搜索

- 基于搜索的问题求解
- 盲目搜索
- 启发式搜索

*Slides based on those of Sheila McIlraith

Why search

- 搜索被成功地应用于博弈程序中
- 许多其他的 AI 问题都可以成功地用搜索求解

搜索问题的形式化定义

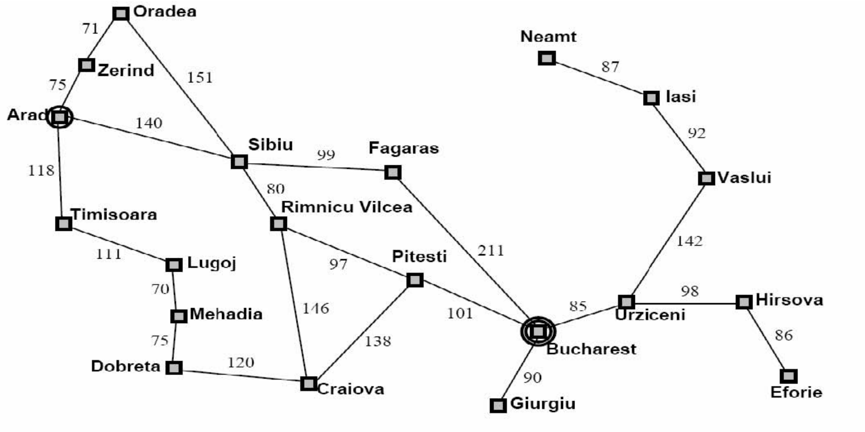
把一个问题形式化为一个搜索问题，我们需要以下成分：

1. 搜索的状态空间：状态空间需要对现实问题进行抽象
2. 使状态发生改变的动作集合：这里的动作是现实世界动作的抽象
3. 最好地表示当前状态的初始状态
4. 拟达到的目标或预期条件

问题的一个解是把初始状态转变成一个满足目标条件的状态的一个动作序列

例 1: 罗马利亚旅行

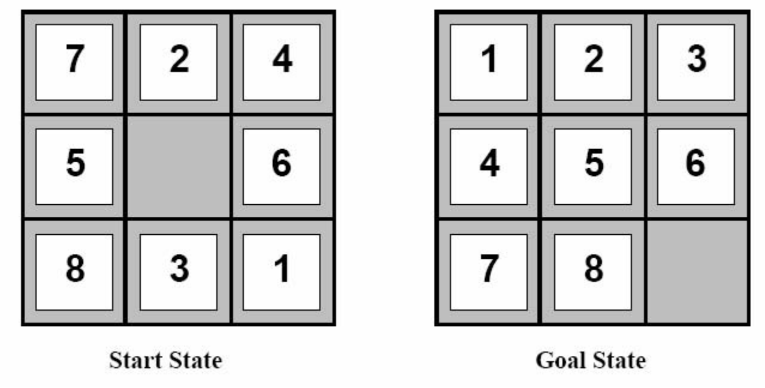
当前在阿拉德 (Arad), 需要到达布加勒斯特 (Bucharest)



例 1

- 状态: 不同的城市.
- 动作: 驾车到一个邻近的城市.
- 初始状态: 阿拉德
- 目标: 布加勒斯特
- 解: 从阿拉德到布加勒斯特的一条路线.

例 2: 8 数码问题



规则: 把一个数码滑动到空白处，也可以看作是移动空白

例 2

- 状态: 数码的不同布局. 多少个不同的状态?
- 动作: 把空白上下左右移动. 每个状态下每个动作都可执行吗?
- 初始状态，目标: 如上页所示
- 解: 把初始状态转换为目标状态的一系列移动

搜索算法

输入：

- 初始状态
- 后继函数 S(x)= 从状态 x 通过一个动作可以到达的状态集
- 目标测试函数：判断一个状态是否满足目标
- 单步代价函数 $C(x, a, y)$: 从状态 x 使用动作 a 移动到状态 y 的代价（$=\infty$ 如果 a 不能把 x 变为 y）

