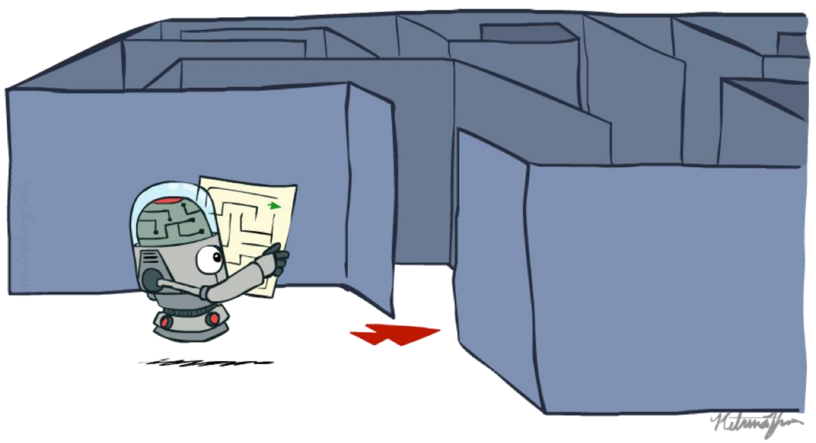
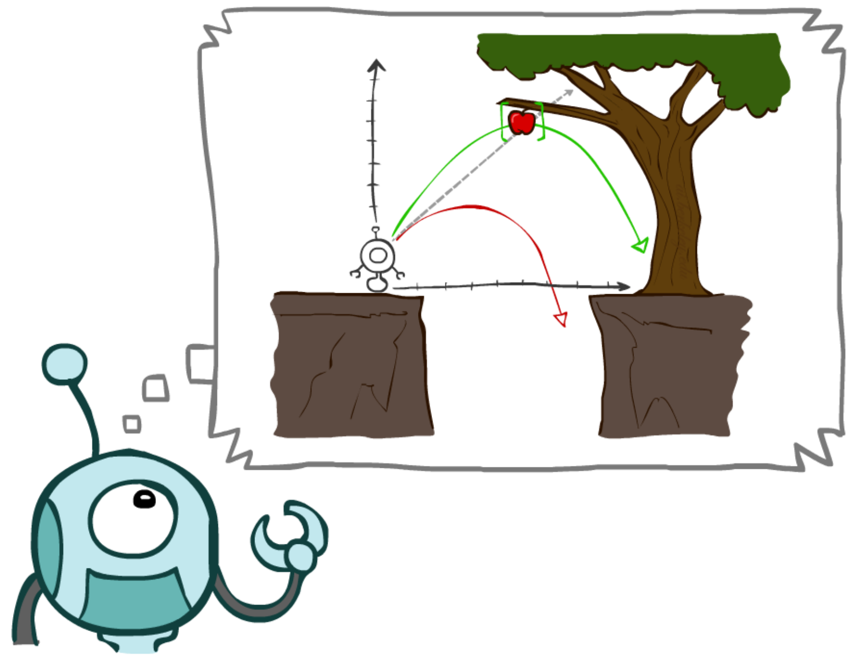
人工智能导论：搜索



