目 录

盲目搜索策略

- 无信息搜索又名**盲目搜索**：
  - 在搜索时，只有问题定义信息可用。
  - 在搜索时，当有策略可以确定一个非目标状态比另一种更好的搜索，称为**有信息的搜索**。

- 盲目搜索策略仅利用了问题定义中的信息，所有的搜索策略是由节点扩展的顺序加以区分。

  - 宽度优先搜索
  - 代价一致搜索
  - 深度优先搜索
  - 深度有限搜索
  - 迭代深度搜索
  - 双向搜索

搜索策略评价

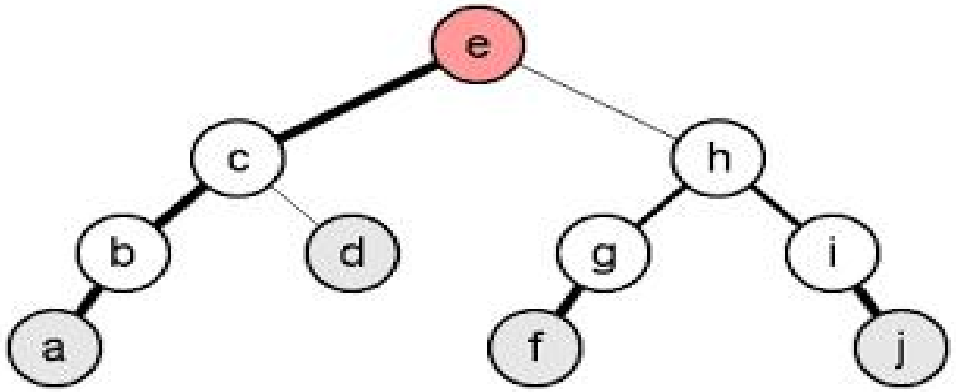
- 搜索**策略**指节点扩展顺序的选择。
- 搜索策略的**性能**由下面四个方面来**评估**:
  - **完备性:** 如果问题的解存在时它总能找到解。
  - **时间复杂性:** 产生的节点个数。
  - **空间复杂性:** 搜索过程中内存中的最大节点数。
  - **最优性:** 它总能找到一个代价最小的解。

搜索策略评价

- **问题难度**由时间和空间复杂度的定义来度量的：

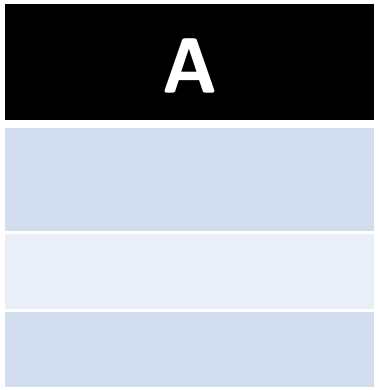
✓时间和空间复杂度根据下面三个量来表达：

- *b:* 搜索树的最大分支数
- *d:* 最小代价解所在的深度
- *m*: 状态空间的最大深度(可能是∞)

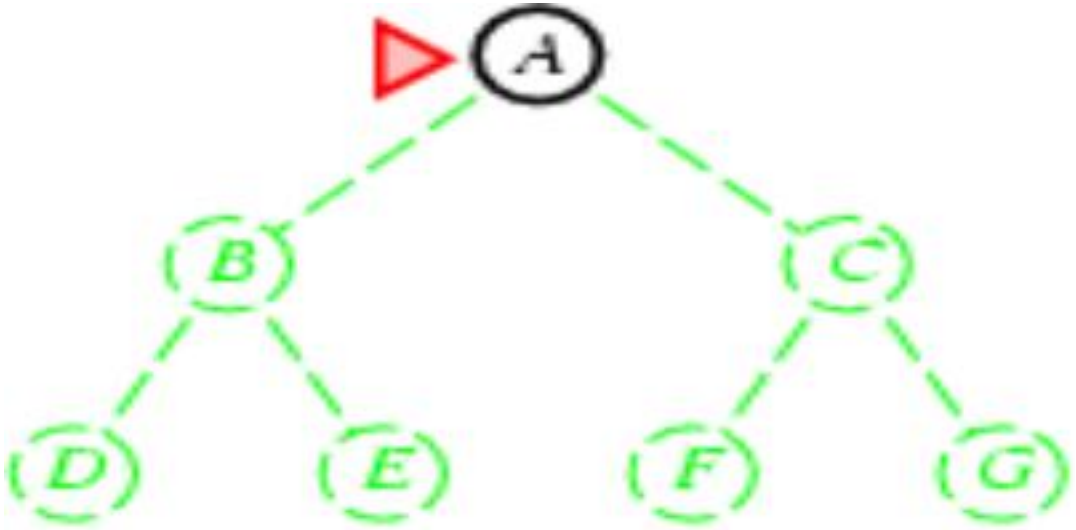


4.1 宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - Fringe表采用先进先出队列（ FIFO queue），即新的后续节点总是放在队列的末尾

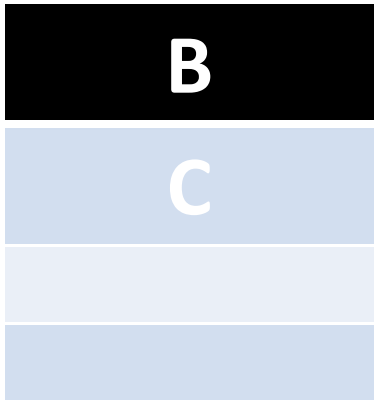


Fringe表

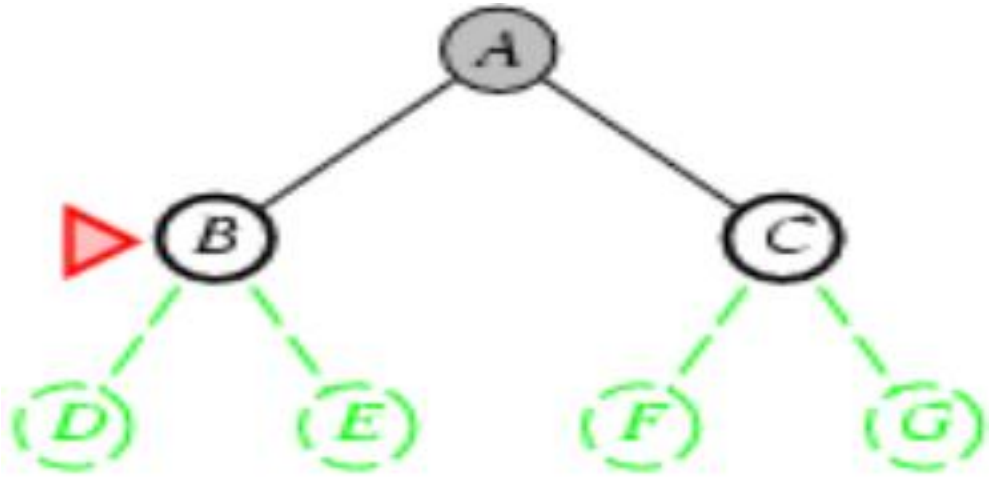


宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - Fringe表采用先进先出队列（ FIFO queue），即新的后续节点总是放在队列的末尾

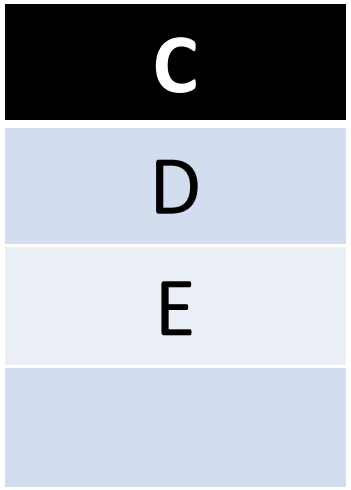


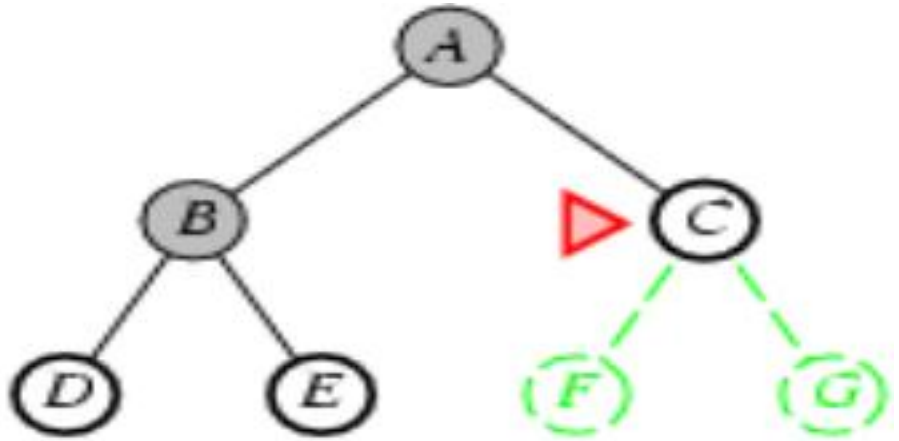
Fringe表



宽度优先搜索

- 优先扩展最浅层的未扩展节点
- 实现方法:
  - Fringe表采用先进先出队列（ FIFO queue），即新的后续节点总是放在队列的末尾

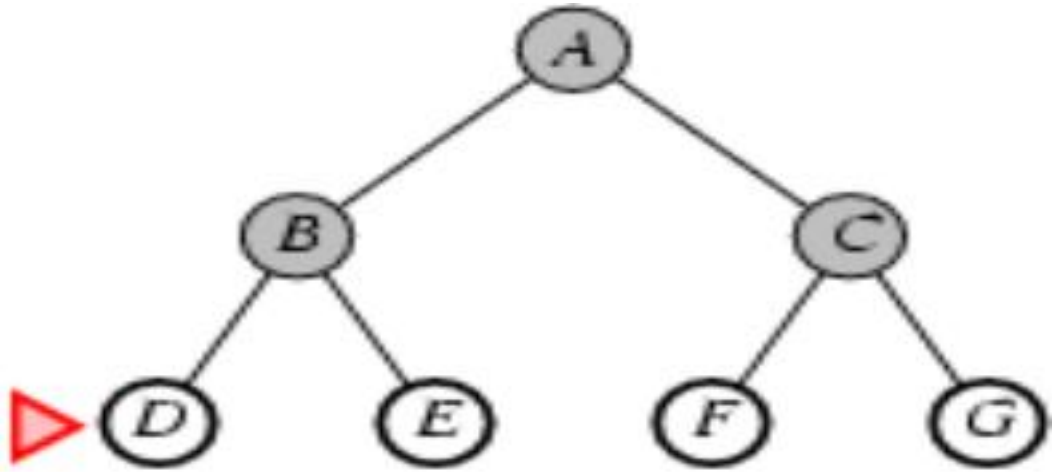




宽度优先搜索

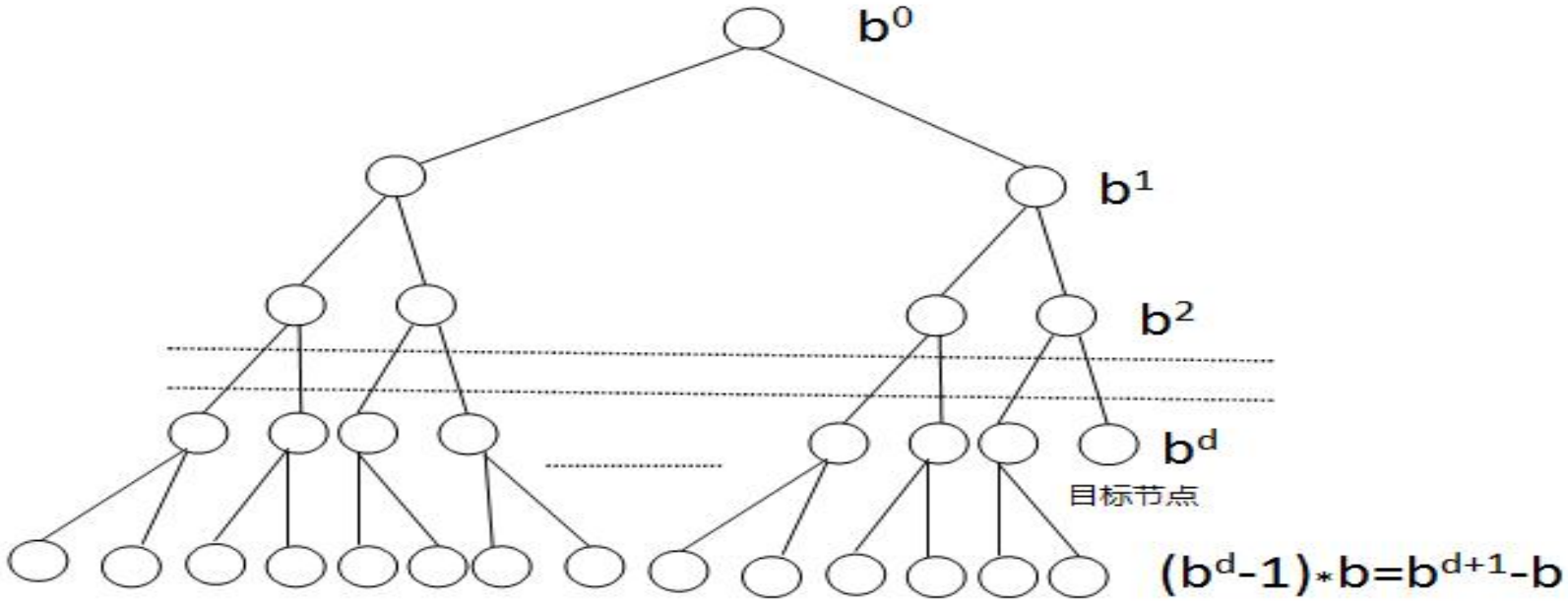
- 优先扩展最浅层的未扩展节点
- 实现方法:
  - Fringe表采用先进先出队列（ FIFO queue），即新的后续节点总是放在队列的末尾