输出：

- 从初始状态到一满足目标的状态的一个状态系列

得到动作序列

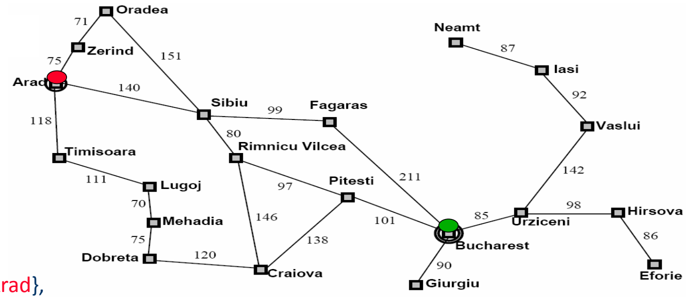
- 状态 x 的后继集可能由不同的动作产生，如 $x \to a \to y$, $x \to b \to z$
- 后继函数 S(x)= 从状态 x 通过一个动作可以到达的状态集
- 我们可以记录一个新状态是由哪个动作产生的，如
  - $S(x) = \{\langle y, a\rangle, \langle z, b\rangle\}$, y 通过 a，z 通过 b
  - $S(x) = \{\langle y, a\rangle, \langle y, b\rangle\}$, y 通过 a，y 也通过 b

树搜索算法

- 边界是我们还没有探索/扩展但想探索的
- 初始调用时令边界为初始状态的集合

```
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?)
```

例子: 罗马利亚旅行



{Arad},

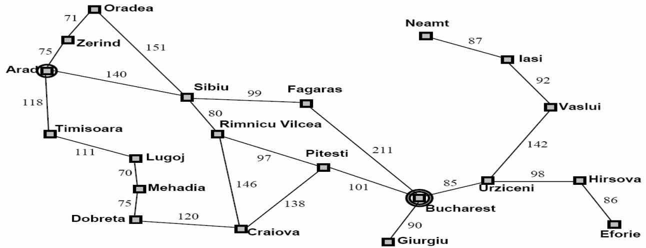
{**Z<A>, T<A>, S<A>**},

{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

另一个解



{Arad}

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

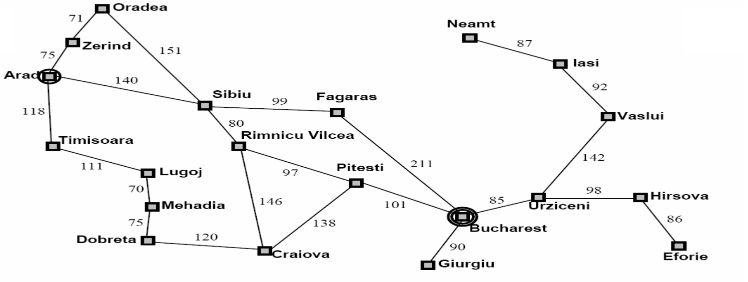
{Z<A>, T<A>, **A<S,A>, O<S,A>, F<S,A>, R<S,A>**}

{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

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>**}

.....

边界是路径的集合而不是状态的集合：环成为一个问题

选择规则

这个例子说明从边界选择状态的顺序对于搜索操作有重要影响

- 是否能找到解
- 找到的解的代价
- 搜索所需要的时间和空间

搜索的关键性质

- 完备性：如果有一个解，搜索是否总能找到解
- 最优性: 搜索是否总能找到最小代价解? (当动作有代价时)
- 时间复杂性: 所生成或扩展的节点的最大数量?
- 时间复杂性: 必须保留在内存中的节点的最大数量?

盲目搜索策略

- 这些策略采用固定的规则来选择下一个要扩展的状态。
- 不管要解决的搜索问题如何，规则都不会改变。
- 这些策略不考虑关于特定搜索问题的任何领域相关信息。

常用盲目搜索技术

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

通过排序来选择

- 我们将采用的一种简单一致的选择方法
  - 对边界上的元素排序.
  - 总是选择第一个元素.
- 任何选择规则都可以通过使用边界集的适当排序来实现.

宽度优先

- Place the successors of the current state at the end of the frontier.
- Example:
  - let the states be the positive integers {0,1,2,…}
  - let each state n have as successors n+1 and n+2
    - E.g. S(1) = {2, 3}; S(10) = {11, 12}
  - Start state 0
  - Goal state 5

Example

{0<>}

{1,2}

{2,2,3}

{2,3,3,4}

{3,3,4,3,4}

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

...

