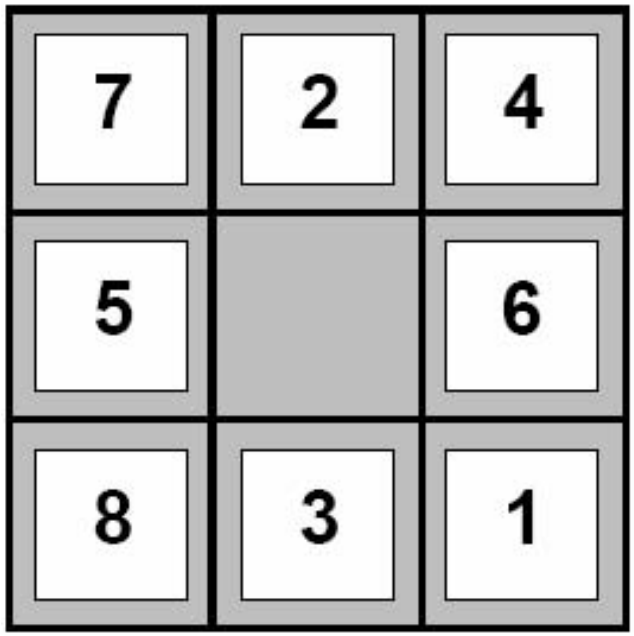
当前位于Arad，需要到达Bucharest



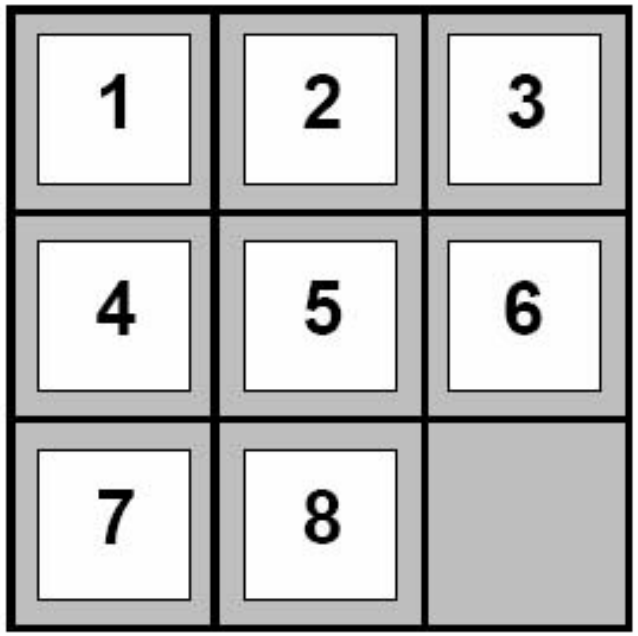
例1：罗马尼亚旅行问题

- 状态：可以到达的任一个城市
- 动作：在相邻的城市之间移动
- 初始状态：在城市Arad
- 目标状态：在城市Bucharest
- 问题的解：旅行路线，从Arad到Bucharest途径城市组成的序列

例2：八数码问题



Start State



Goal State

规则：可以把数字移到空格里（也可以视为移动空格到临近的数字的位置）

例2：八数码问题

- 状态：各种不同的方格摆放方式
- 动作：向上/下/左/右移动空格，但是不是所有位置都能完成四种动作
- 初始状态：前面左图所示的方格摆放方式
- 目标状态：前面右图所示的方格摆放方式
- 问题的解：移动空格的步骤组成的序列

例3：吸尘器世界

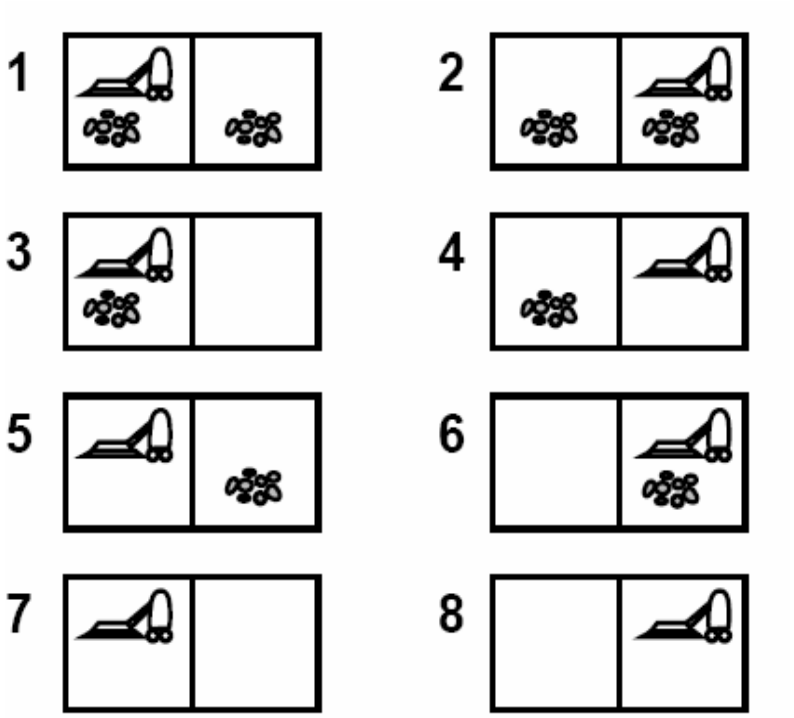
- 前面的示例中，一个状态是对应一个真实世界中的状态
- 然而，一个状态也可以对应智能体如何认识世界的情况：智能体的认知状态（the agent's knowledge state）

例3：吸尘器世界

问题描述：

- 有一个吸尘器需要清扫两个房间
- 每个房间有两个状态：

  干净或不干净
- 吸尘器有向左(left)和向右(right)两个动作（左边/右边没有房间时动作不起作用）
- 吸尘器有吸尘(suck)的动作，该动作使得吸尘器所在的房间状态变为干净（即使房间本身就是干净的）