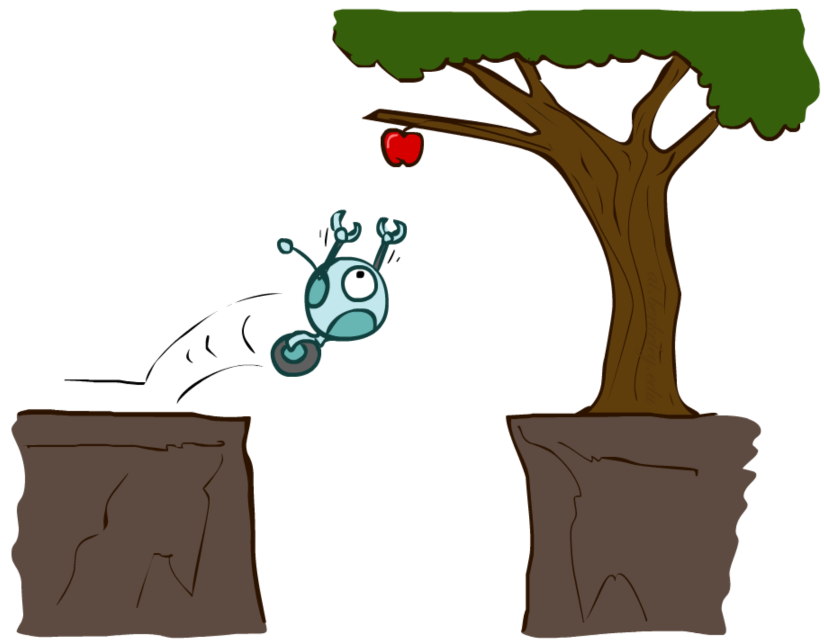
提纲

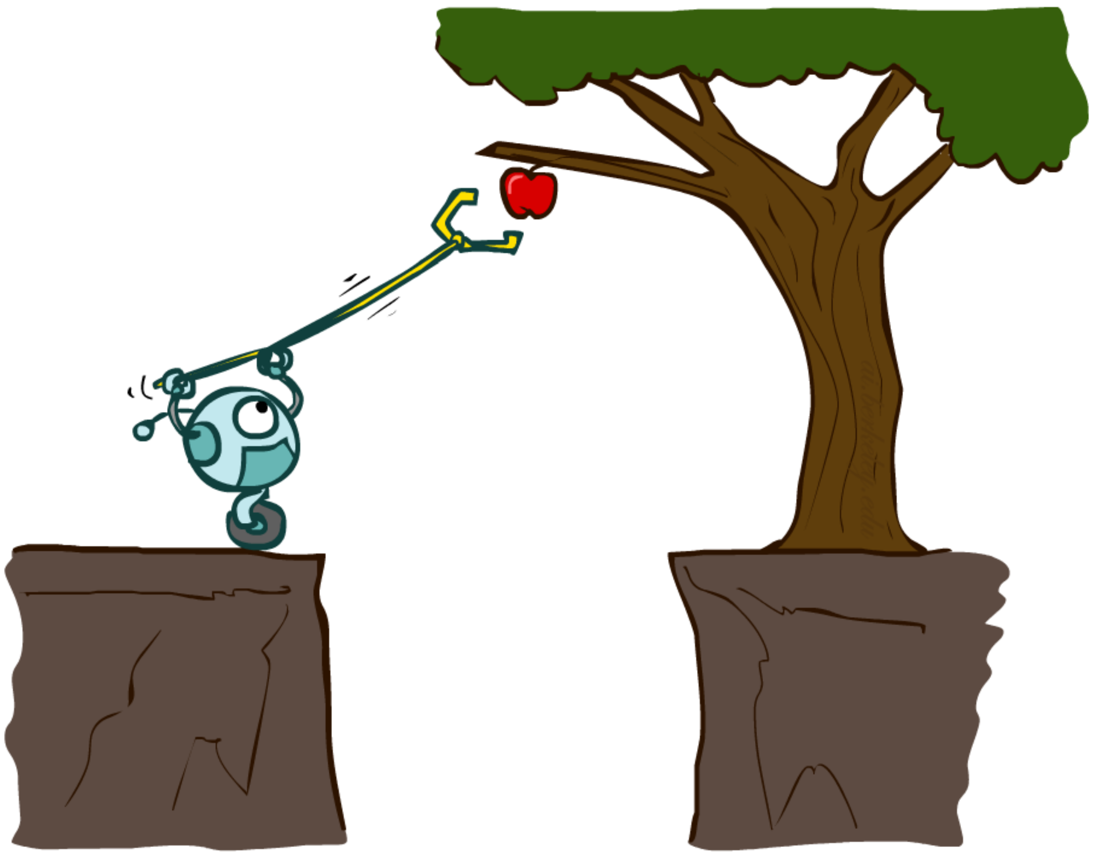
- 提前计划的智能体
- 搜索问题
- 基础搜索法
  - 深度优先搜索
  - 广度优先搜索
  - 基于成本的统一搜索