D
E
F
G



宽度优先搜索的性能指标

- 时间? $1+b+b^2+b^3+\ldots+b^{d-1}+(b^d-1)*b=O(b^{d+1})$



宽度优先搜索的性能指标

- 完备性? Yes (只要 *b* 是有限的)
- 时间? $1+b+b^2+b^3+\ldots+b^d+b(b^d-1) = O(b^{d+1})$
- 空间? $O(b^{d+1})$
- 最优性? Yes (只要单步代价是一样的)

Assume: branch factor b=10， 1 million nodes/s， 1 Kbytes/node：

DEPTH2	NODES	TIME	MEMORY
2	111	0.11 milliseconds	107 kilobyte
4	11,111	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabytes
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabytes
14	10^{14}	3.5 years	1 exabyte
16	10^{16}	350 years	10 exabyte

- 空间是一个比时间更严重的问题.
- 指数复杂性的搜索问题不能通过无信息搜索的方法求解。（除了最小的实例）

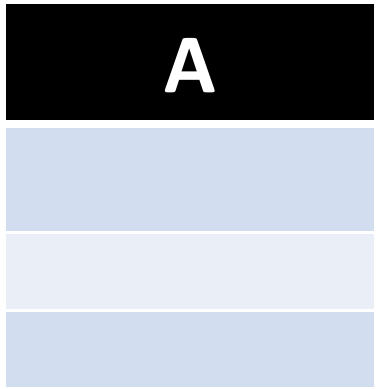
4.2一致代价搜索

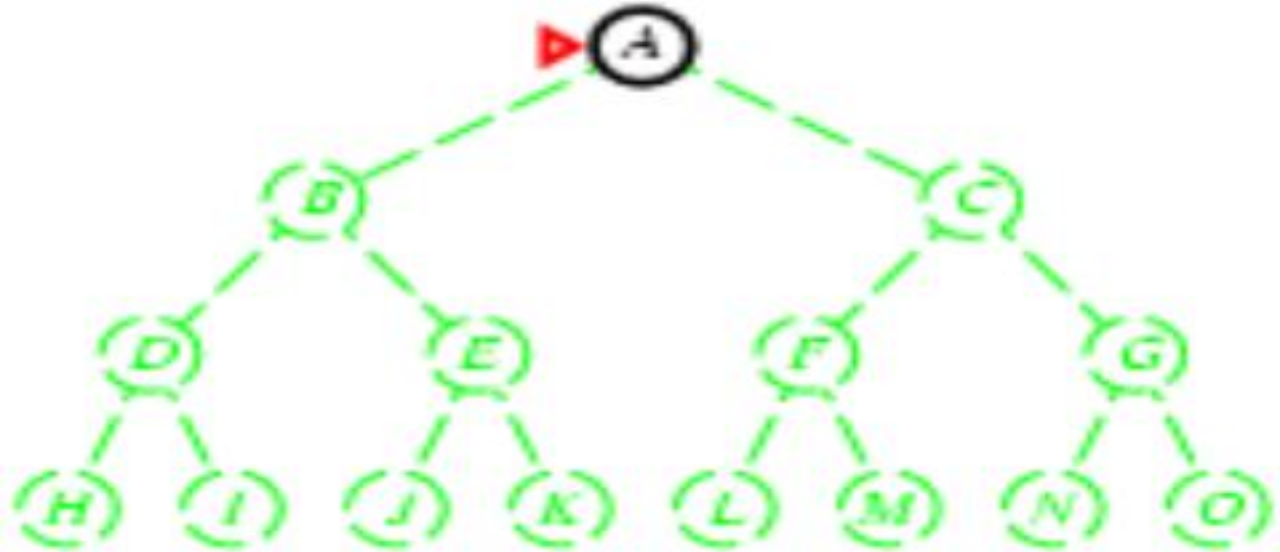
- 优先扩展具有**最小代价**的未扩展节点
- 实现：　　*fringe* 是根据路径代价排序的队列
- 在单步代价相等时与宽度优先搜索一样
- 完备性？　Yes，只要单步代价不是无穷小
  时间？　　代价小于最优解的节点个数，$O(b^{ceiling(C^*/\ \varepsilon)})$
- 空间？　　代价小于最优解的节点个数，$O(b^{ceiling(C^*/\ \varepsilon)})$
- 最优性？　Yes　－　节点是根据代价排序扩展的

注：　C^* 最优解的代价
　　ε 是至少每个动作的代价
　　ceiling取上整

4.3深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

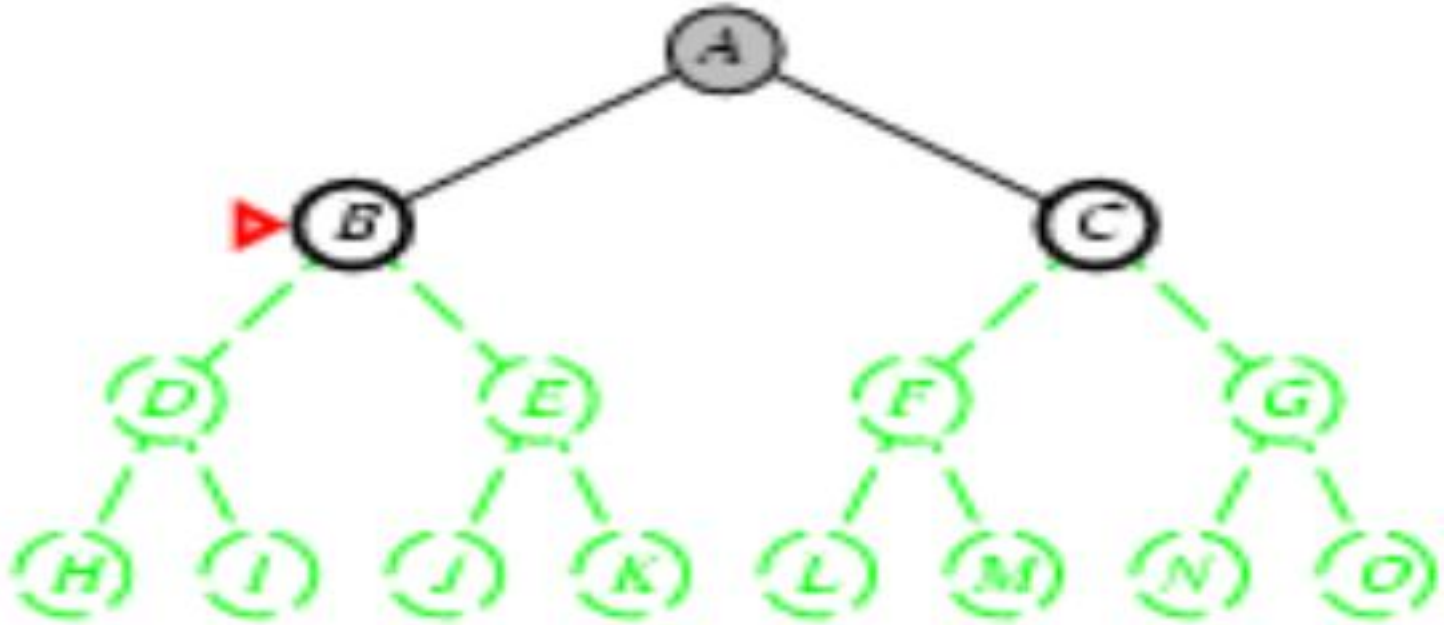




深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

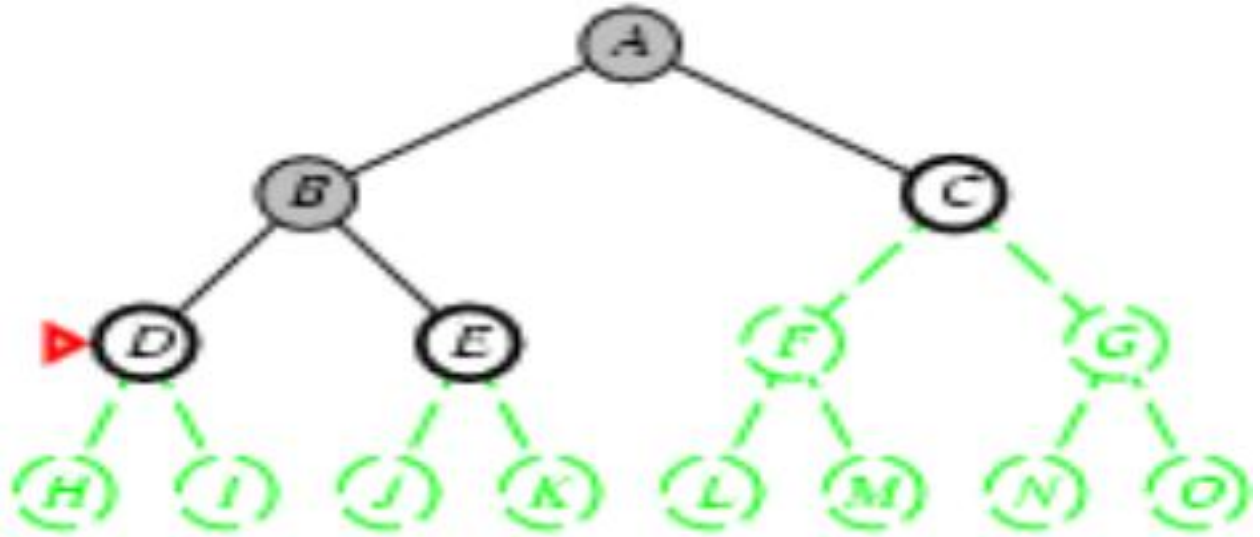
B
C



深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

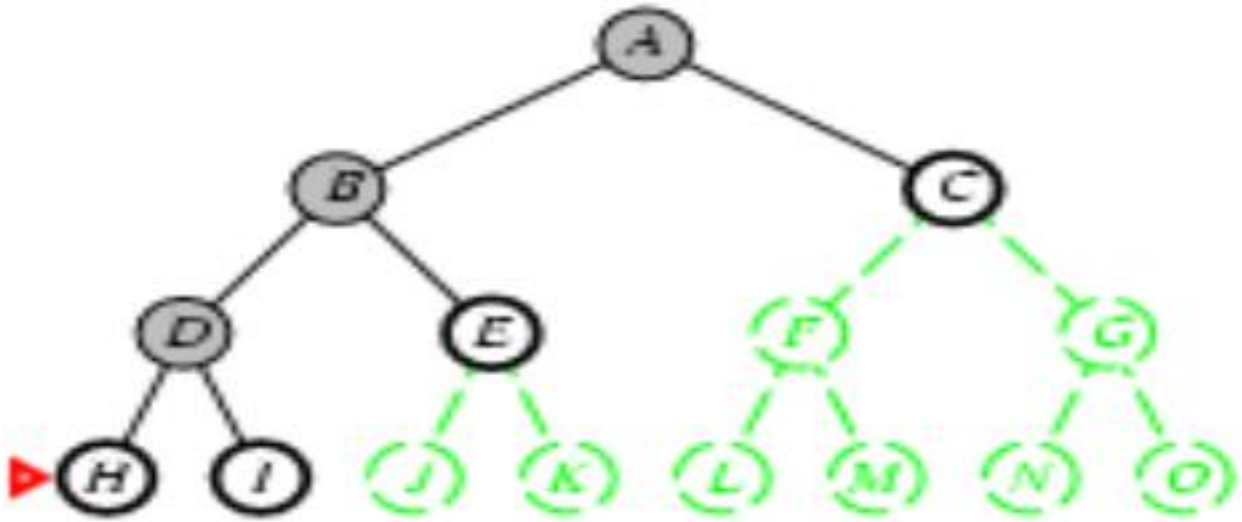
D
E
C



深度优先搜索

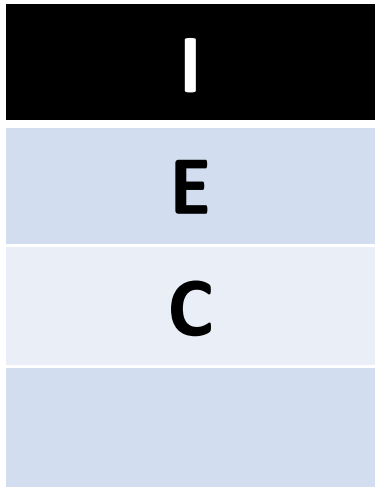
- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

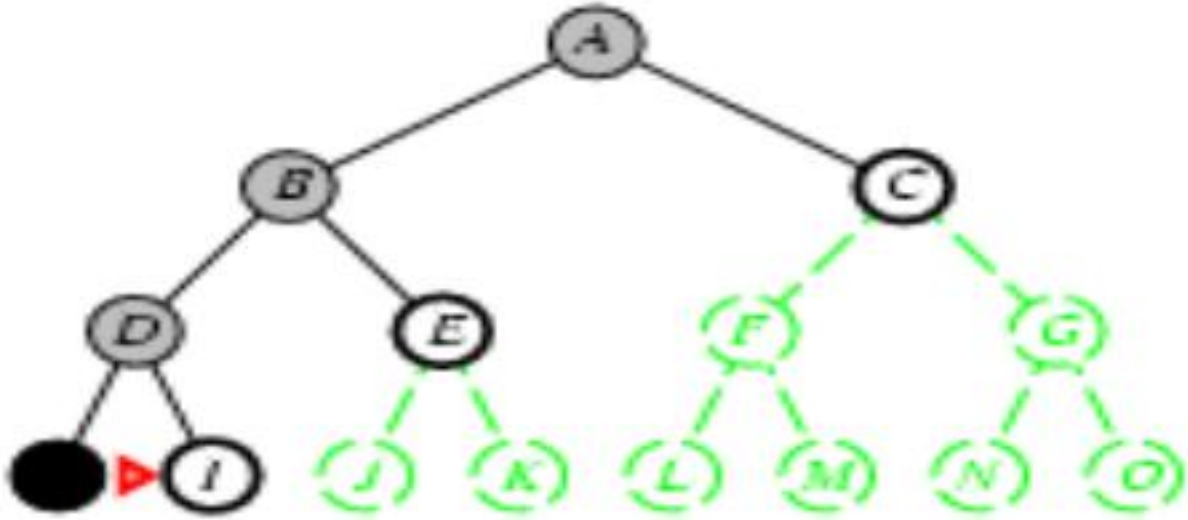
H
I
E
C



深度优先搜索

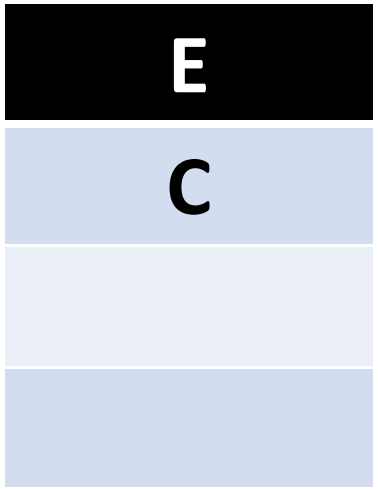
- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