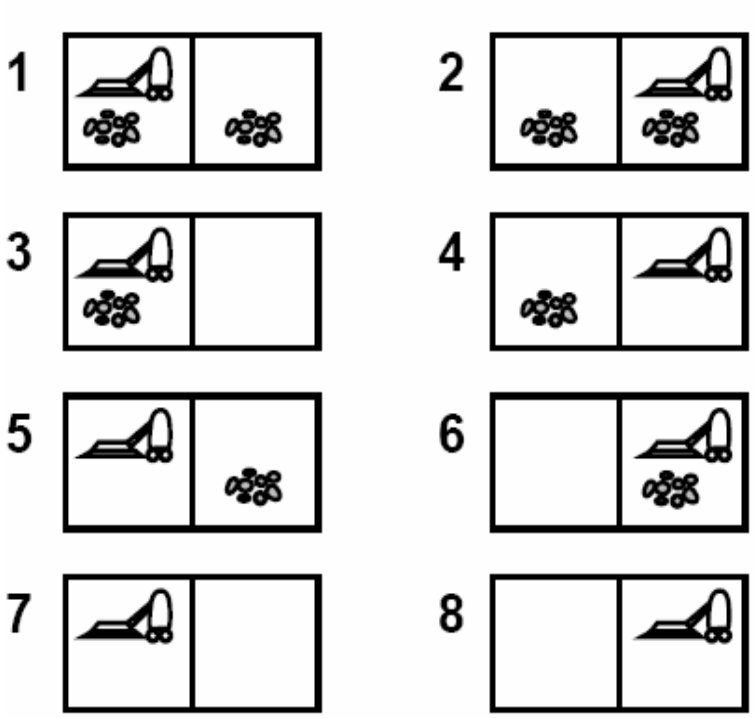


物理状态

例3：吸尘器世界

认知层面的状态空间

- 一个认知状态是物理状态空间的一个子集。也就是智能体知道自己处于几个物理状态中的一个状态中，但是不知道具体是哪一个。



目标是是所有房间的状态都为干净

例3：吸尘器世界

认知层面的状态空间

- **完全不清楚物理世界**

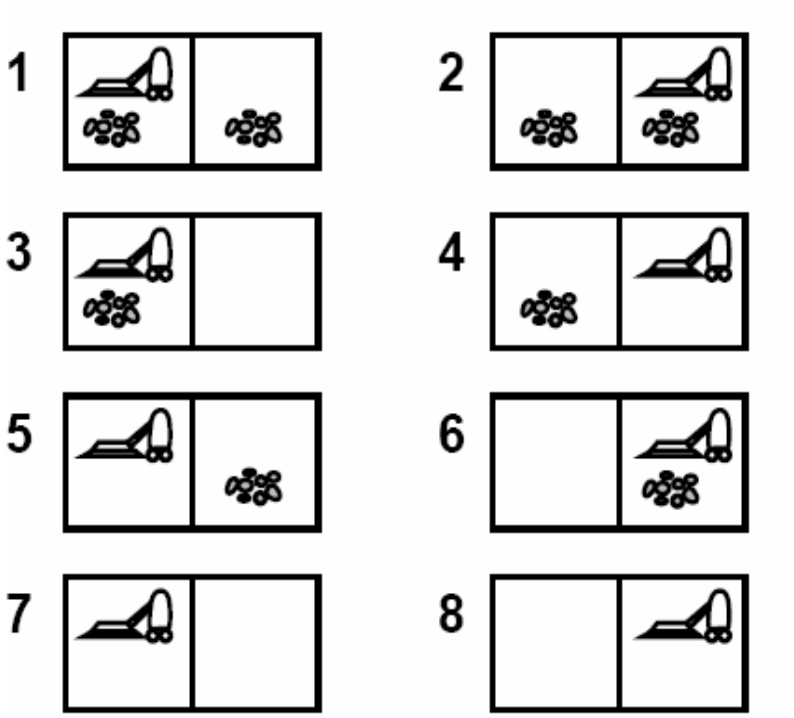
  认知状态是物理状态的集合

- **初始认知状态为{1,2,3,4,5,6,7,8}**

  智能体不知道自己身处具体是哪个物理状态

- 但不论如何，动作序列 <right, suck, left, suck> 都可到达目标，认知状态的变化情况如下

  {1,2,3,4,5,6,7,8} right→ **{2, 4, 6, 8}** suck → **{4, 8}** left → **{3, 7}** suck → **{7}**



目标是是所有房间的状态都为干净

搜索算法

算法输入:

- 具体的**初始状态**：一个具体的物理状态，或是一个物理状态的集合表示的智能体的认知状态，等等
- **后继函数**：S(x) = {x状态经过一个动作之后可以到达的状态的集合}.
- **目标测试**：一个作用于状态上，当该状态满足目标条件时返回真的函数
- **前进成本**：C(x,a,y) = 从x状态通过动作a到达y状态所需要的成本. (x状态无法到达y状态时，$\forall_a$C(x,a,y) = ∞)

算法输出:

- 从初始状态到某个满足目标测试的状态的状态序列

获取动作系列

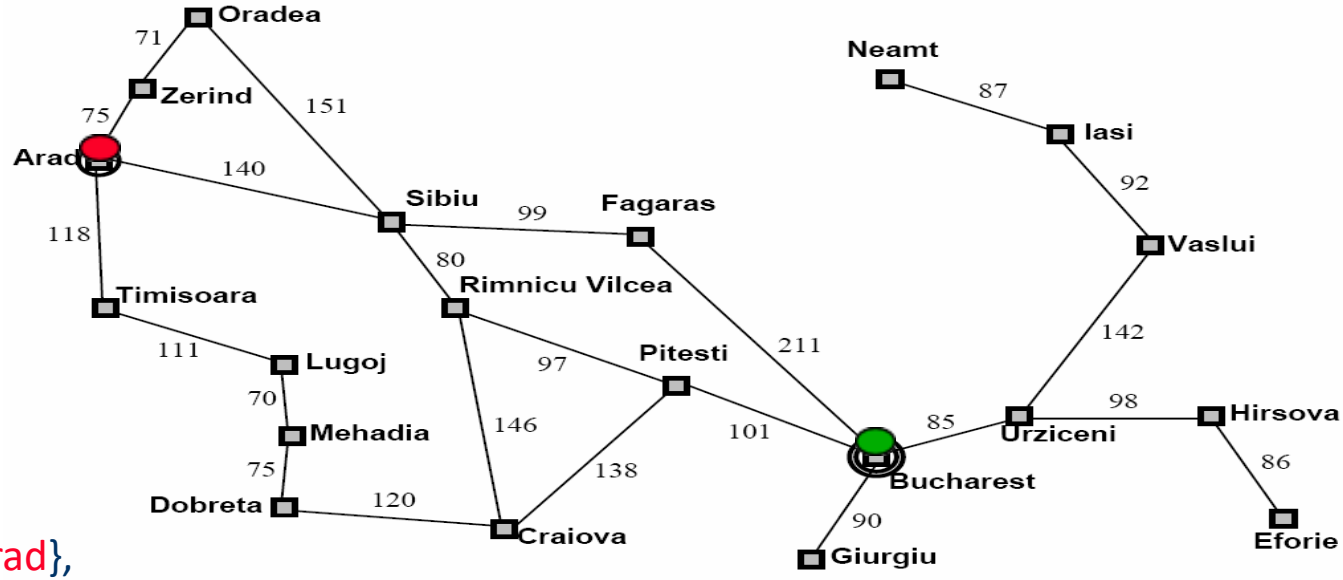
- 状态x的后继状态可能来自不同的动作，e.g.,
  - x → a → y
  - x → b → z
- 后继函数 S(x) 返回的状态集合中的状态是状态x通过任何一个动作能达到的状态，因此需要把来自不同动作的后继状态加以区分
  - 因此修改S(x)的返回值，不仅包括后继状态，还要记录获得这个后继状态所经过的动作
    - S(x) = {<y,a>, <z,b>}
      状态y 通过动作a得到，状态z 通过动作b得到.
    - S(x) = {<y,a>, <y,b>}
      状态y 通过动作a得到，状态y 通过动作b得到.