对比反射智能体

- 仅根据当前的感知选择行动
- 也许有记忆或有一个世界当前状态的模型
- 不考虑行动的未来结果

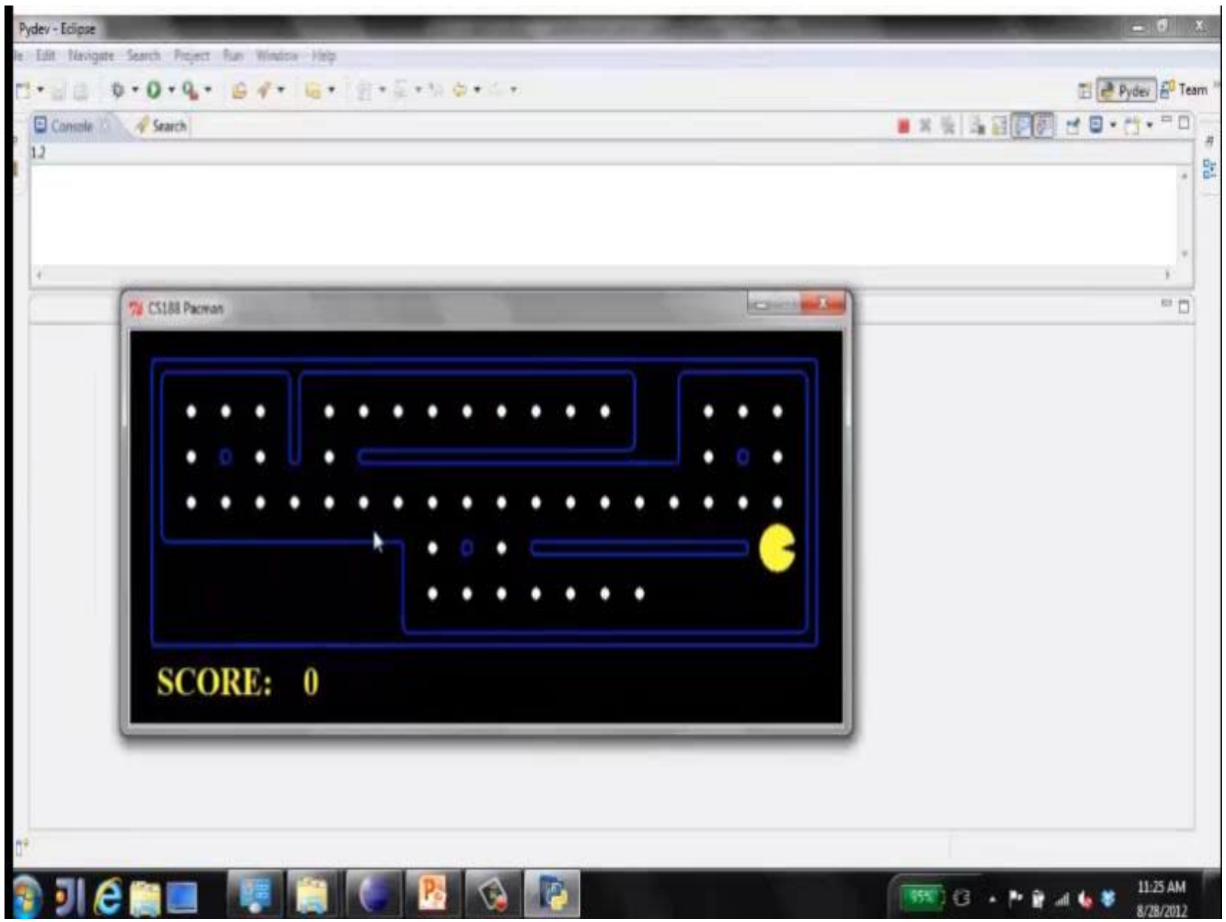




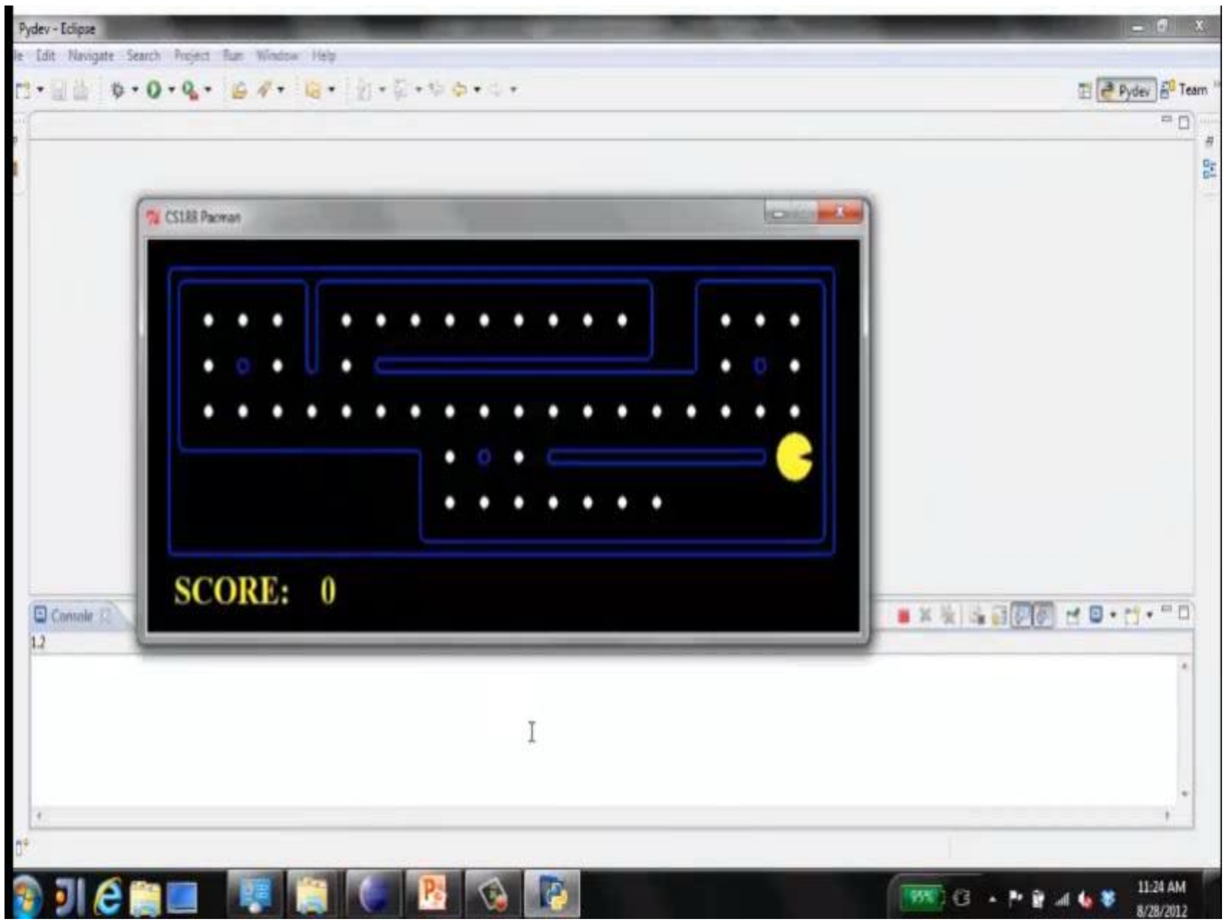
提前计划的智能体

- 做计划的智能体
  - 决策基于对行动的预测结果
  - 一定有一个转换模型（transition model）：根据不同的行动，环境是如何演变的
  - 必须设定一个目标
- 可能的策略
  - 在开始之前制定一个完整、优化的计划(offline plan)，然后再执行。
  - 先制定一个简单、贪心(greedy)的计划，开始执行；当走错的时候再重新计划。