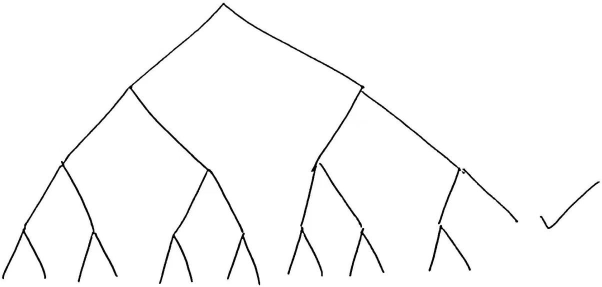
宽度优先性质

- 令 b 为任何状态的最大后继数量
- 令 d 为最优解中动作的数量.

完备性和最优性

- 所有较短的路径都在较长路径之前扩展
- 任何长度的路径只有有穷多个
- 最终我们必须考查长度为 d 的所有路径，从而找到最短解

时间和空间复杂性



- Time complexity: $1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- Space complexity: $b(b^d - 1) = O(b^{d+1})$

空间复杂性是一个真正的问题

- E.g., let b = 10, and say 1000 nodes can be expanded per second and each node requires 100 bytes of storage:

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

- Run out of space long before we run out of time in most applications.

深度优先

- 将当前状态的后继放在边界的前面
- 因此总是扩展边界中最深的节点

针对宽度优先搜索的例子

$\{0\}$
$\{1, 2\}$
$\{2, 3, 2\}$
$\{3, 4, 3, 2\}$
$\{4, 5, 4, 3, 2\}$
$\{5, 6, 5, 4, 3, 2\}$
...

深度优先的性质

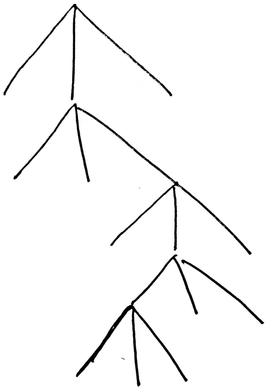
- 完备性:
  - 无穷状态空间: No
    - *e.g.*, $S(0) = \{1, 2\}$, $S(n) = \{n + 2, n + 4\}$ for $n > 0$, init state 0, goal is 6
  - 有穷状态空间但有无穷路径: No
    - *e.g.*, $S(0) = \{1, 2\}$, $S(1) = \{0\}$, init state 0, goal is 2
  - 有穷状态空间并不考虑有重复状态的路径? Yes
    - 这样的路径只有有穷多条
    - 如果有解，最终会找到一个解
- 最优性: No

时间复杂性

- $O(b^m)$ 这里 m 是状态空间中最长路径的长度 (可能探索搜索树中的每个分支)
- 很差，如果 m 比 d 大得多
- 但是如果有很多解，可能比宽度优先快得多 (运气好能很快找到解).

空间复杂性

- 深度优先回溯点 $=$ 当前路径上未探索节点的兄弟节点
- 一次只探索一条路径
- 边界只包含当前路径上最深的节点以及回溯点
- $O(bm)$, 线性空间!
- DFS 的显著优势



一致代价

- 边界上的路径按代价的增序排列
- 始终扩展代价最小的路径
- 如果每个动作的代价相同，则与广度优先一样

完备性和最优性

- 假设每条边的代价 $\geq \epsilon > 0$ (每条边的代价都为正数并且不能任意小)
- 所有代价较小的路径都在任何较昂贵的路径之前扩展
- 令 C^* 为最优解的代价
- 最优解的长度至多 C^*/ϵ
- 因而代价 $< C^*$ 的路径只有有穷多条
- 最终我们将考查最优解

时空复杂性

- 宽度优先的时空复杂性是 $O(b^{d+1})$，这里 d 是最优解的长度
- 用 C^*/ϵ 代替 d, 得到 $O(b^{C^*/\epsilon+1})$