算法框架（Tree search）

- 边界：还没有被探索，但准备下一步探索的状态的集合
- 初始边界 = 初始状态集合

```
TreeSearch(Frontier, Sucessors, Goal? )
    If Frontier is empty return failure
    Curr = select state from Frontier
    If (Goal?(Curr)) return Curr.
    Frontier' = (Frontier – {Curr}) U Successors(Curr)
    return TreeSearch(Frontier', Successors, Goal?)
```

示例：罗马尼亚旅行问题



{Arad},

{**Z<A>, T<A>, S<A>**},

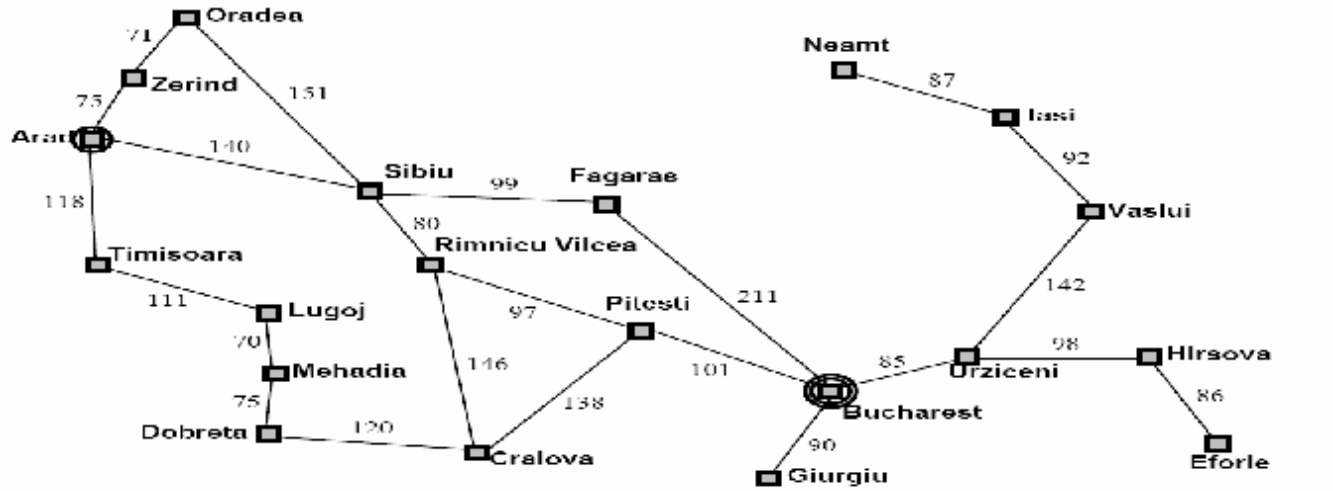
{Z<A>, T<A>, **A<S;A>, O<S;A>, F<S;A>, R<S;A>**}

{Z<A>, T<A>, A<S;A>, O<S;A>, R<S;A>, **S<F;S;A>, B<F;S;A>**}

Solution: Arad -> Sibiu -> Fagaras -> Bucharest

Cost: 140 + 99 + 211 = 450

罗马尼亚旅行问题的另一个解



{Arad}

{Z<A>, T<A>, S<A>},

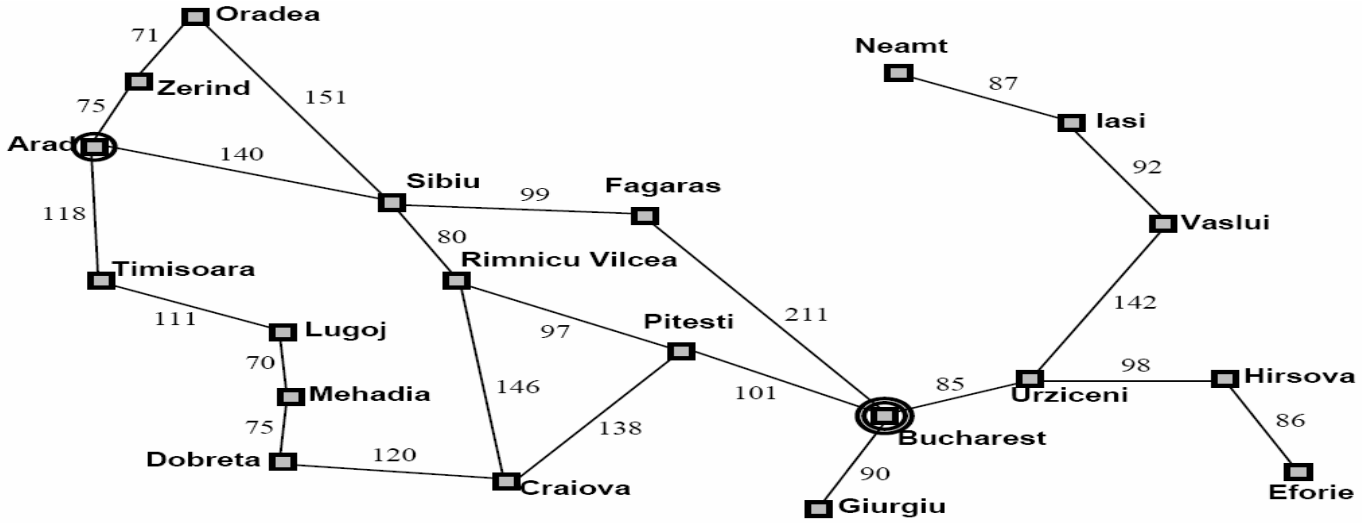
{Z<A>, T<A>, **A<S,A>, O<S,A>, F<S,A>, R<S,A>**}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, **S<R,S,A>, P<R,S,A>, C<R,S,A>**}

{Z<A>, T<A>, A<S,A>, O<S,A>, F<S,A>, S<R,S,A>, C<R,S,A>, **R<P,R,S,A>, C<P,R,S,A>, Bucharest<P,R,S,A>**}

Solution: Arad -> Sibiu -> Rimnicu Vilcea -> Pitesti -> Bucharest
Cost: 140 + 80 + 97 + 101 = 418

循环问题



{Arad}

{Z<A>, T<A>, S<A>},

{Z<A>, T<A>, **O<S;A>, F<S;A>, A<S;A>, R<S;A>**}

{Z<A>, T<A>, O<S;A>, F<S;A>, R<S;A>, **Z<A;S;A>, T<A;S;A>, S<A,S,A>**}

.....

边界不是状态的集合而是路径的集合，因此只要路径不同就会往边界上添加新的元素，导致了循环问题

选择法则

上面的例子表明了状态选择的顺序会对搜索操作产生重要的影响:

- 会影响搜索是否能找到解
- 会影响搜索到的节的成本大小
- 会影响搜索过程中所需要的时间和空间资源

搜索算法的重要特征

- 完备性（Completeness）：搜索算法是否总能在问题存在解的情况下找到解
- 最优性（Optimality）：当问题中的动作是需要成本时，搜索算法是否总能找到成本最小的解
- 时间复杂度（Time complexity）：搜索算法最多需要探索/生成多少个节点来找到解
- 空间复杂度（Space complexity）：搜索算法最多需要将多少个节点储存在内存中

盲目搜索策略

- 这些策略都采用固定的规则来选择下一需要被扩展的状态
- 这些规则不会随着要搜索解决的问题的变化而变化
- 这些策略不考虑任何与要解决的问题领域相关的信息

常用的盲目搜索方法

- 宽度优先（Breadth-First）
- 深度优先（Depth-First ）
- 一致代价（Uniform-Cost）
- 深度受限（Depth-Limited）
- 迭代加深搜索（Iterative-Deepening search）