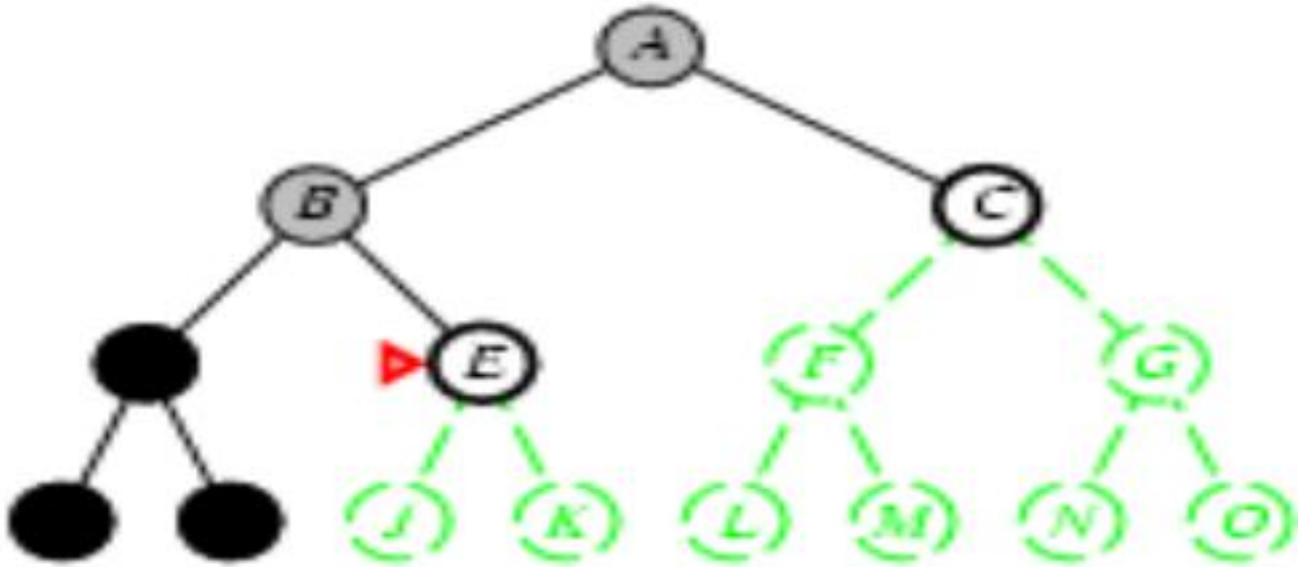




深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

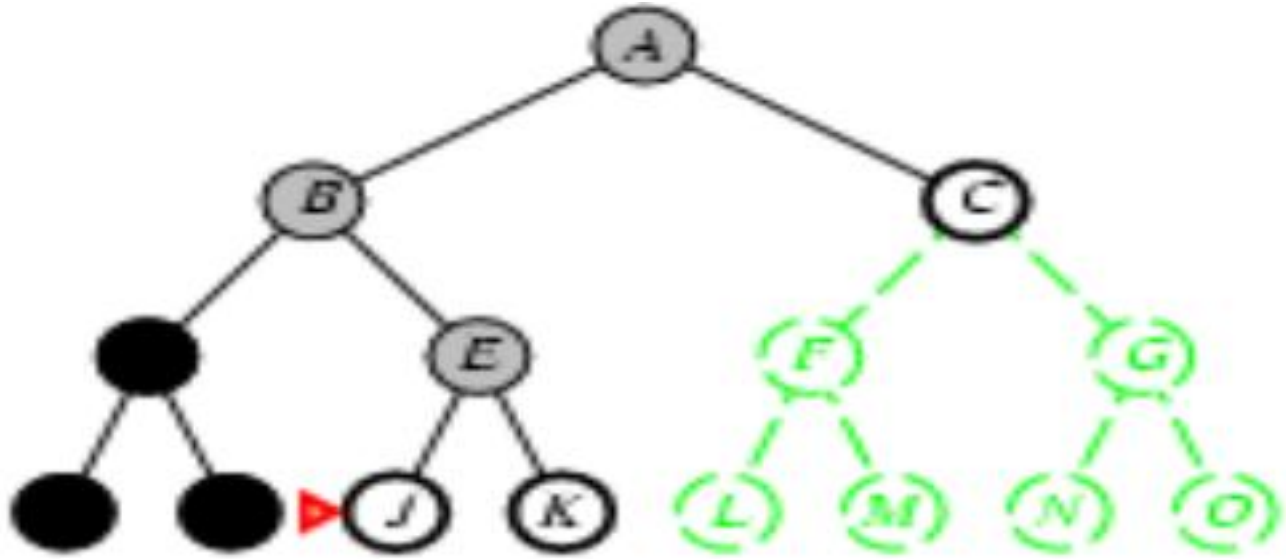




深度优先搜索

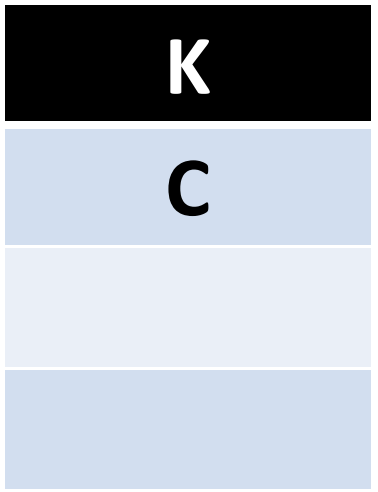
- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

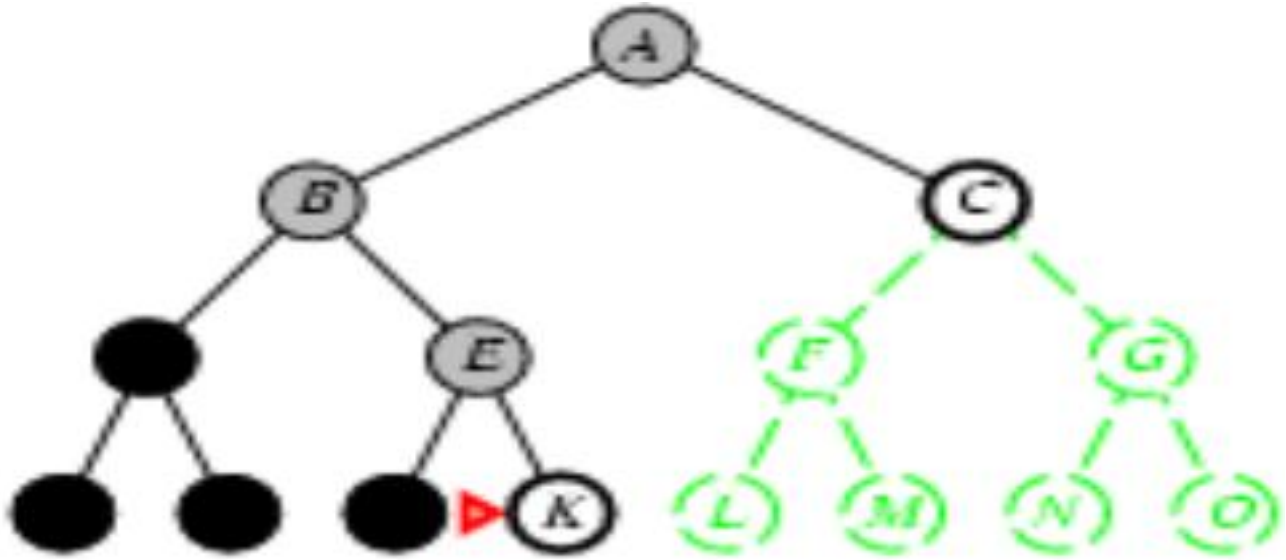
J
K
C



深度优先搜索

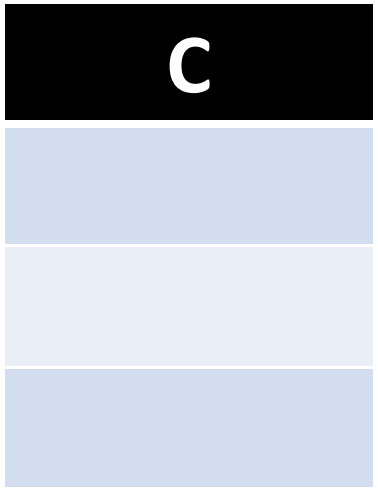
- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

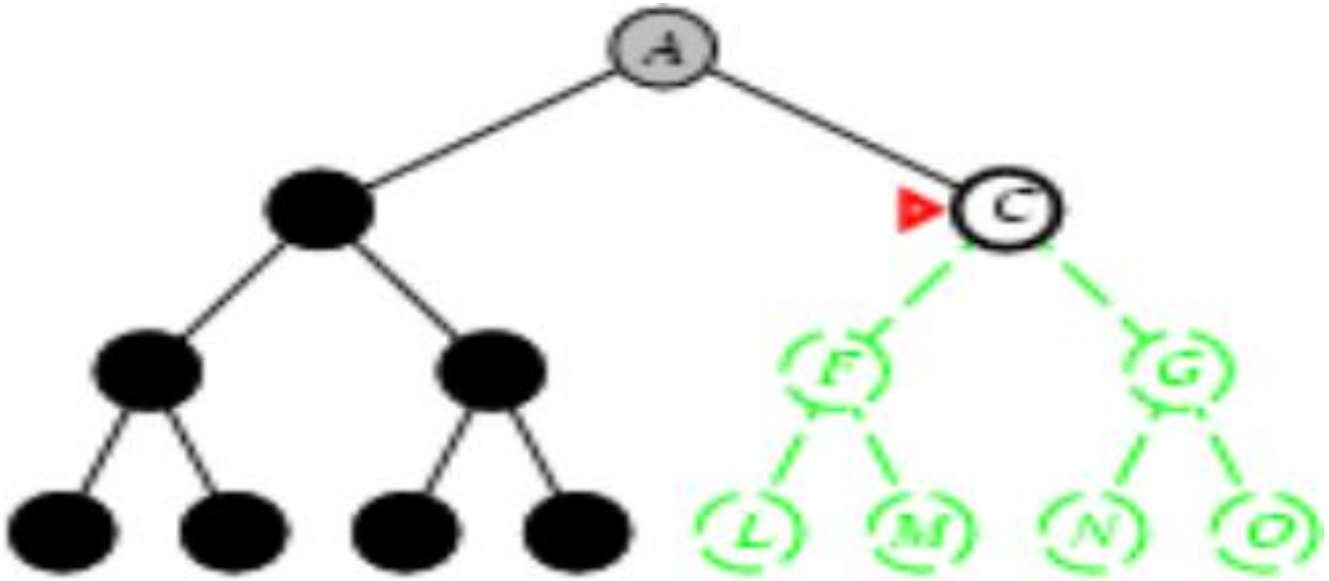




深度优先搜索

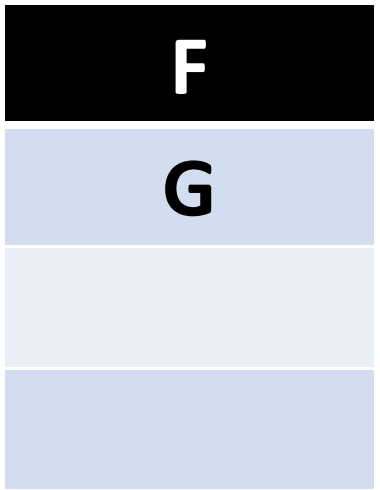
- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