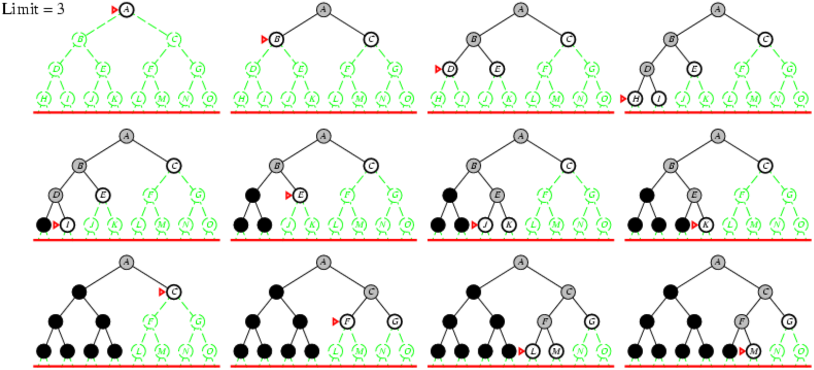
深度受限搜索

- 宽度优先在空间复杂度方面存在问题
- 深度优先可能选择了一条很长的或无限长的路径

深度受限搜索

- 执行 DFS，但只到预先指定的深度限制 L
- 现在无限长的路径不是问题
- 但只有当长度 $< L$ 的解存在时才会找到解

一个例子



深度受限性质

- Completeness: No (if an optimal solution has length $> L$)
- Optimality: No
- Time complexity: $O(b^L)$
- Space complexity: $O(bL)$

迭代加深搜索

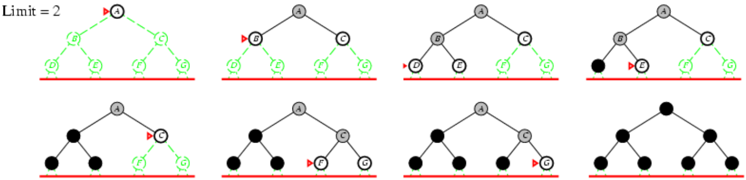
- 通过扩展深度受限搜索，解决了深度优先和广度优先的问题
- 从深度限制 $L = 0$ 开始，我们迭代增加深度限制，对每个深度限制执行深度受限搜索
- 如果找到解，或者深度受限搜索失败且没有因为深度限制而剪掉任何节点，则停止
- 如果没有节点被剪掉，则搜索考查了状态空间中的所有路径但没有找到解，因此无解。

An example

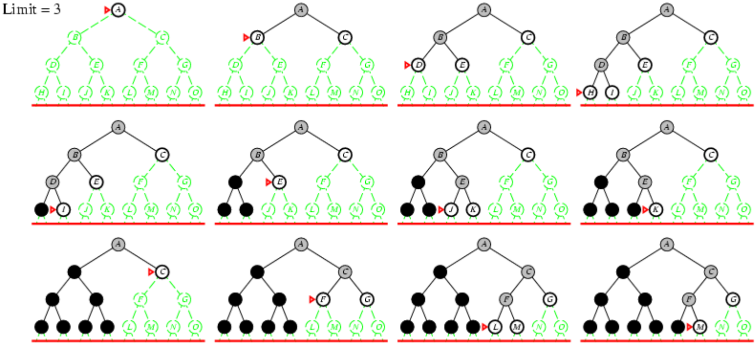
Limit = 0

An example

An example

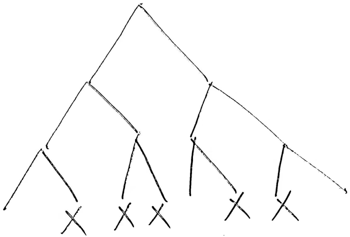


An example

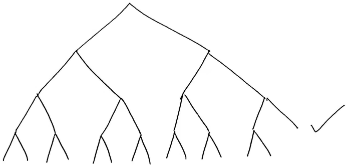


完备性和最优性

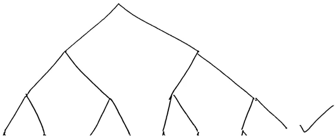
- 完备性：Yes
- 最优性：Yes 如果动作代价是一致的
- 如果动作代价不是一致的，代替深度限制，可以用一个代价限制
  - 只展开成本小于成本限制的路径
  - 在每次深度优先迭代，记录未展开路径的最小代价，在下一次迭代中以此为代价限制
  - 这可能会非常昂贵。有多少不同的路径代价，就需要多少次迭代



时空复杂性



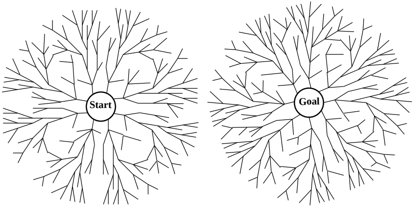
BFS



IDS

- $(d+1)b^0 + db + (d-1)b^2 + \ldots + b^d = O(b^d)$
- 而宽度优先的时间复杂性：
  $1 + b + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$
- IDS 可以比 BFS 更加高效：深度 d 的节点不被扩展，而 BFS 必须扩展所有深度 d 的节点直到扩展一个目标节点
- 空间复杂性：$O(bd)$，因为深度限制 $\leq d$

双向搜索



- 同时从初始状态向前搜索和从目标向后搜索，当两个搜索在中间相遇时停止
- 假设双向都使用 BFS
- 完备性：Yes
- 最优性：如果动作有一致代价
- 时空复杂性：$O(b^{d/2})$

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.