选择 vs. 排序

- 通用的选择方式：
  - 对边界上的元素进行排序
  - 总是选择第一个元素
- 任何选择规则都可以视为对边界采用某种合适的排序方式

宽度优先

- 把当前要扩展的状态的后继状态放在边界的最后
- 例子:
  - 假设使用正整数表示状态 {0,1,2,…}
  - 状态n的后继状态为状态n+1和状态n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - 初始状态为 0
  - 目标状态为5

示例

{0<>}

{1,2}

{2,2,3}

{2,3,3,4}

{3,3,4,3,4}

{3,4,3,4,4,5}

…

宽度优先的性质

$b =$ 问题中一个状态最大的后继状态个数

$d =$ 最短解的动作个数

完备性和最优性

宽度优先搜索具有完备性和最优性

- 短的路径会在任何比它长的路径之前被遍历
- 给定路径长度，该长度的路径是有限的
- 最终可以遍历所有长度为d的路径，因此一定可以找出最短的解

时间和空间复杂度

- 时间复杂度: $1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- 空间复杂度: $b(b^d - 1) = O(b^{d+1})$

空间复杂度会带来较大的问题

- E.g., 假设 b = 10, 并且每秒扩展1000个节点，每个节点需要100bytes来存储:

Depth	Nodes	Time	Memory
1	1	1 millisec.	100 bytes
6	10^6	18 mins.	111 MB
8	10^8	31 hrs.	11 GB

- 实际情况下，内存需求会先于时间限制算法的运行

深度优先

- 把当前要扩展的状态的后继状态放在边界的最前面
  边界上总是扩展最深的那个节点

与宽度优先搜索的示例比较：

{0}

{1,2}

{2,3,2}

{3,4,3,2}

{4,5,4,3,2}

{5,6 4,5,4,3,2}

…

深度优先的性质

完备性:

- 在状态空间无限的情况下: No
- 在状态空间有限，但是存在无限的路径（例如存在回路）的情况下: No
- 在状态空间有限，且对重复路径进行剪枝的情况下：Yes

最优性: No

时间复杂度

时间复杂度为：$O(b^m)$
其中m是遍历过程中最长路径的长度 (Could explore each branch of search tree)

当m远远大于d时，时间效率会很差

当存在多条解路径的情况下深度优先搜索可以比宽度优先搜索更快找到解 (可以碰运气先遍历了到达解的那条路径).

空间复杂度

深度优先回溯点 = 当前路径上的点的未扩展过的兄弟节点

一次只会考虑一条路径

边界上只包含当前探索的最深的节点，以及回溯点

$O(bm)$, 线性复杂度! 是深度优先搜索一个显著的优点

一致代价搜索

- 边界中，按路径的成本升序排列
- 总是扩展成本最低的那条路径
- 当每种动作的成本是一样的时候，和宽度优先是一样的

完备性和最优性

- 假设每个动作的成本 ≥ $s > 0$
- 一致代价搜索中，所有成本较低的路径都会在成本高的路径之前被扩展
- 给定成本，该成本的路径数量是有限的
- 成本小于最优路径的路径数量是有限的
- 最终，我们可以找到最短的路径

时间和空间复杂度

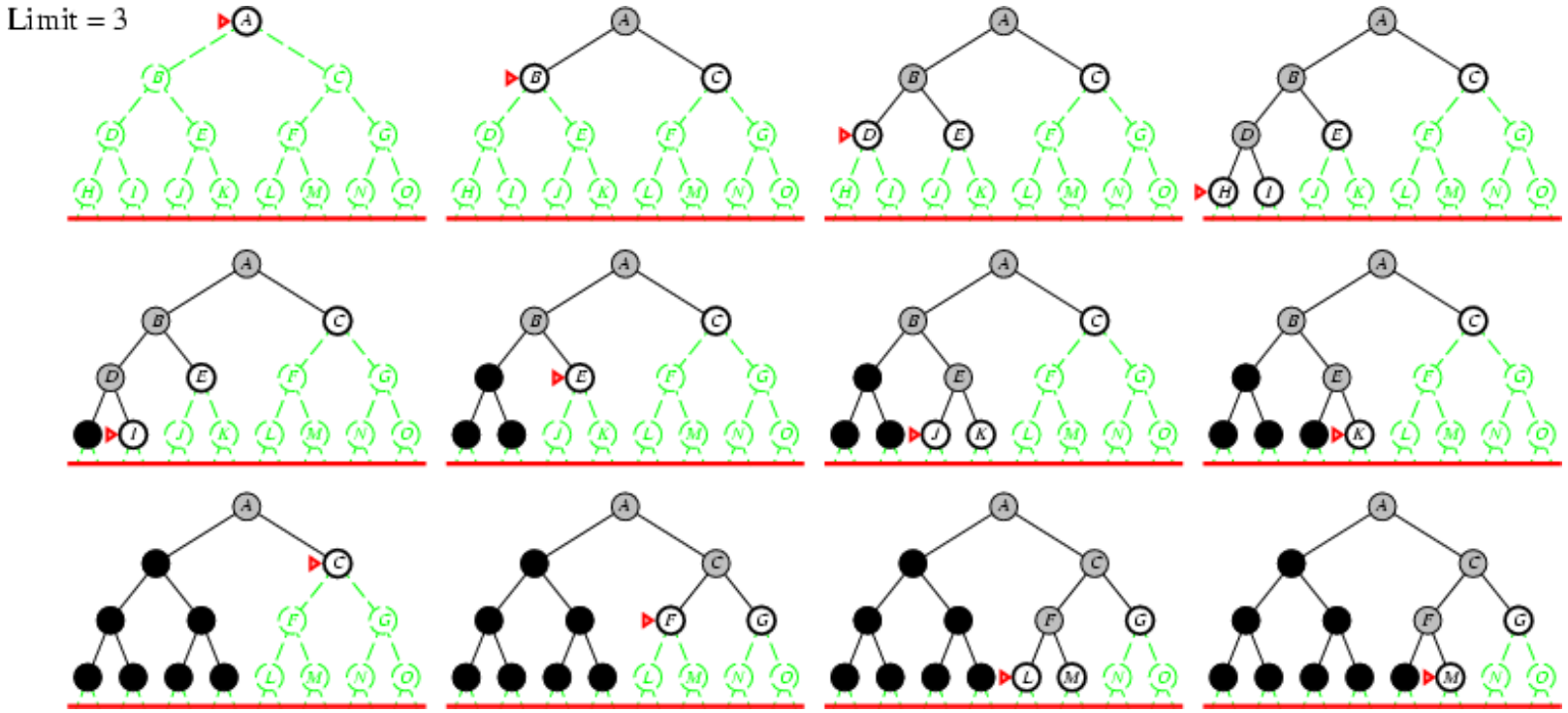
当最优解的路径长度为d时，宽度优先搜索的时间和空间复杂度都是 $O(b^{d+1})$