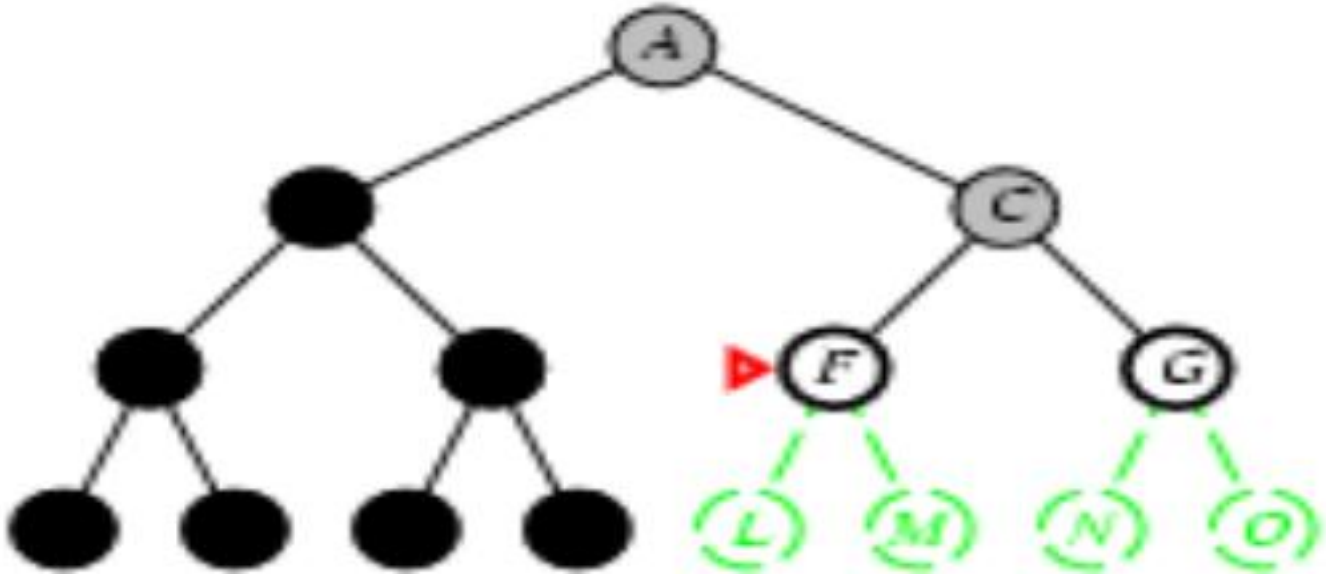




深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

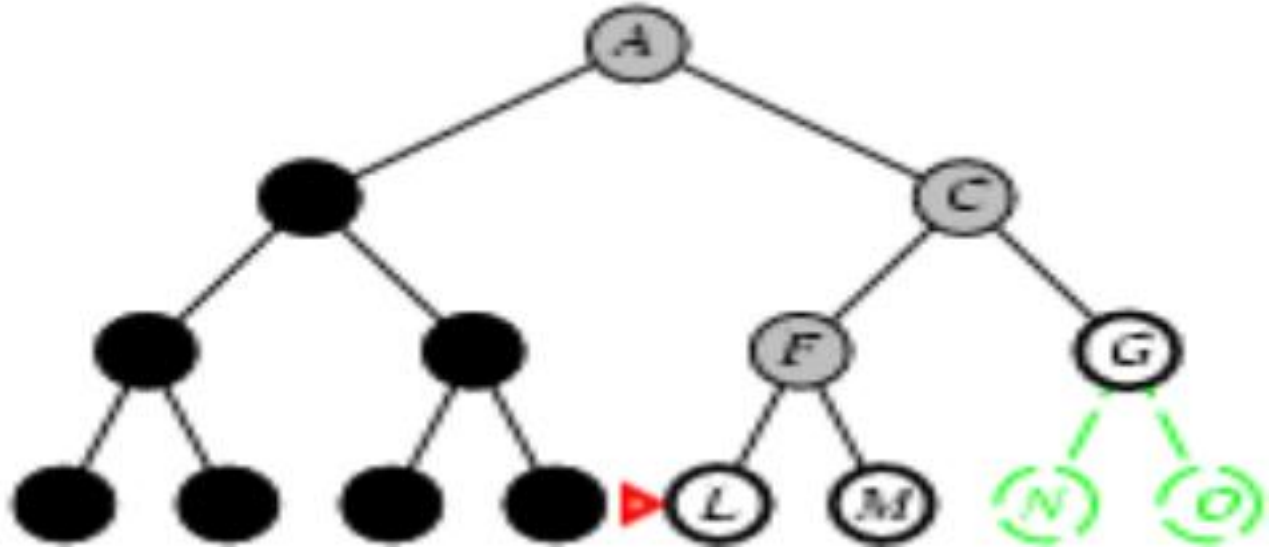




深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

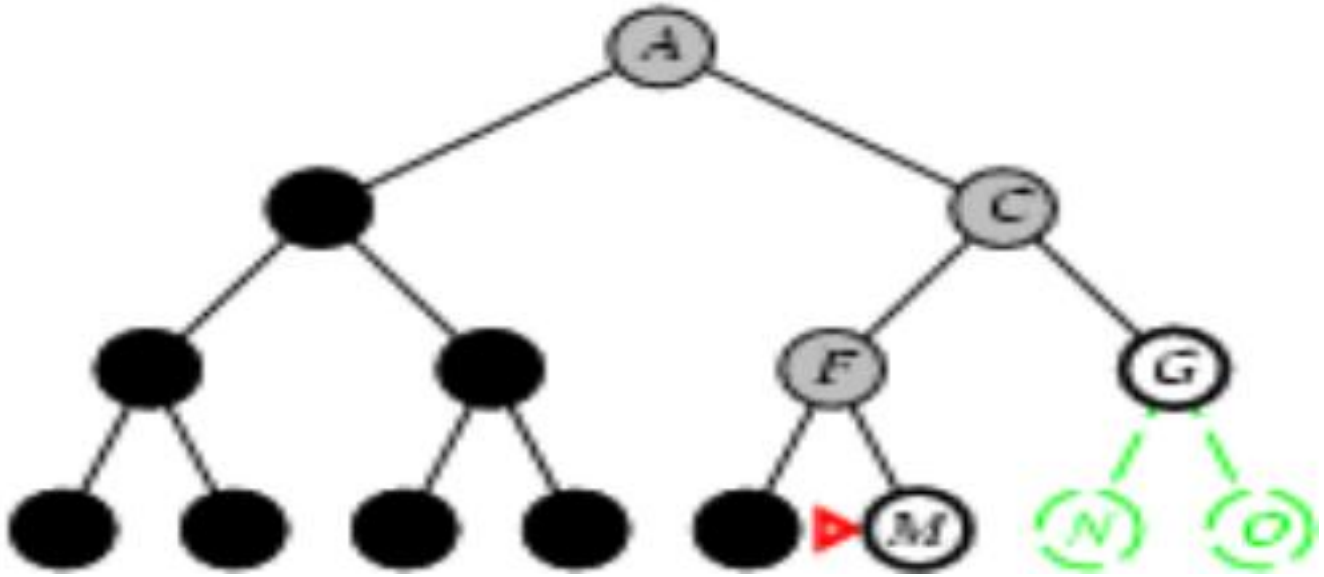
L
M
G



深度优先搜索

- 扩展最深层的未扩展节点
  实现:*fringe* = 后进先出队列（LIFO queue）

M
G



深度优先搜索的性能指标

- 完备性? No: 在无限状态空间中不能保证找到解
- 时间? $O(b^m)$
- 空间? $O(bm)$, i.e., 线性空间!
- 最优性? No

4.4 深度优先搜索改进

4.4.1有深度限制的深度优先搜索

- 递归实现:

```
function DEPTH-LIMITED-SEARCH( problem, limit) returns soln/fail/cutoff
  RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
  cutoff-occurred? ← false
  if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
  else if DEPTH[node] = limit then return cutoff
  else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff-occurred? ← true
    else if result ≠ failure then return result
  if cutoff-occurred? then return cutoff else return failure
```

深度有限搜索的性质

- 完备性? No
- 时间? $O(b^l)$
- 空间? $O(bl)$, i.e., 线性空间!
- 最优性? No

- 空间是一个比时间更严重的问题.
- 指数复杂性的搜索问题不能通过无信息搜索的方法求解。（除了最小的实例）

4.4.2迭代深入搜索

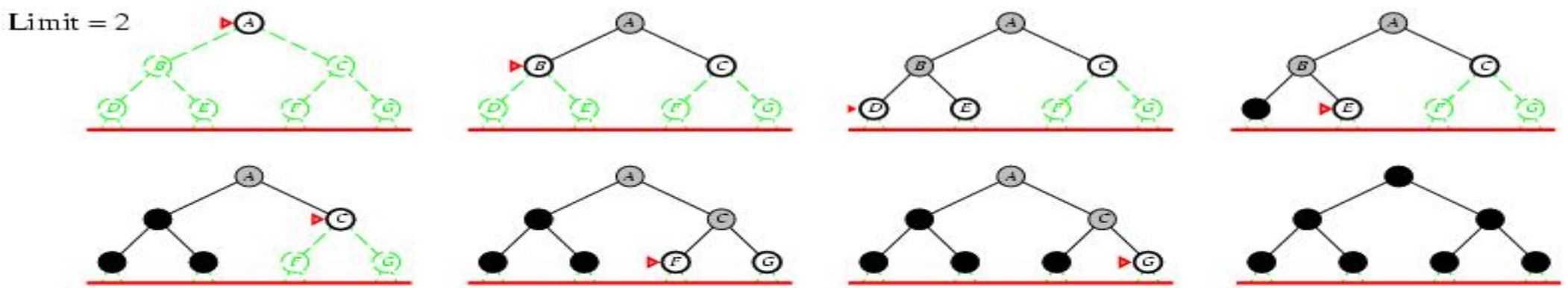
```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

迭代深入搜索 *l* =0

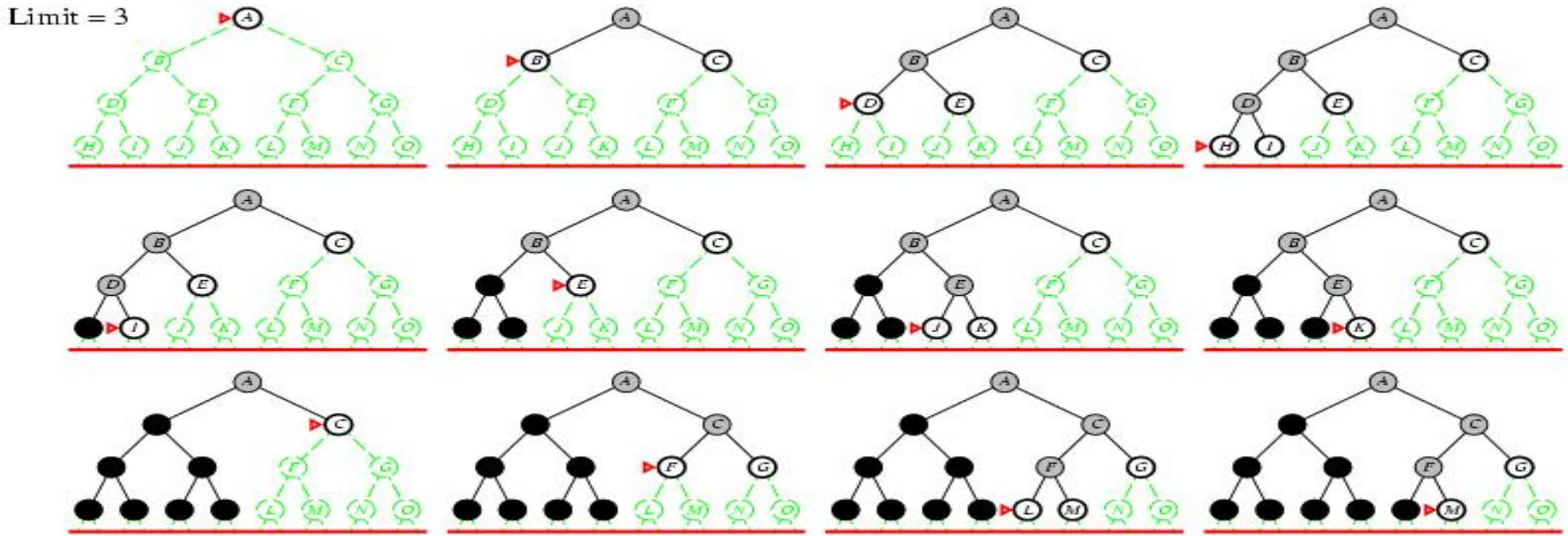
Limit = 0

迭代深入搜索*l* =1

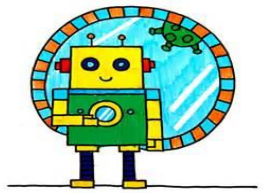
迭代深入搜索*l* =2



迭代深入搜索*l* =3



迭代深入搜索性能分析



- 深度有限搜索（Deep limited search, DLS）搜索到d层时产生的节点数：

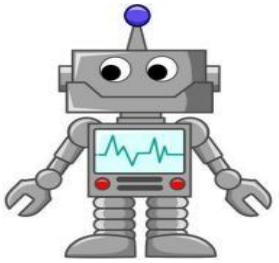
$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

- 迭代深入搜索（iterative deepening search ， IDS）搜索到d层时产生的节点数：

$$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

注： 宽度优先搜索 $1+b+b^2+b^3+\ldots+b^d + (b^d-1)*b = O(b^{d+1})$

迭代深入搜索性能分析



- **对于分支数$b = 10$, 深度$d = 5$的问题**

深度有限：$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$

$N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$

迭代深度有限：$N_{IDS} = (d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$

- $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$
- 超出比率 = (123,456 - 111,111)/111,111 = 11%

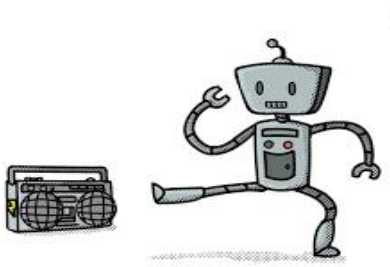
而宽度优先：$1+b+b^2+b^3+\ldots+b^d + (b^d-1)*b = O(b^{d+1})$

N= 1 + 10 + 100 + 1,000 + 10,000 + 100,000 +10(100000-1)=1,111,101

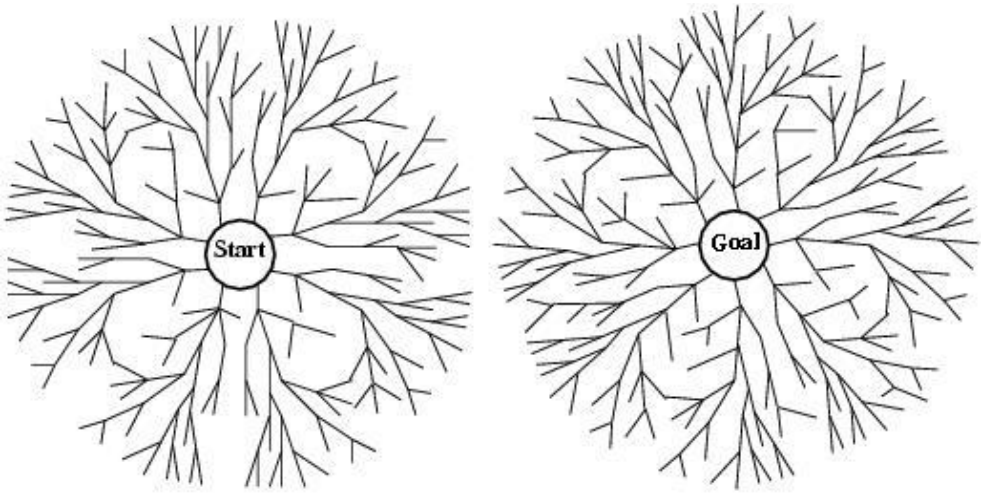
超出比达：88.9%

迭代深入搜索的性质

- 完备性? **Yes**
- 时间? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- 空间? $O(bd)$
- 最优性? **Yes,**只要单步代价相等



4.5 双向搜索



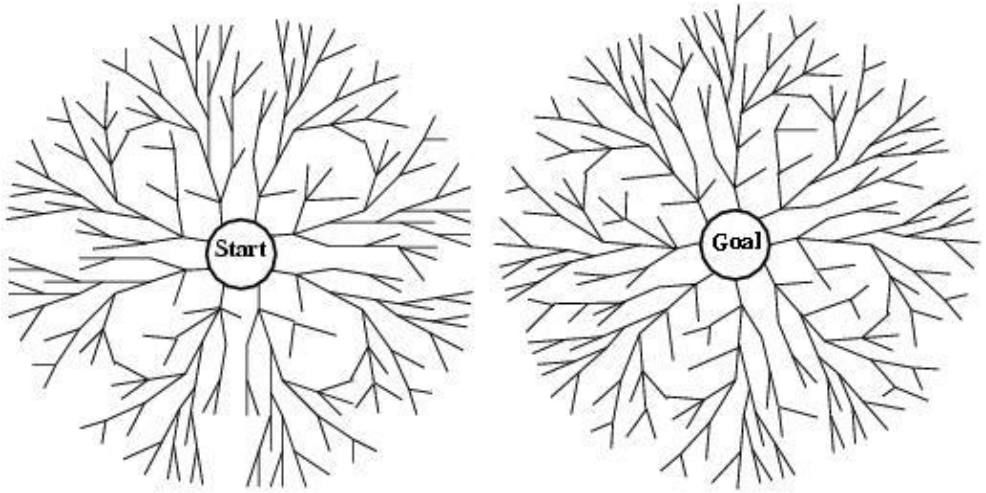
- 从初始状态和目标状态同时出发：
  - 原理： $b^{d/2} + b^{d/2} \leq b^d$
- 检查当前节点是否是其他fringe表的节点。
- 空间复杂度仍然是最大的问题。
- 如果双向都采用宽度优先则算法是完备的和最优性的.

比较 b=10 以及 d=6 的问题扩展节点数：

$$N(\mathrm{BiD}) = 2 * (1 + 10 + 100 + 1000) = 2222$$

$$N(BFS) = 1 + 10 + 100 + 1000 + 10000 + 100000 + 1000000 = 1111111$$

如何反向搜索?



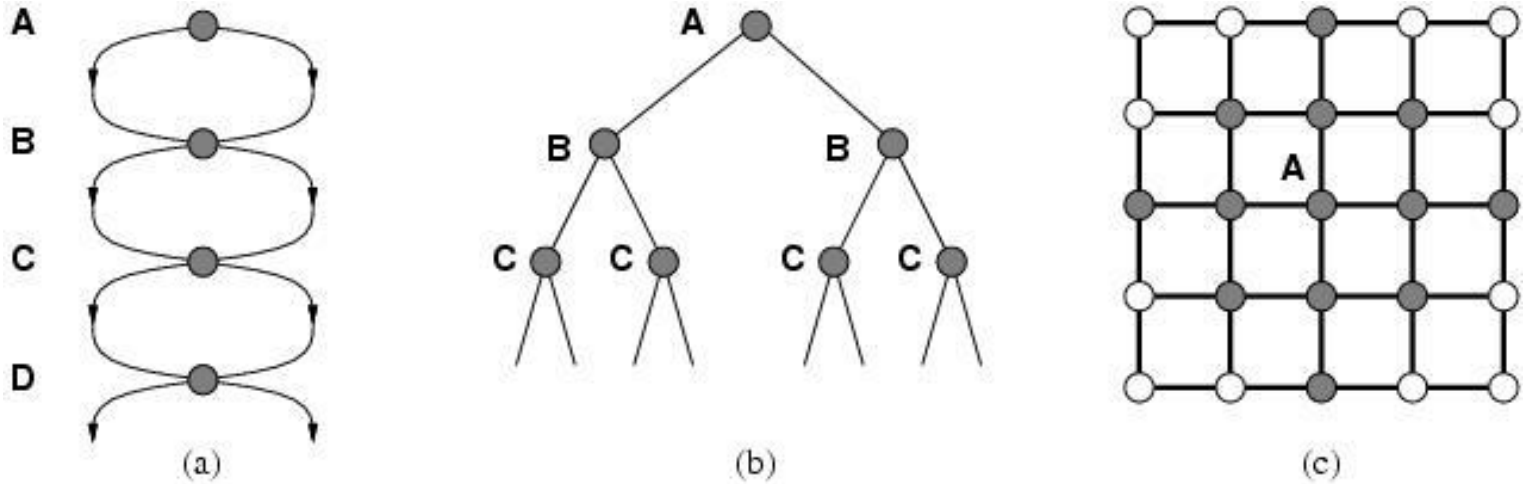
- 每个节点的前状态应是有效的可计算。
- 行动是容易可逆的。

4.6 无信息搜索算法性能评价小结

评价	宽度优先	代价一致	Depth-First	Depth-limited	Iterative deepening	Bidirectional search
完备性	YES*	YES*	NO	YES, if $l \geq d$	YES	YES*
时间复杂度	b^{d+1}	$b^{C^*/\varepsilon}$	b^m	b^l	b^d	$b^{d/2}$
空间复杂度	b^{d+1}	$b^{C^*/\varepsilon}$	bm	bl	bd	$b^{d/2}$
最优性	YES*	YES*	NO	NO	YES	YES

重复状态

- 如果不能检测重复的状态，会导致把一个线性空间问题变成指数级的问题（不可解）。



实现：图搜索算法