完整计划后再执行—演示视频

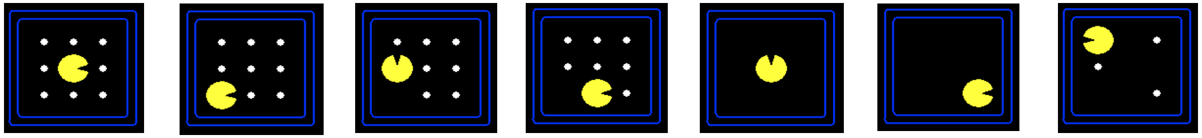


行动后改变计划—演示视频

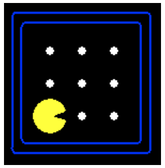


搜索问题

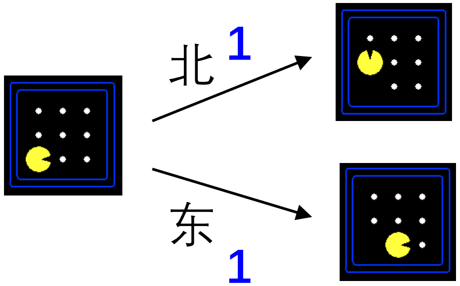
搜索问题



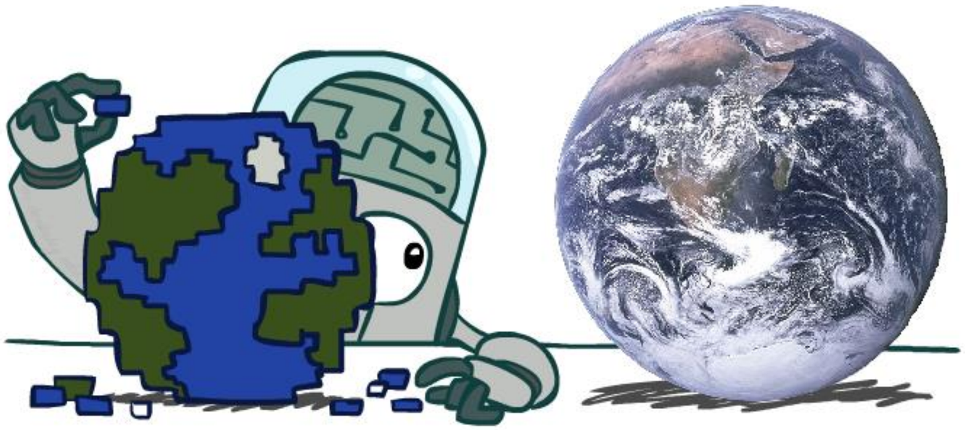
- 一个搜索问题包括:
  - 一个状态空间
  - 在每一个状态里，一个可允许的动作集合
  - 一个转换模型　Results(s, a)
  - 一个步骤成本函数　cost(s, a, s')
  - 一个开始状态，和一个目标到达测试