对于一致代价搜索，当最优解的成本为C^*，则时间和空间复杂度为$O(b^{C^*/s+1})$

深度受限搜索

- 宽度优先搜索存在空间复杂度过大的问题
- 深度优先搜索存在可能运行时间非常长，甚至在存在无限路径时无限运行下去的问题
- 深度受限搜索
  - 深度优先搜索，但是预先限制了搜索的深度 L
  - 因此无限长度的路径不会导致深度优先搜索无法停止的问题
  - 但只有当解路径的长度 $\leq L$ 时，才能找到解

示例



深度受限的性质

- 完备性: No
- 最优性: No
- 时间复杂度: $O(b^L)$
- 空间复杂度: $O(bL)$

迭代加深搜索

- 为了解决深度优先搜索和宽度优先搜索存在的问题
- 一开始设置深度限制为$L = 0$，我们迭代地增加深度限制，对于每个深度限制都进行深度受限搜索
- 如果找到解，或者深度受限搜索没有节点可以扩展的时候可以停止当前迭代，并提高深度限制L
- 如果没有节点可以被剪掉（深度限制不能再提高）仍然没有找到解，那么说明已经搜索所有路径，因此这个搜索不存在解

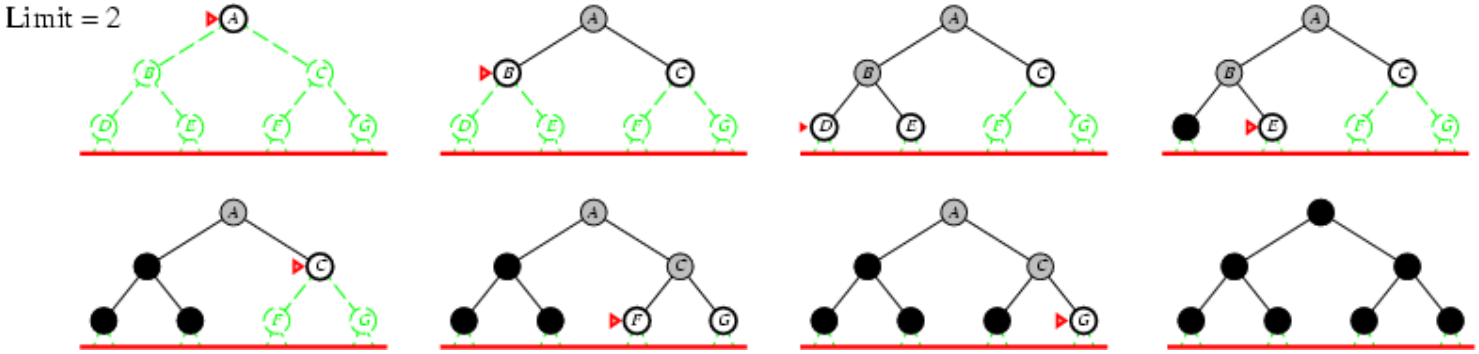
示例

Limit = 0

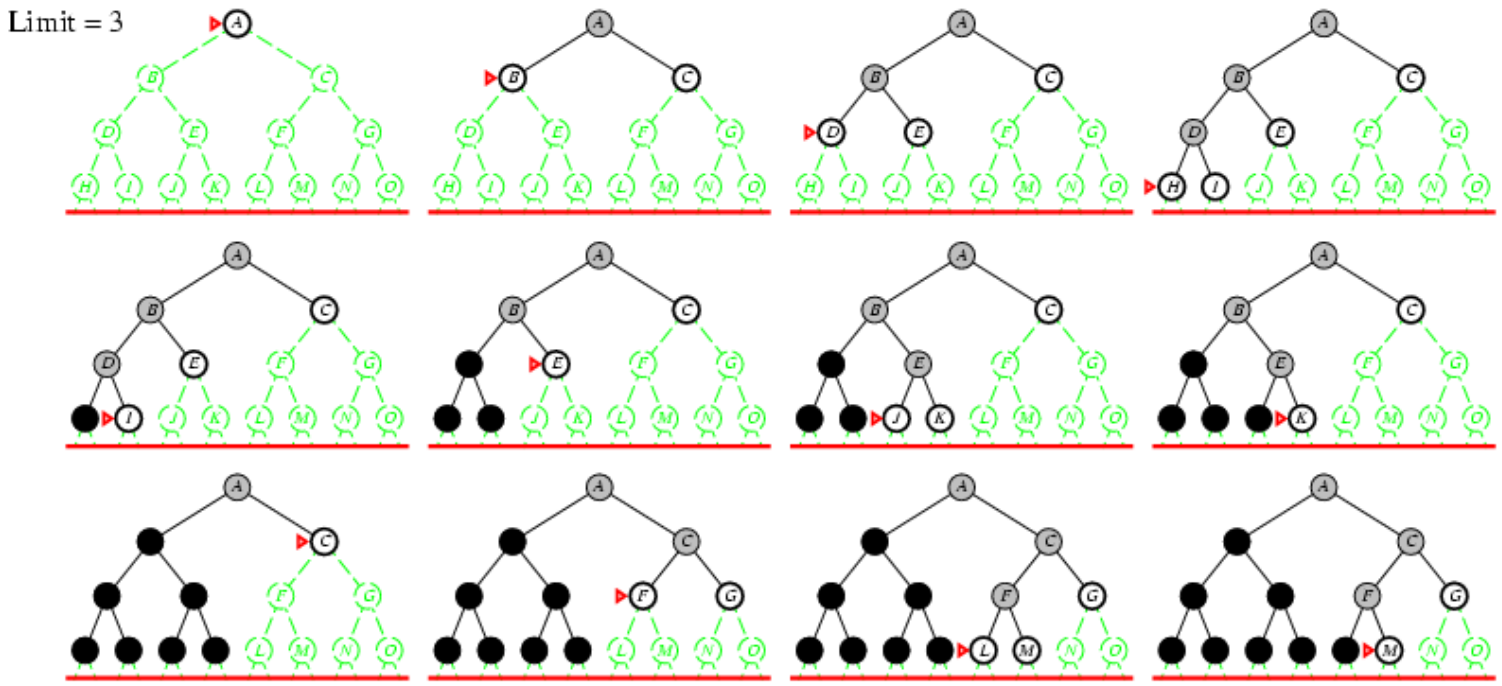
示例

Limit = 1

示例



示例



完备性和最优性

完备性：Yes
最优性：Yes （在每个动作的成本一致的情况下）

如果动作成本不一致，则可以使用成本边界(cost bound)代替深度限制L：

- 只扩展成本低于成本边界(cost bound)的路径
- 每次迭代时记录当前还未扩展路径中的最小成本
- 下一次迭代则提高成本边界

这样开销会很大，迭代数量为成本数值的构成的集合的大小

时间和空间复杂度

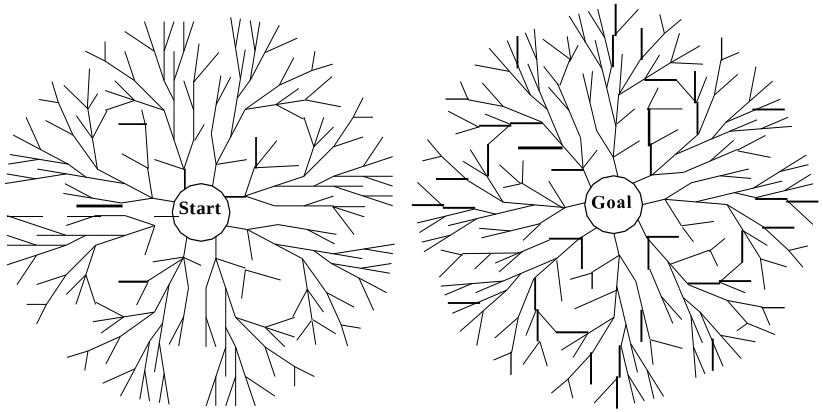
时间复杂度：$(d+1)b^0 + db + (d-1)b^2 + \ldots + b^d = O(b^d)$