```
//explored 存储所有已扩展的节点
function GRAPH-SEARCH( problem, fringe) return a solution or failure
    explored ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
    if EMPTY?(fringe) then return failure
       node ← REMOVE-FIRST(fringe)
    if GOAL-TEST[problem] applied to STATE[node] succeeds
         then return SOLUTION(node)
    if STATE[node] is not in explored then
        add STATE[node] to explored
         fringe ← INSERT-ALL(EXPAND(node, problem), fringe)
```

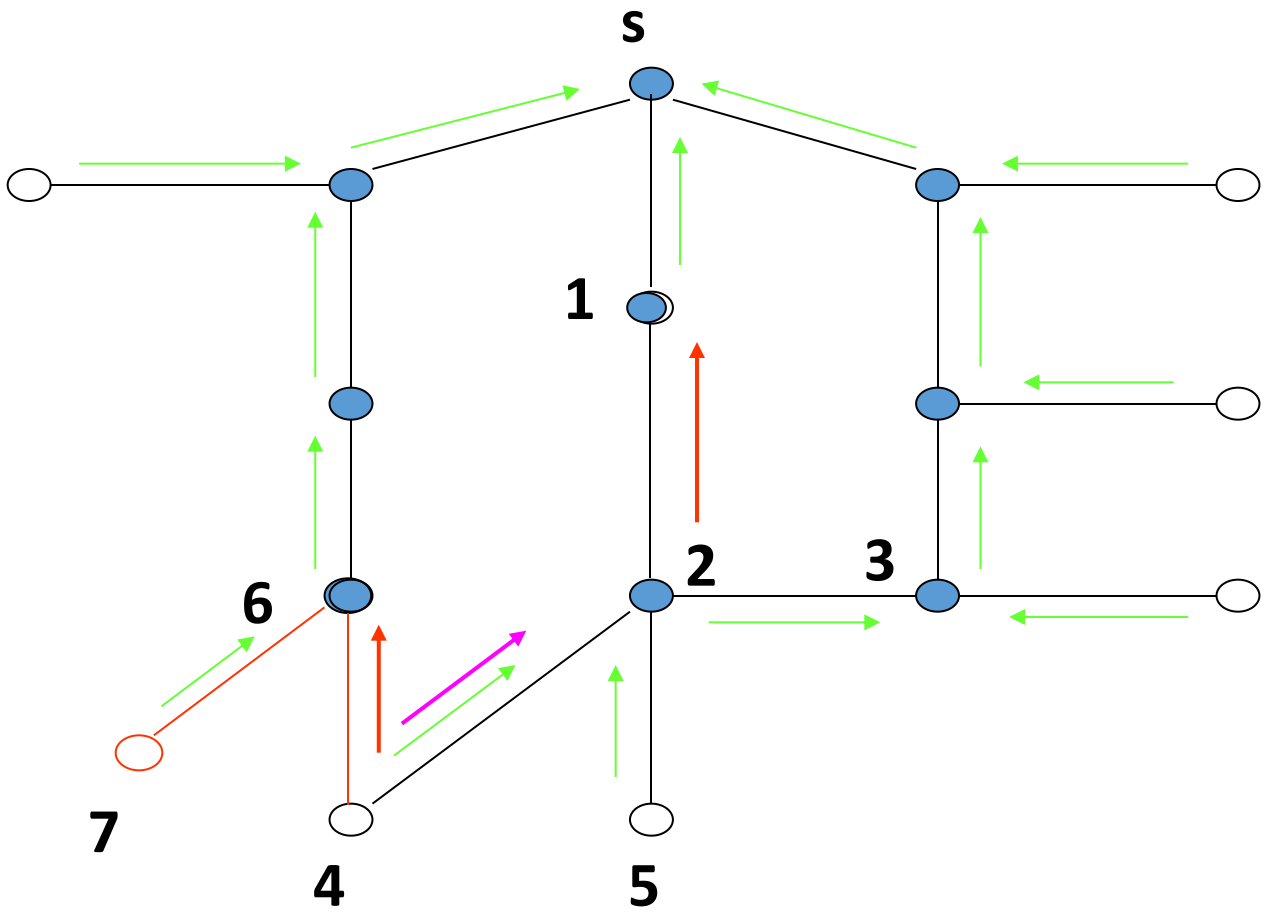
图搜索

- **完备性：**是的
- **最优性:**
  - 图搜索截断了新路径，这可能导致**局部**最优解。
  - 当单步代价相同宽度搜索或代价一致的搜索时是最优的。
- **时间和空间复杂度：**
  - 与状态空间的大小成比例的（可能远远小于$O(b^d)$）
  - 深度和迭代深度搜索算法不再是线性空间。（因为，需要保存**explored**表）。

图搜索例

fringe(1,4,7,5,…)

explored (s,2,3, 6,…)



cost(4)=5 cost(2)=4

Expand 6，⇒7，4

cost(4)=4

Modify 4 pointer， point to 6

Expand 1， ⇒ 2

cost(2)=2

Modify 2 pointer， Point to 1

cost(4)=3

Modify 4 pointer， Point to 2

无信息搜索思考题：

- 传教士和野人问题M-C问题（**Missionaries & Cannibals Problem**）

✓ **已知**：传教士人数M=3，野人人数C=3，一条船一次可以装载不超过2人K<=2。

✓ **条件**：任何情况下，如果传教士人数少于野人人数则有危险。

✓ **问题**：传教士为了安全起见，应如何规划摆渡方案，使得任何时刻，

河两岸以及船上的野人数目总是不超过传教士的数目。

即求解传教士和野人从左岸全部摆渡到右岸的过程中，任何时刻满足：

M(传教士数)≥C(野人数)和M+C≤k的摆渡方案。

✓**要求**：（1）形式化该问题，并计算状态空间大小；

（2）应用无信息搜索算法求解；（*考虑重复状态？*）

（3）这个问题状态空间很简单，你认为是什么导致人们求解它和困难？

问题表示

- **问题形式化：**

  用一个**三元组(m，c，b)**来表示河岸上的状态，其中m、c分别代表某一岸上传教士与野人的数目，b=1表示船在这一岸，b=0则表示船不在。

✓**条件**是: 两岸上M≥C, 船上M+C≤2。

说明：由于传教士与野人的总数目是一常数，所以只要表示出河的某一岸上的情况就可以了，为方便起见，我们选择传教士与野人开始所在的岸为所要表示的岸，并称其为左岸，另一岸称为右岸。显然仅用描述左岸的三元组就足以表示出整个情况了。

✓综上，我们的**状态空间**可表示为：**(ML，CL，BL)**，其中0≤ML，CL≤N，BL∈{0, 1}。

状态空间的总状态数为**(N+1)×(N+1)×2**，

✓ 问题的**初始状态**是(N，N，1)，**目标状态**是(0，0，0)。

转换模型

- 该问题主要有两种操作：从**左岸划向（条件）**右岸和从**右岸划向**左岸，以及每次摆渡的传教士和野人**个数变化（行动）**。
  我们可以使用一个2元组（BM，BC）来表示每次摆渡的传教士和野人个数，我们用i代表每次过河的总人数，i = 1~k，则每次有BM个传教士和BC=i-BM个野人过河，其中BM= 0~i，而且当BM!=0时需要满足BM>=BC。则

✓从左到右的操作为：（ML-BM，CL-BC，B = 1）

✓从右到左的操作为：（ML+BM，CL+BC，B = 0）

因此，当N=3，K=2时，满足条件的（BM，BC）有：

（0,1）、（0,2）、（0,3）、（1,0）、（1,1）、（2,0）、

（2,1）、（2,2）、（3,0）、（3,1）、（3,2）、（3,3）。

由于从左到右与从右到左是对称的，所以此时一共有24种操作。

目 录

有信息搜索

- **无信息搜索**又名盲目搜索：
  - 在搜索时，只有问题定义信息可用。
  - 盲目搜索策略仅利用了问题定义中的信息。

- **有信息搜索：**
  - 在搜索时，当有**策略**可以确定一个非目标状态比另一种更好的搜索，称为有信息的搜索。

回顾:树搜索

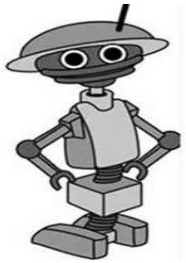
- 搜索策略就是节点扩展顺序的选择

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
```

5.1 最佳优先搜索

- **思想:** 使用一个**评估函数 f(n)**给每个节点估计他们的希望值。 优先扩展最有希望的未扩展节点。
- **实现:** **fringe**表中节点根据评估值从大到小排序
- **最佳优先**搜索策略有:
  - 贪婪最佳优先搜索
  - A* 搜索