注. 表来自于 RN 教材。在 RN 教材中，对 BFS, 当一个节点被生成时就做目标测试, 因而时空复杂性都是 $O(b^d)$ 而非 $O(b^{d+1})$.

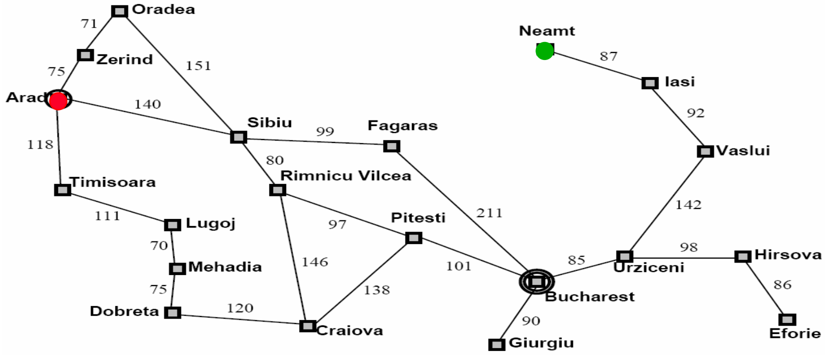
路径检测

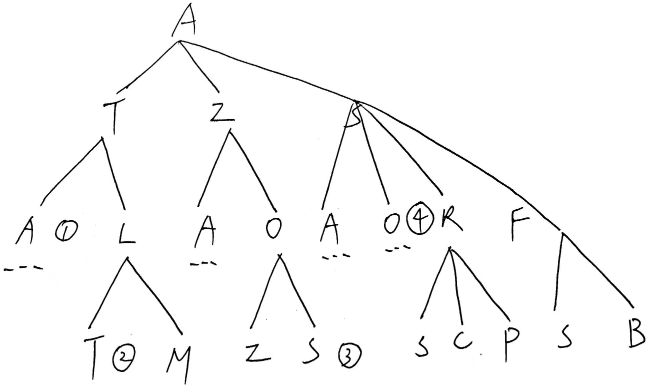
- 注意边界上存贮的是路径
- 如果 $\langle n_1, \ldots, n_k \rangle$ 是一条到 n_k 的路径, 并且我们扩展 n_k 得到孩子 c, 则 $\langle n_1, \ldots, n_k, c \rangle$ 是到 c 的路径
- 路径检测确保 c 不同于路径上它的任何祖先节点
- 因而路径是单独检查的!

环检测/ 多路径检测

- 记录在搜索过程中扩展过的所有状态
- 当我们扩展 n_k 得到孩子 c 时，确保 c 不同于任何先前扩展的状态
- 为什么我们不能在深度优先搜索中使用这种技术?
- 空间复杂度高，仅对广度优先搜索有用

Example: Arad to Neamt





- If path checking, nodes 1 and 2 are not generated
- If cycle checking, node 3 is not generated since it is expanded before; but if only path checking, node 3 is generated
- If cycle checking, node 4 is generated, because it is only generated before, not expanded before

环检测保最优性吗?

- 对于一致代价搜索，我们仍然找到最优解
- 一致代价第一次扩展一个状态时，它已经找到了到达该状态的最小代价路径。
- 这意味着被环检测剪掉的节点不会有更好的路径。
- 例如，在上一页，当我们扩展第一个 O 生成节点 3 时, S 已被扩展, 因而 $c(A \rightarrow S) \leq C(A \rightarrow Z \rightarrow O)$. 因而节点 3 可以被安全地剪掉.

路径 / 环检测总结

- 路径检测: 当我们扩展 n 得到孩子 c 时, 确保 c 不同于路径上它的任何祖先节点
- 环检测：记录在搜索过程中扩展过的所有状态，当我们扩展 n 得到孩子 c 时, 确保 c 不同于任何先前扩展的状态
- 对于一致代价搜索，环检测保持最优性

Exercise

Running breadth-first with cycle-checking to get from Sibiu to Bucharest