对比宽度优先搜索的时间复杂度:
$1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$

迭代加深搜索可以比宽度优先搜索更高效: 不用扩展深度限制上的节点。 但是宽度优先搜索需要扩展直到目标节点。

空间复杂度：$O(bd)$

双向搜索



同时进行从初始状态向前的搜索和从目标节点向后搜索，在两个搜索在中间相遇时停止搜索

假设两个搜索都使用宽度优先搜索

- 完备性: Yes
- 最优性: Yes（在每条边/每个动作的成本一致的情况下）
- 时间和空间复杂度: $O(b^{d/2})$

盲目搜索总结

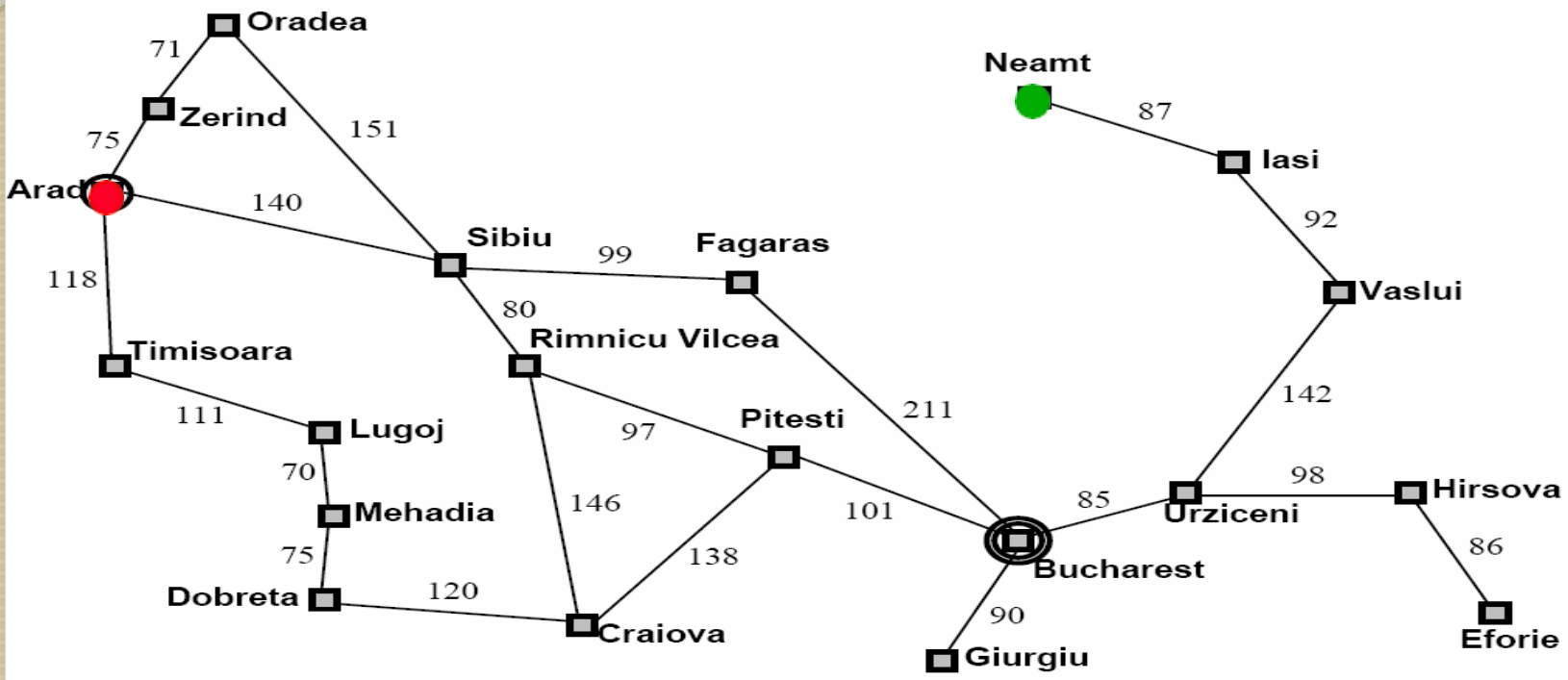
Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[a]	Yes[a,b]	No	No	Yes[a]	Yes[a,d]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon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