贪婪最佳优先搜索



- 评估函数： f(n) = h(n) （heuristic，启发函数)

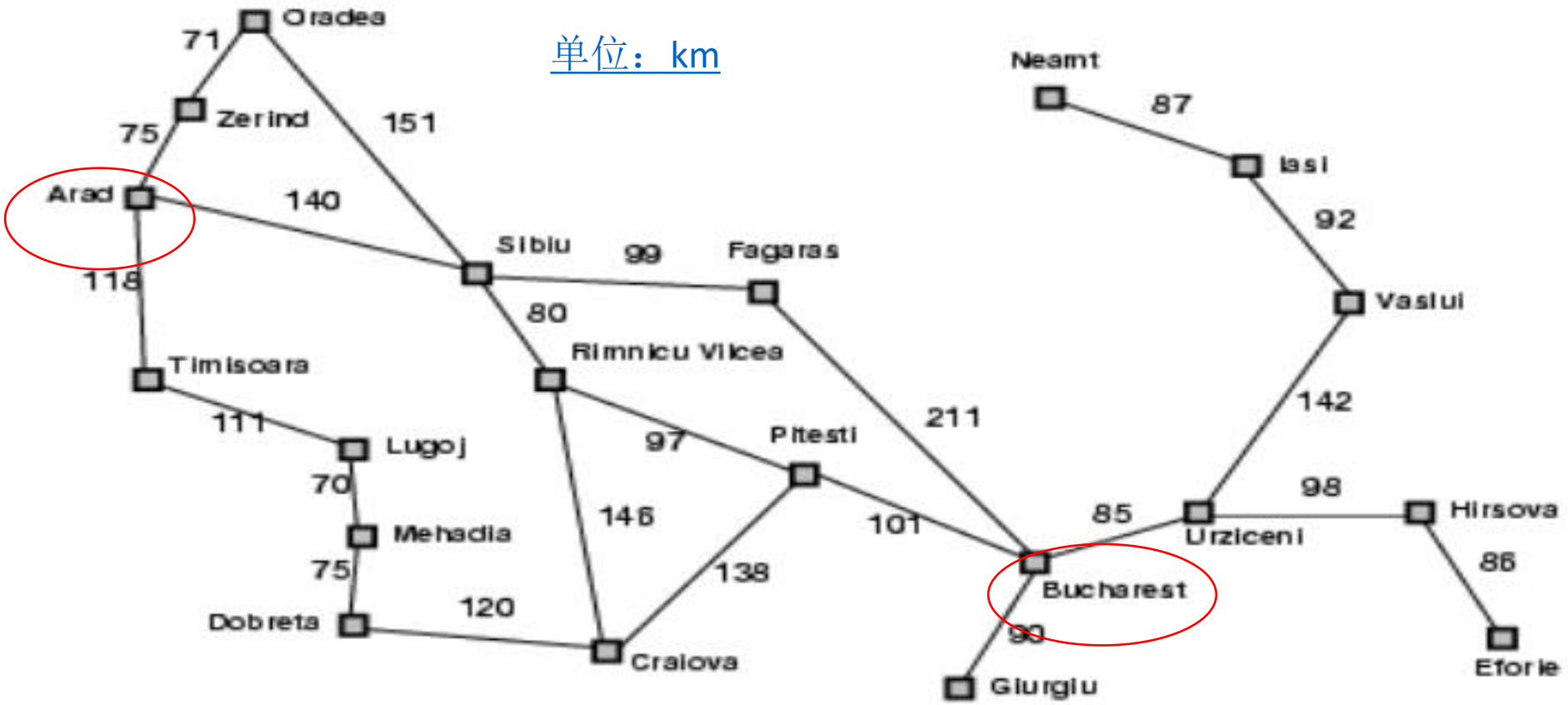
  = 估计从节点n到目标的代价

  例如：$h_{SLD}(n)$ = 从节点 n 到 Bucharest的直线距离

- **贪婪最佳优先搜索**优先扩展看上去更接近目标的节点（启发式函数评估出来的）。

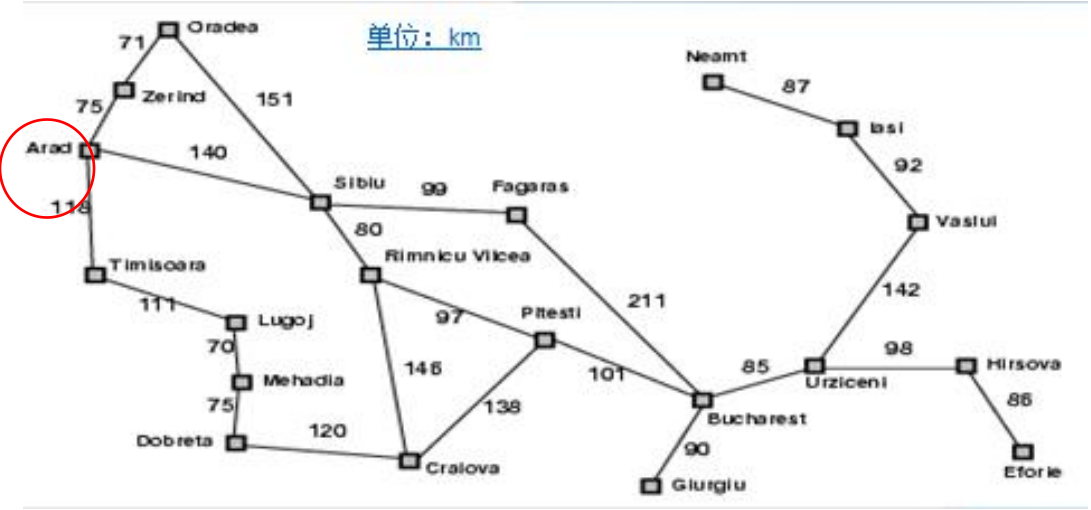
例：罗马尼亚问题

单位：km



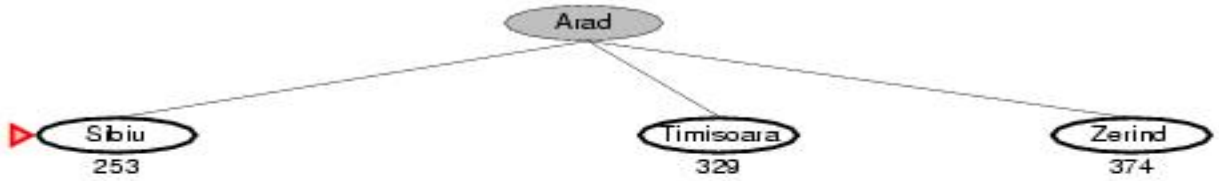
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