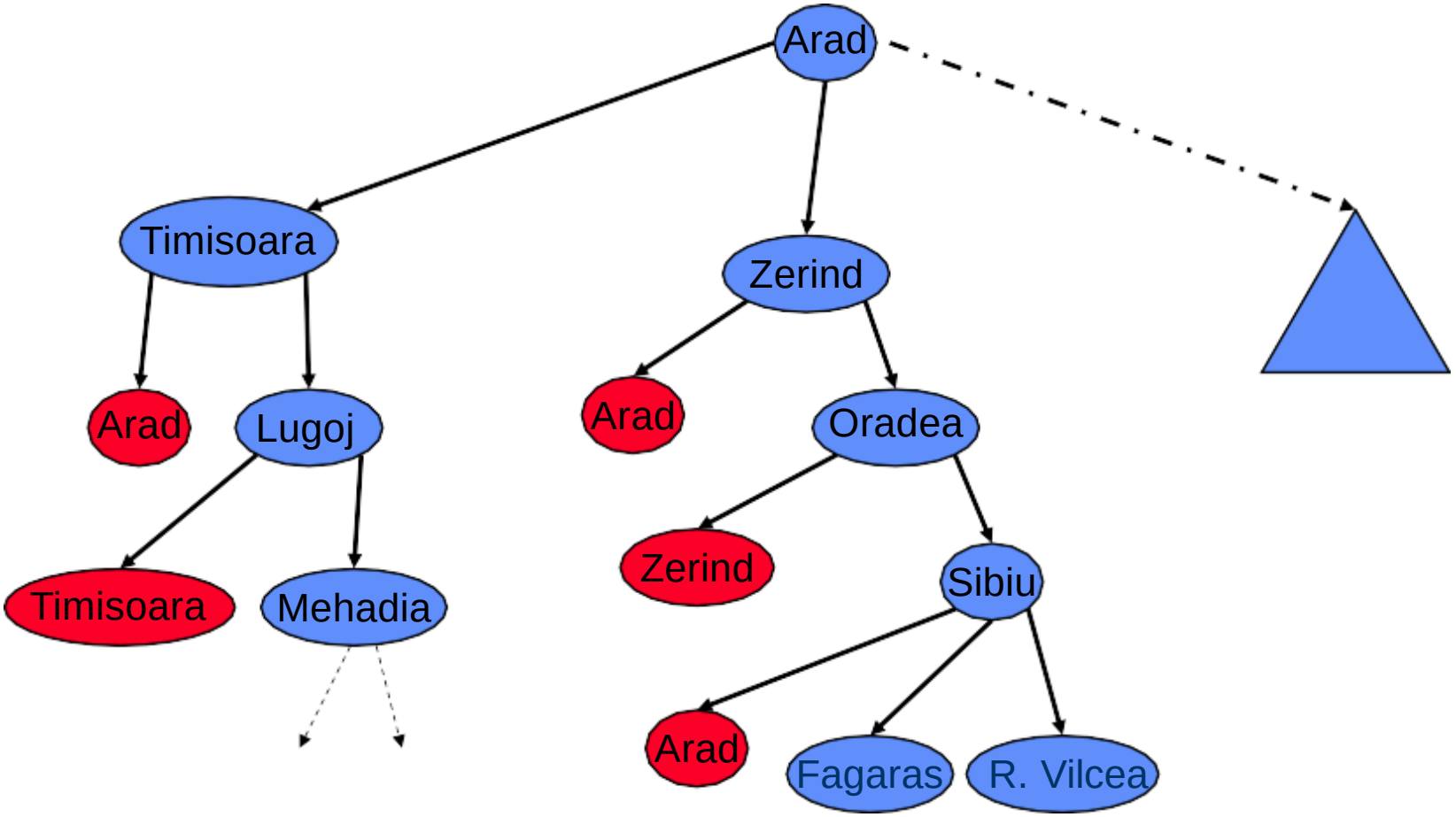
路径检测

- 在边界上保存了路径
- 假设 $(n_1, \ldots, n_k)$ 是一条到达节点 n_k的路径，并且我们要扩展节点 n_k 来获得子节点 c, 因此我们可以获得一条到达节点c 的路径 $(n_1, \ldots, n_k, c)$
- 路径检测用于确保状态（节点） c 与它所在路径上的祖先节点都不相等
- 也就是说，单独检测每条路径是否出现重复节点

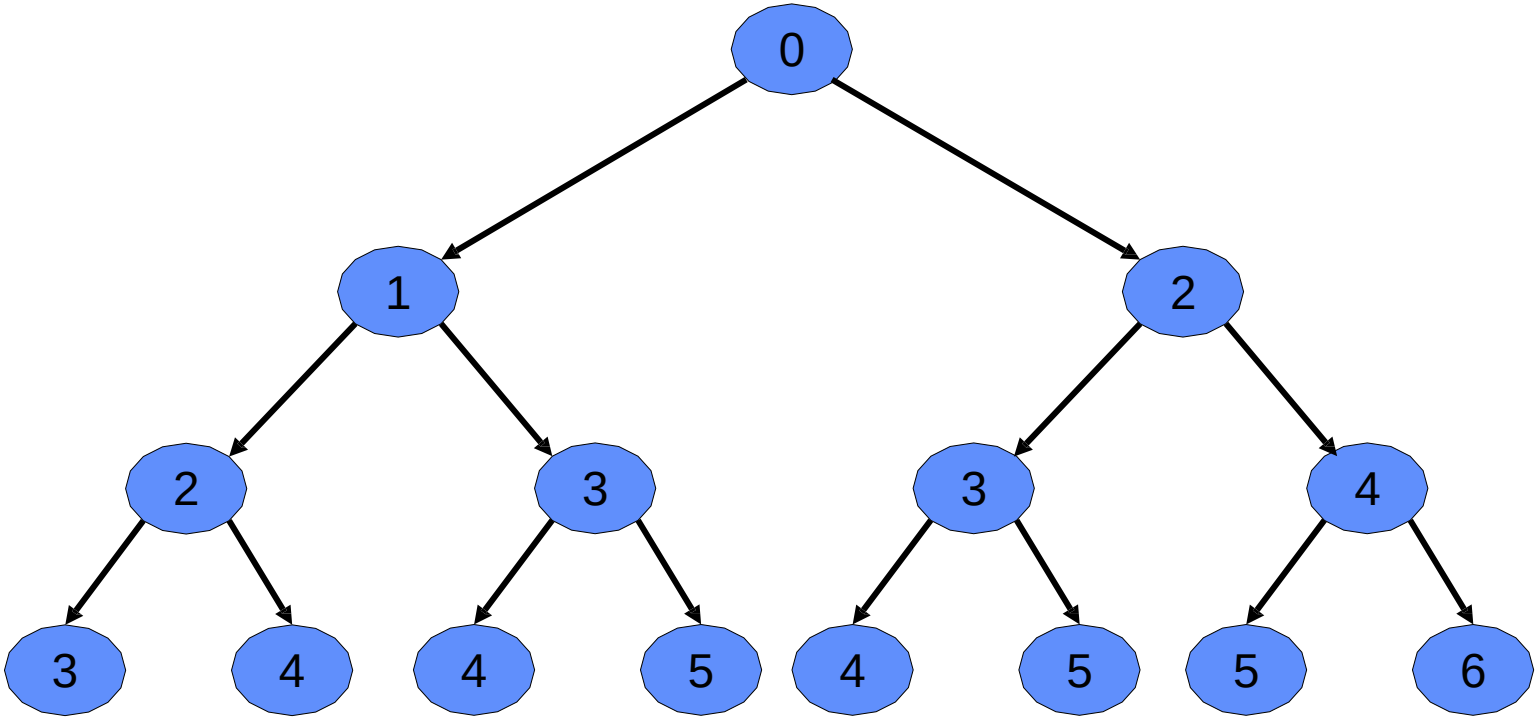
示例：Arad to Neamt



路径检测示例


Arad
Timisoara
Zerind
Arad
Lugoj
Arad
Oradea
Timisoara
Mehadia
Zerind
Sibiu
Arad
Fagaras
R. Vilcea

路径检测示例



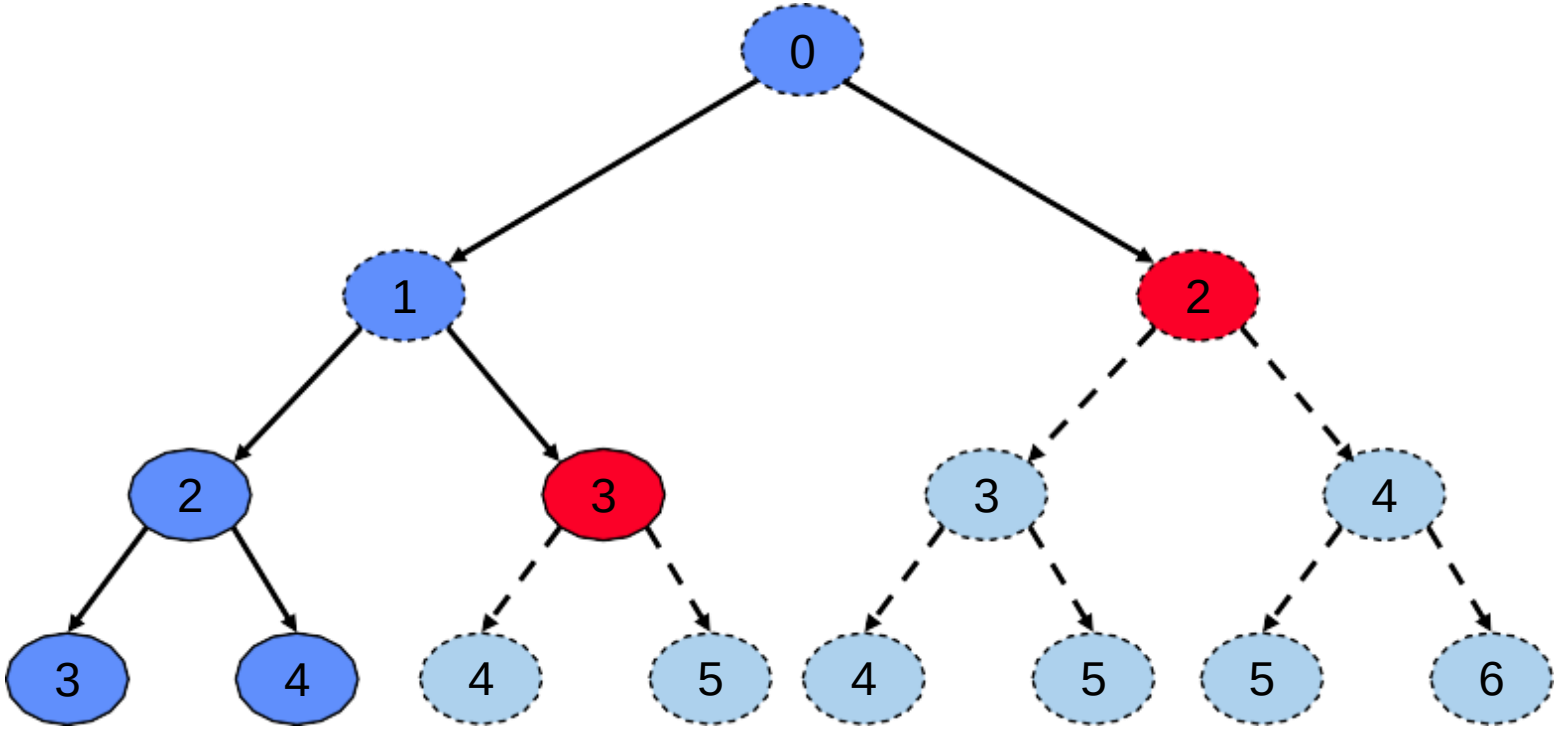
环检测/多路检测

记录下在之前的搜索过程中扩展过的所有节点
当扩展节点 n_k 获得子节点c时，确保节点c不等于之前任何扩展过的节点

不能将这个方法用于深度优先搜索，为什么?

因为会产生较高的空间复杂度，因此只能用于宽度优先搜索

环检测示例（深度优先搜索）



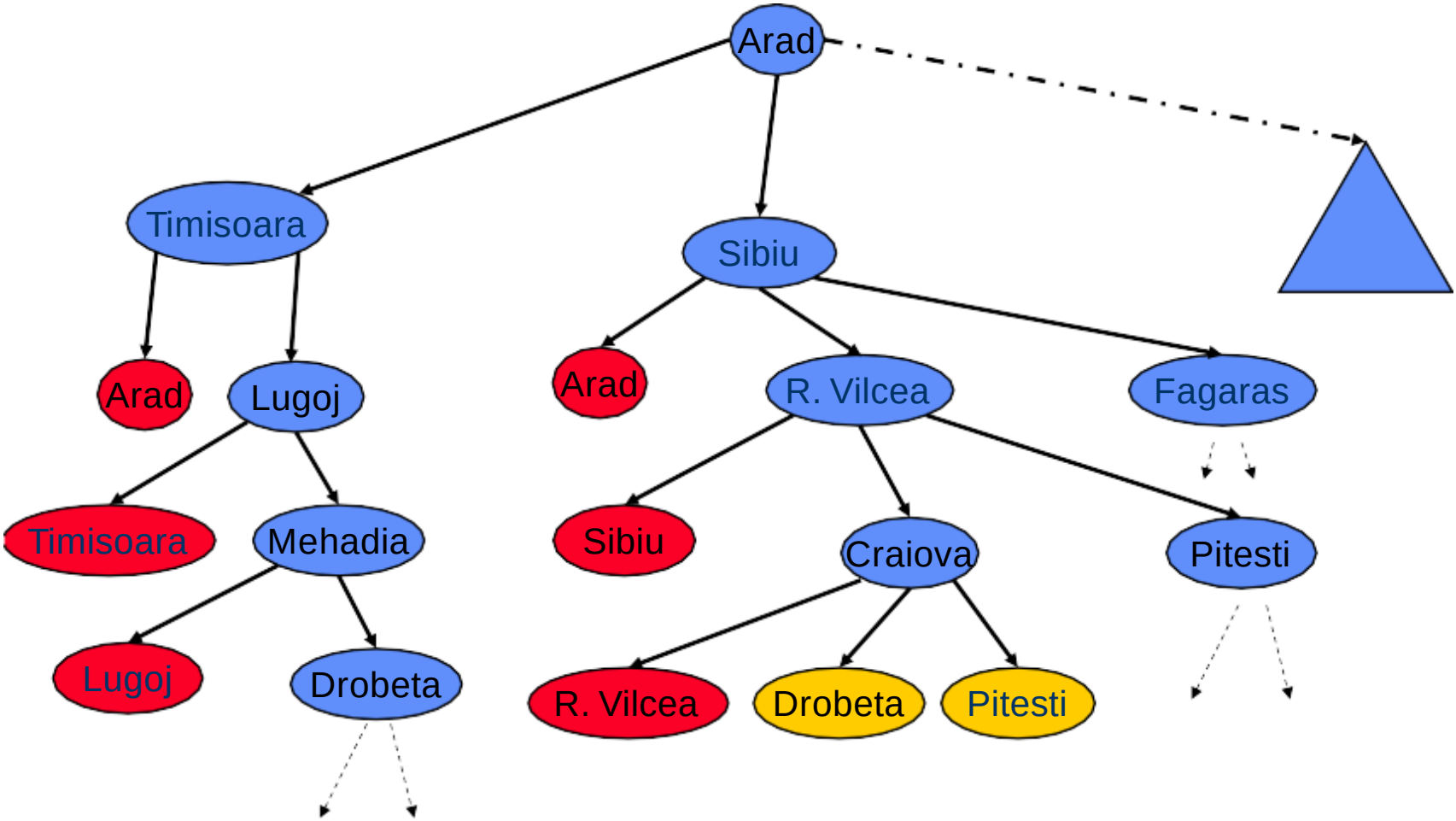
环检测中关于最优性的问题

对于一致代价搜索，使用环检测后仍能找到最优的解

- 一致代价搜索在第一次扩展到某个节点时，其实已经找到了到达这个节点的成本最低的路径
- 这意味着被环检测剔除的节点不可能出现一条更短/成本更低的路径

之后会看到，对于启发式搜索，这个性质不一定会成立

环检测示例（宽度优先搜索）



路径/环检测

路径检测：当扩展节点n来获得子节点c时，确保节点c不等于到达节点c的路径上的任何祖先节点

环检测：记录下在之前的搜索过程中扩展过的所有节点
当扩展节点 n_k 获得子节点c时，确保节点c不等于之前任何扩展过的节点

对于一致代价搜索，环检测可以保留一致代价搜索的最优性