贪婪最佳优先搜索示例

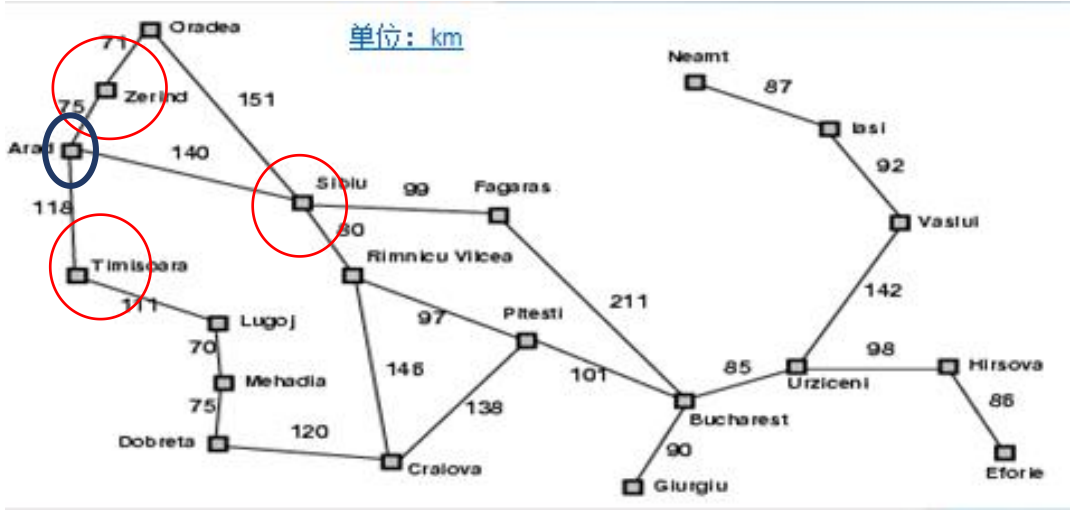


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

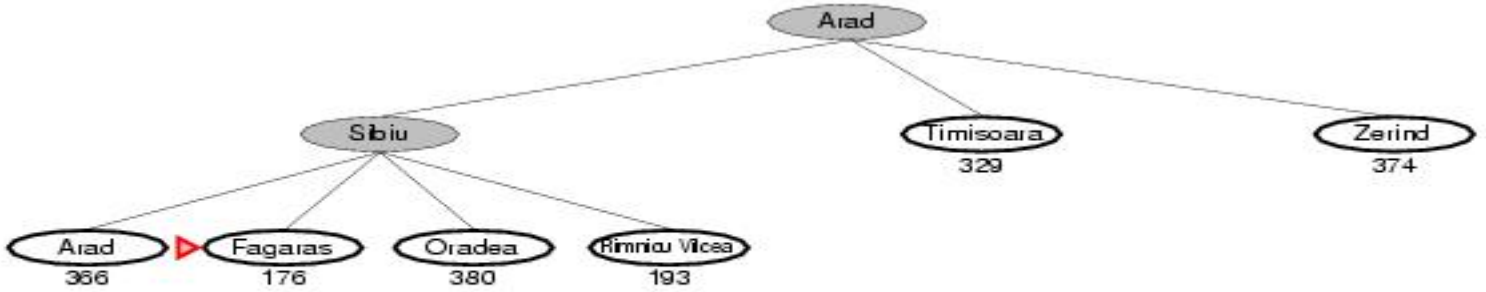
贪婪最佳优先搜索示例

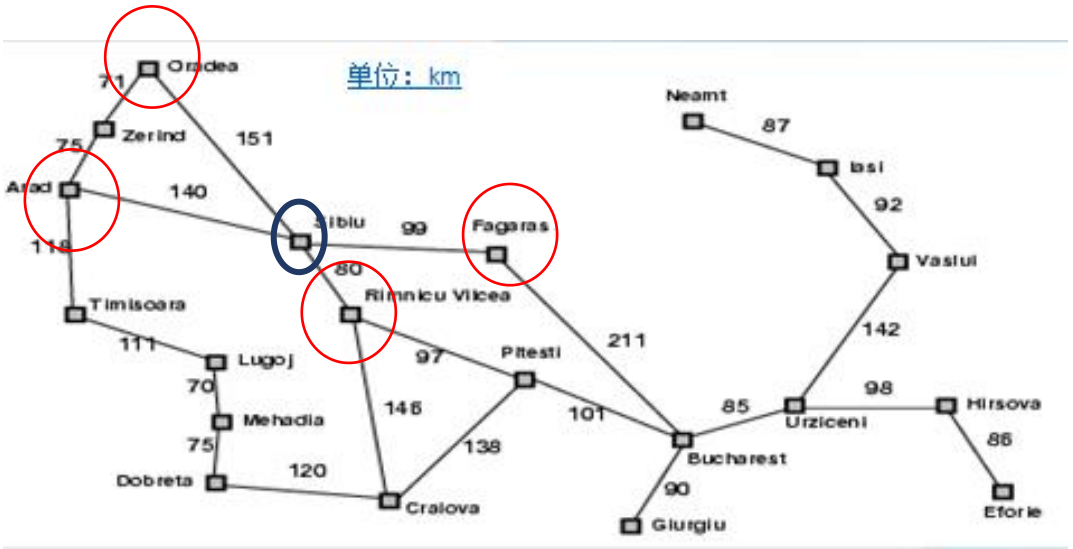


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



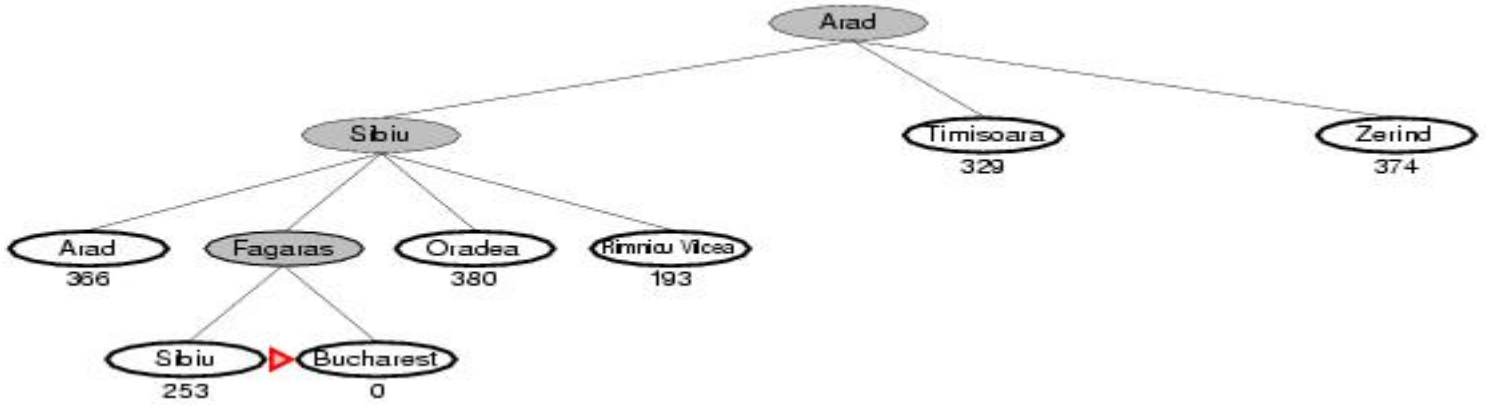
贪婪最佳优先搜索示例



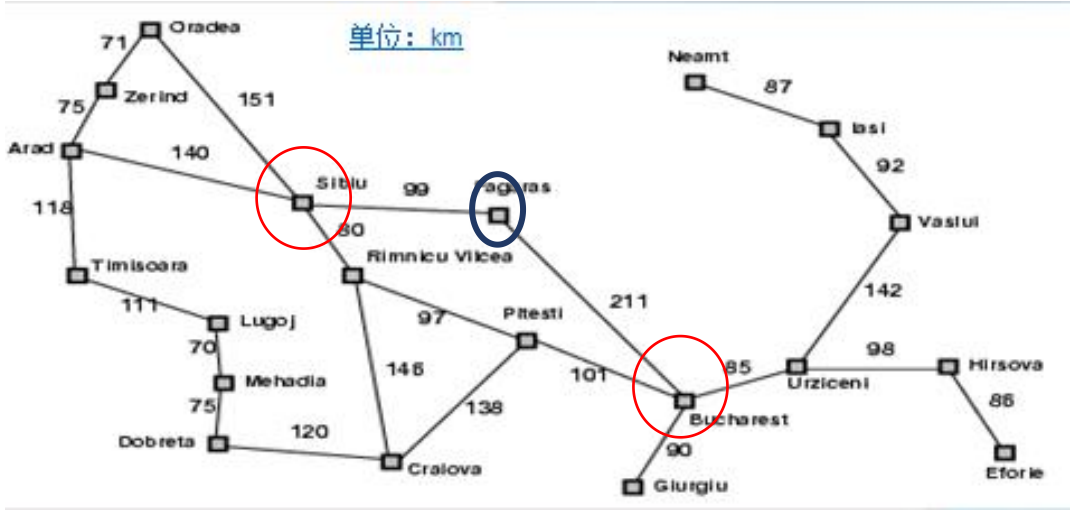


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

贪婪最佳优先搜索示例

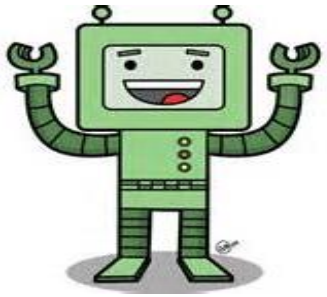


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

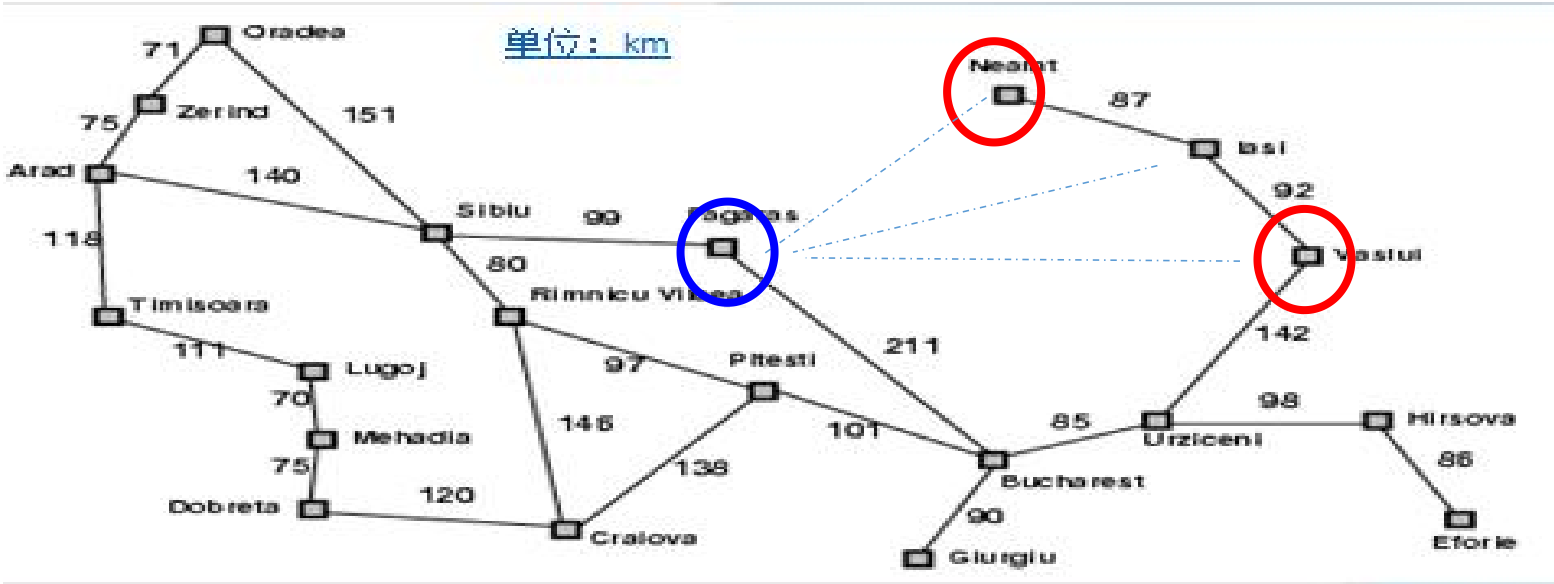


路径代价：140+99+211=450

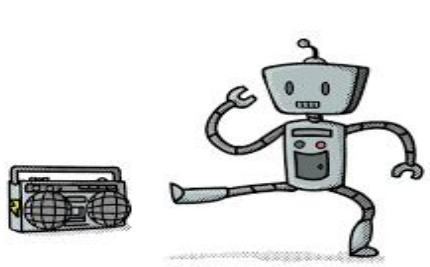
贪婪最佳优先搜索的属性



- 完备性? No – 可能陷于死循环当中,
- 时间? 比如, Iasi → Neamt → Iasi → Neamt → *O*(b^m), 但一个好的启发式函数能带来巨大改善
- 空间? *O*(b^m)
- 最优性? No

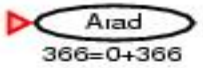


5.2 A* 搜索

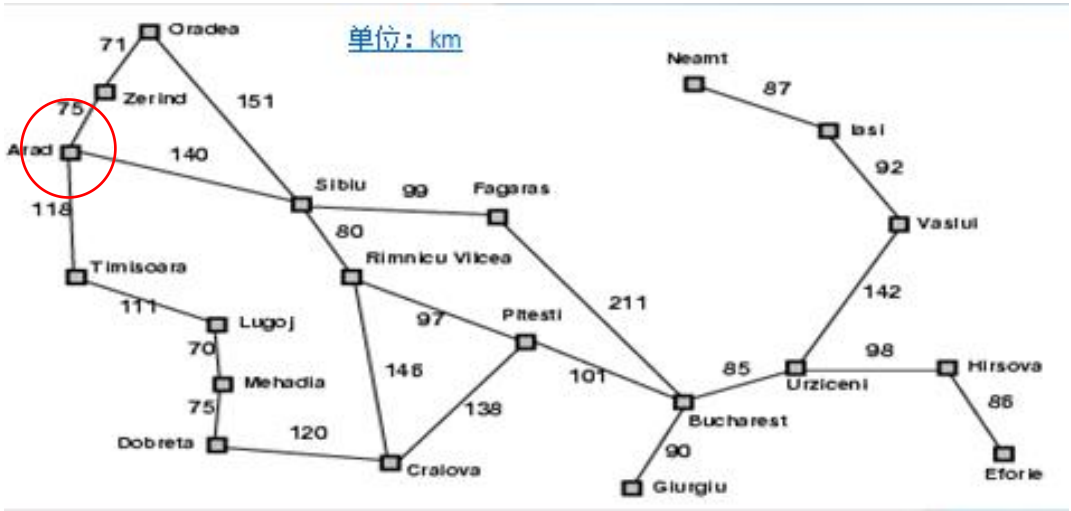


- **思想:** 避免扩展代价已经很高的节点。
- 评估函数 $f(n) = g(n) + h(n)$
  - $g(n)$ = 到达节点n已经发生的实际代价
  - $h(n)$ = 从节点n到目标的代价估计值
  - $f(n)$ =评估函数，估计从初始节点出发，经过节点n，到目标的路径代价的估计

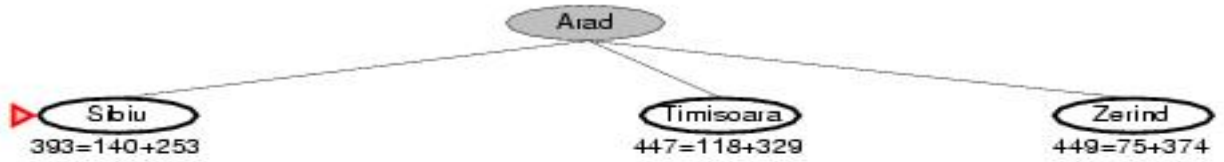
A* 搜索示例



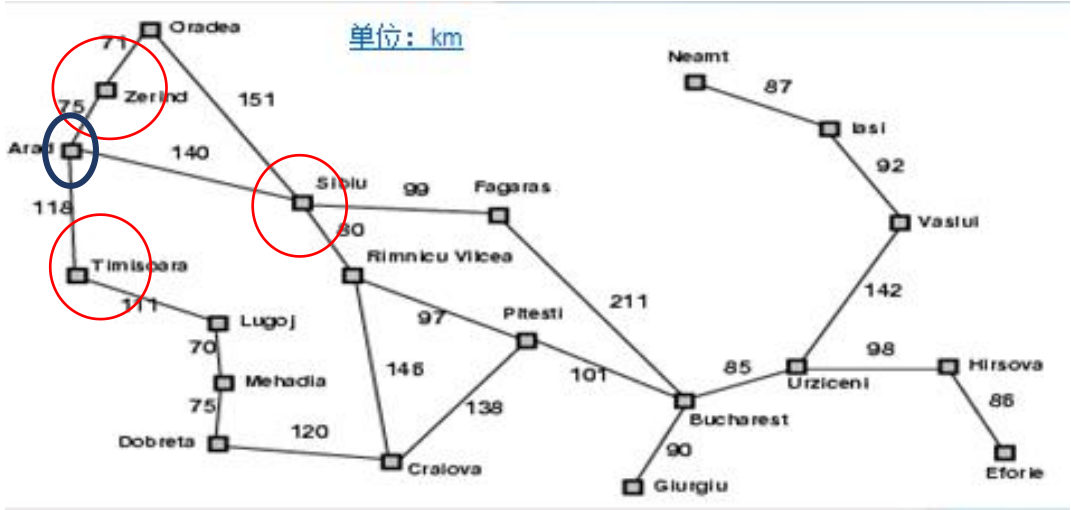
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



A*搜索示例

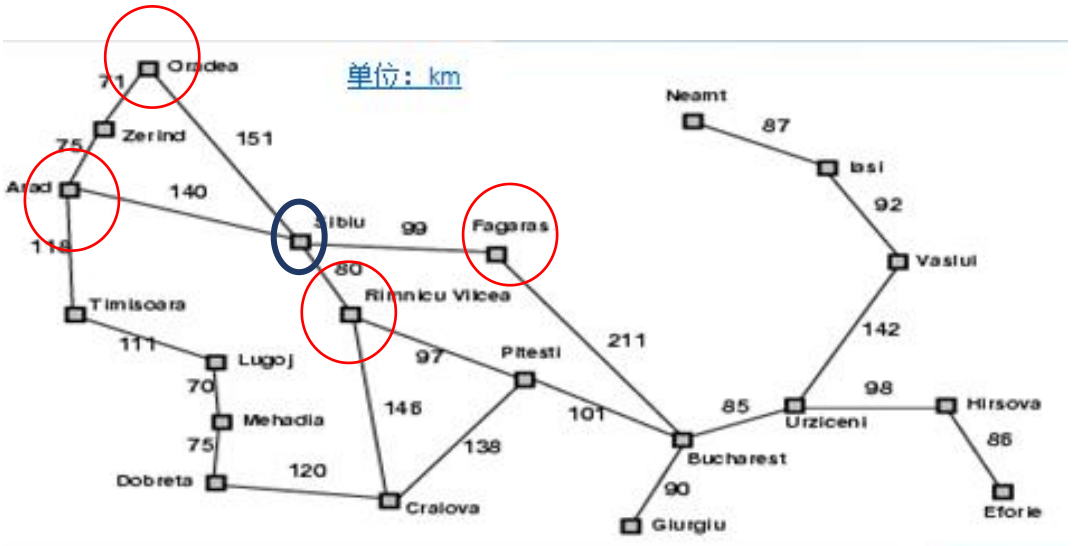


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

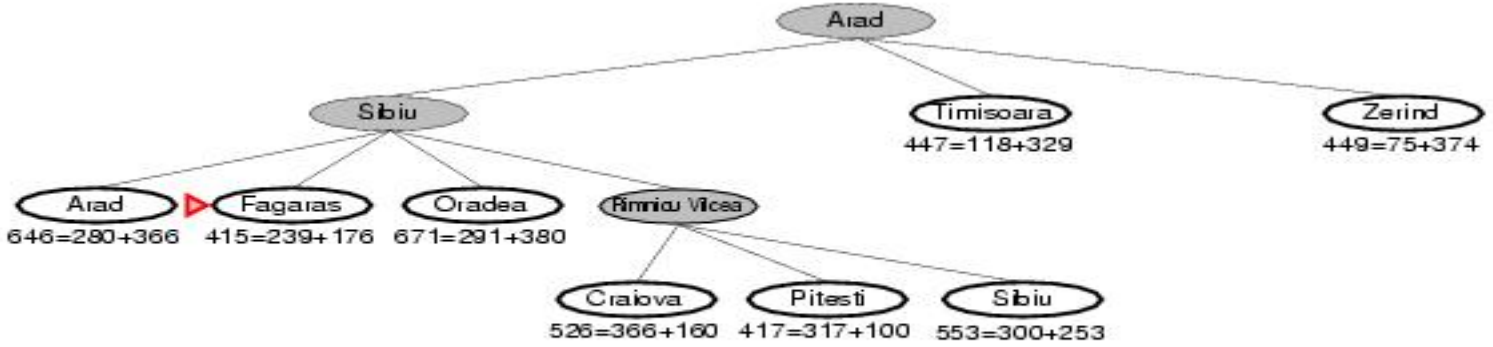


A*搜索示例

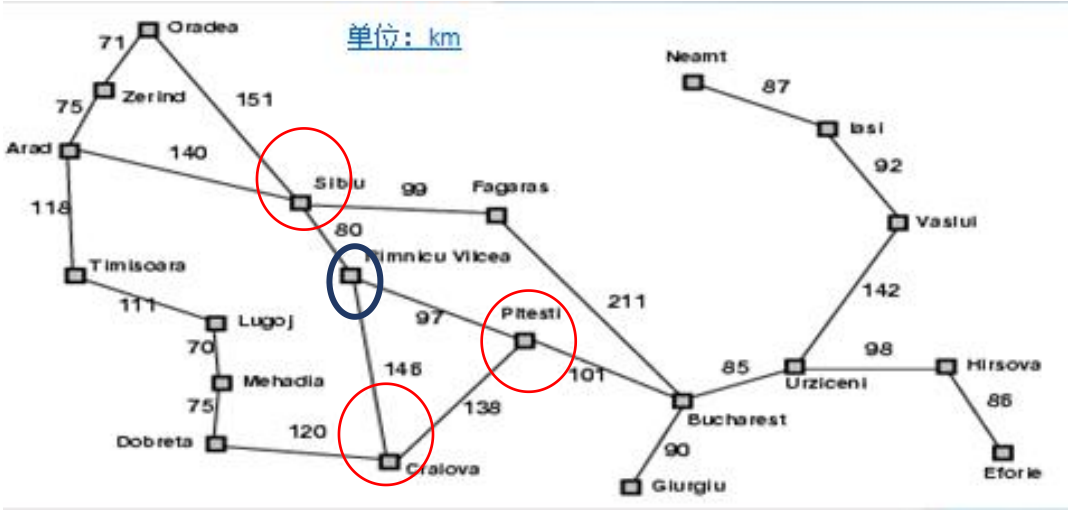
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



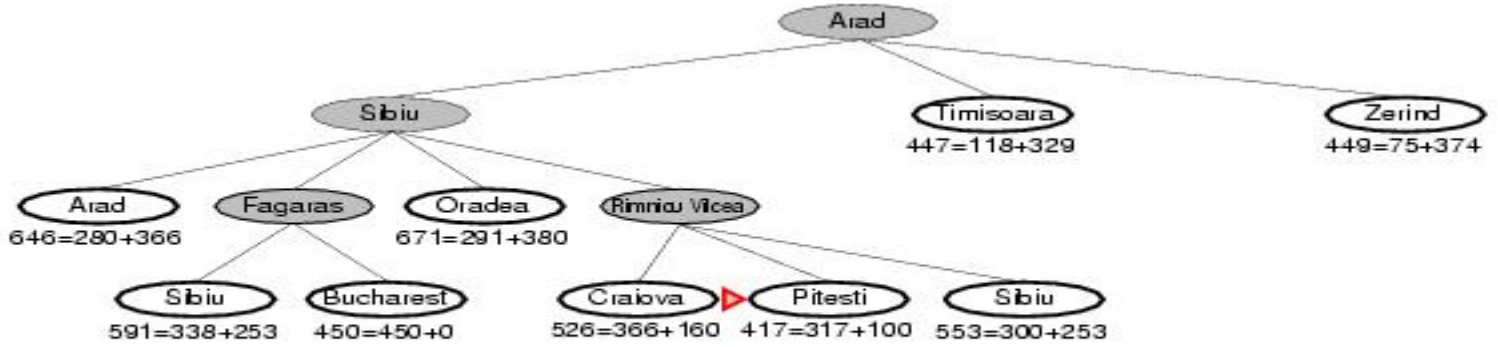
A*搜索示例



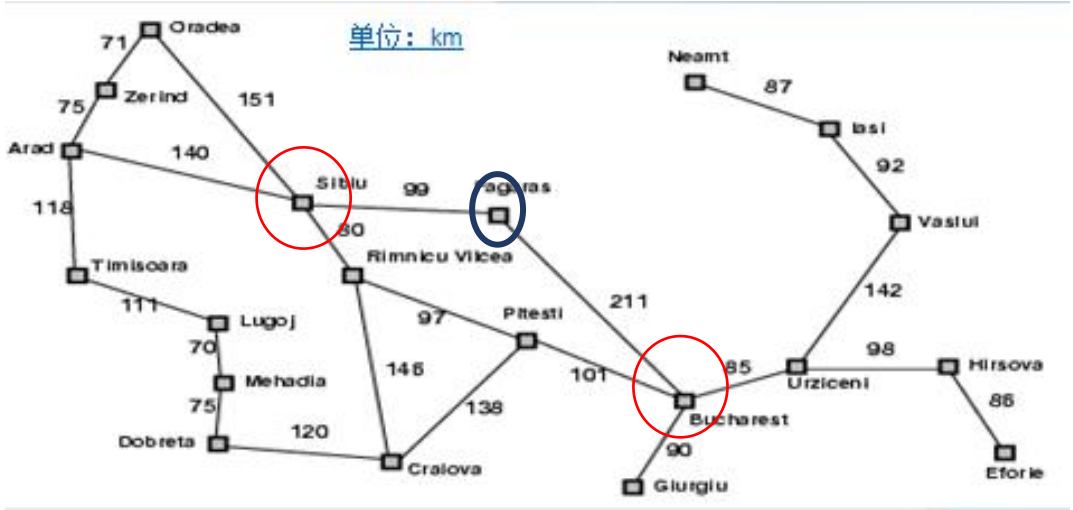
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



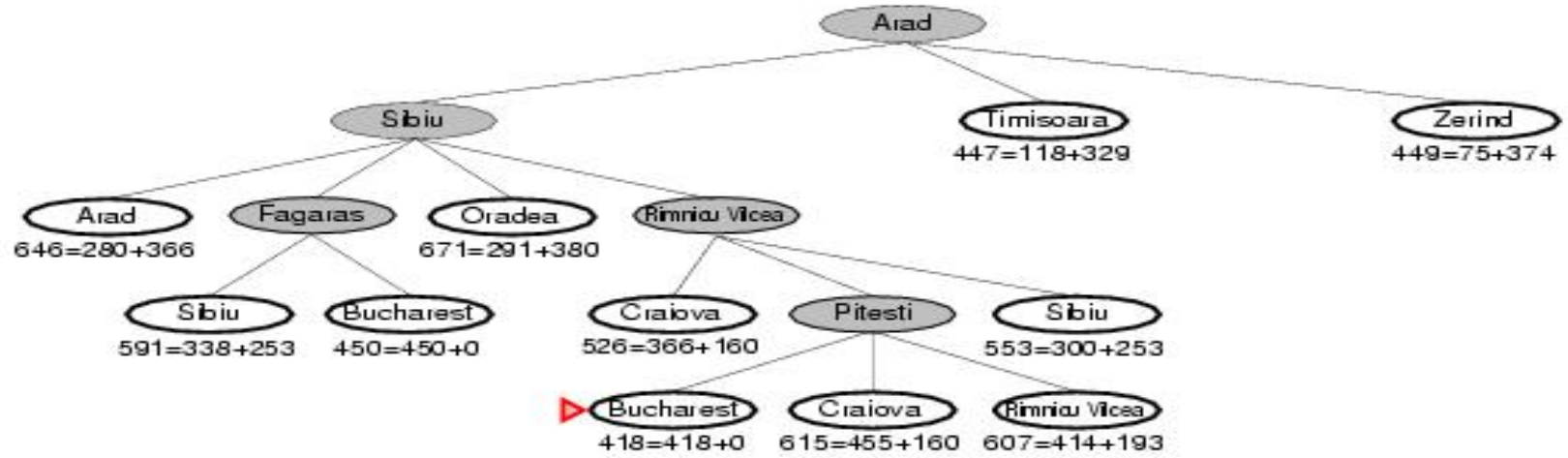
A*搜索示例

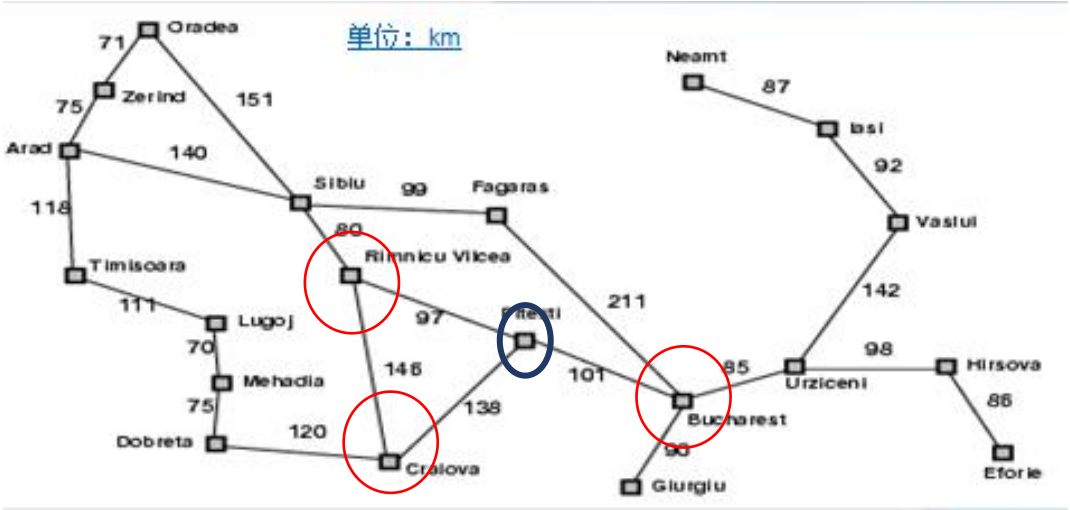


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374



A*搜索示例





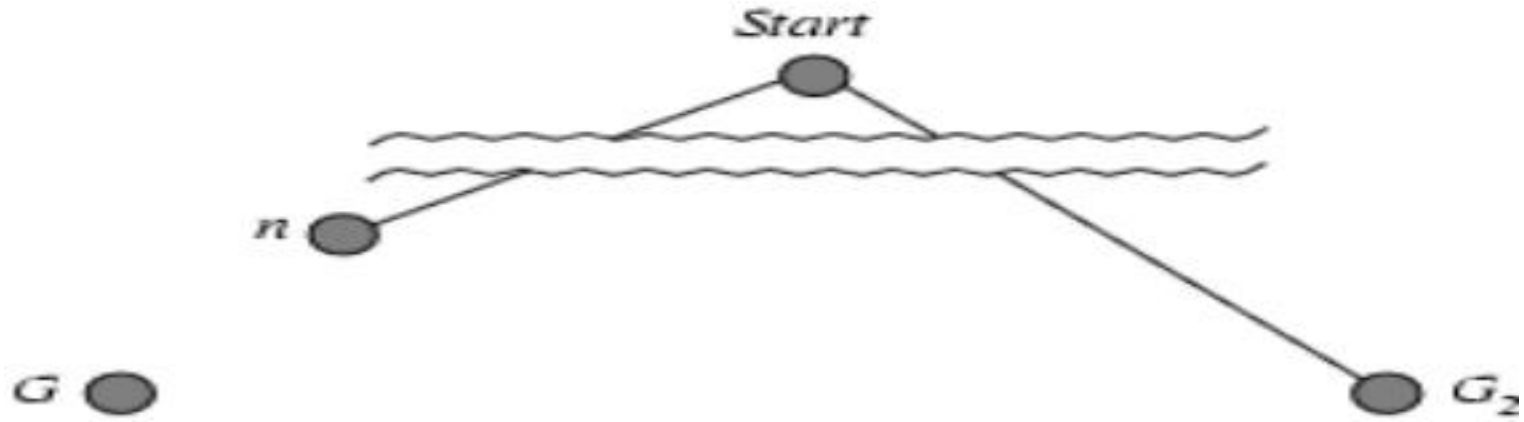
Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

可采纳的启发式函数（admissible heuristic）

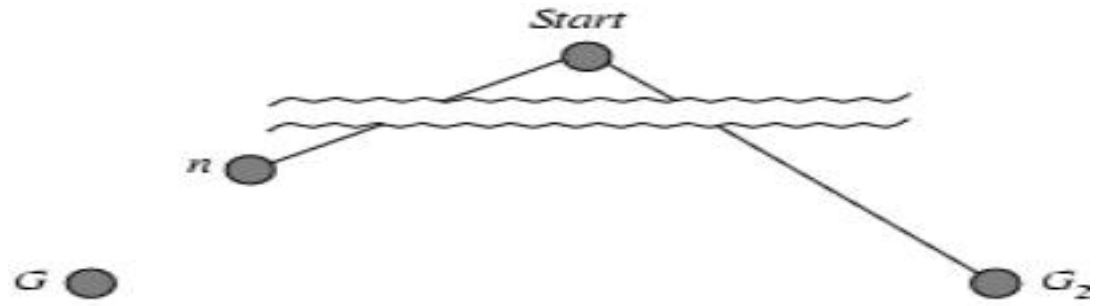
- 如果启发式函数*h(n)*对于任意的节点*n*都满足 $h(n) \le h^*(n)$，这里 $h^*(n)$是指从节点n到达目标的真正代价，则称*h(n)*是可采纳的（admissible heuristic）。
- **定理:** 如果*h(n)*是可采纳的，则 A* 树搜索算法是具有最优性。

A* 的最优性证明

- 假设某次优目标节点G_2 已经产生并在fringe表中排队，设n是fringe表中到达最优目标节点G的最短路径上的一个未扩展节点



A* 的最优性



①$f(G_2) = g(G_2)$

②$f(G) = g(G)$

③$g(G_2) > g(G)$

④$f(G_2) > f(G)$

⑤$h(n) \le h^*(n)$

⑥$g(n) + h(n) \le g(n) + h^*(n)$

⑦$f(n) \le f(G)$

⑧$f(G_2) > f(n)$

所以 A* 决不会在选择节点n之前选择次优节点 G_2 扩展

A*搜索算法的性质

- 完备性? Yes
- 时间? 指数级
- 空间? 将所有产生的节点存储在内存中
- 最优性? Yes