{北, 东}



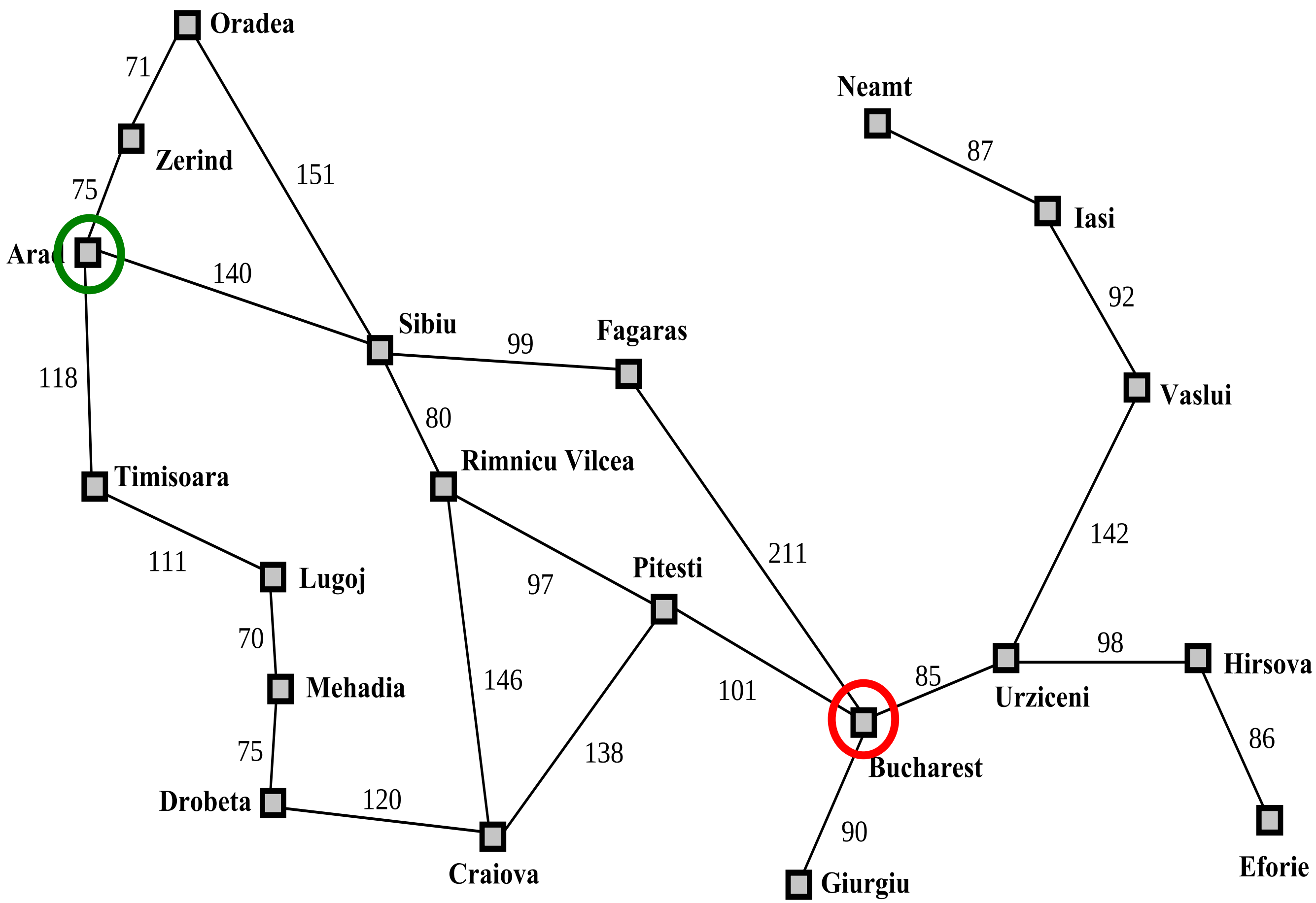
- 一个解决方案是一序列动作（一个规划），从开始状态到一个目标状态



搜索问题

- 定义搜索问题是一个建模的过程
  - 抽象的数学描述

举例：在罗马尼亚旅行



- 状态空间
  - 城市
- 动作
  - 去一个邻居城市
- 转换模型
  - Result(A,Go(B))=B
- 步骤成本
  - 距离
- 开始状态
  - Arad
- 目标测试
  - 当前城市==Bucharest?
- 解决路径？

状态空间可能会很大

- 真实世界里的状态包括许多细节
- 搜索问题定义的状态是一种抽象，去掉不必要的细节

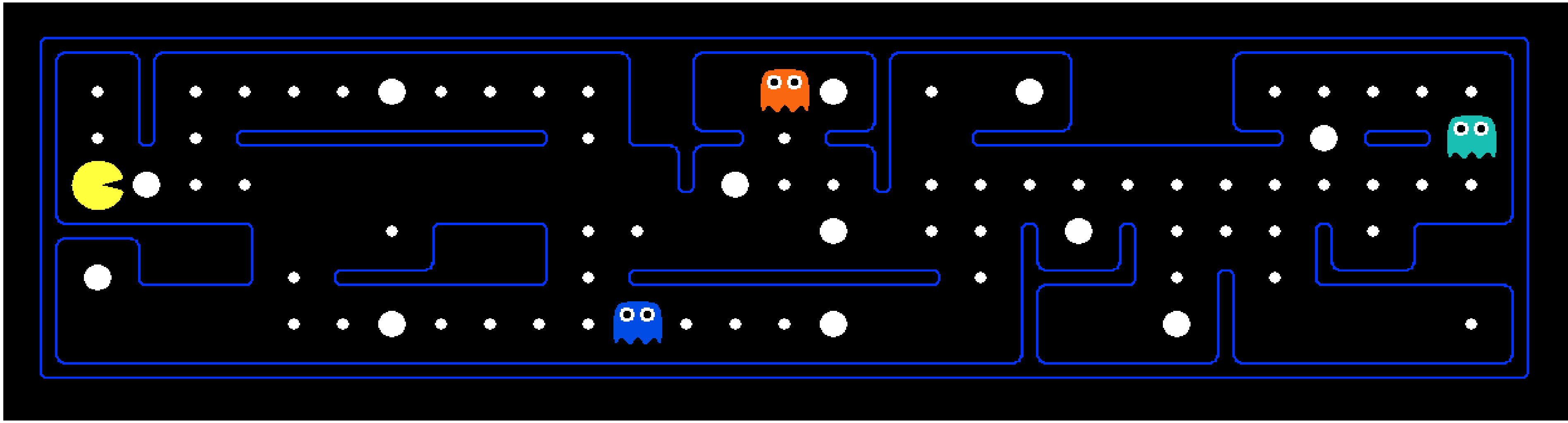
- 路径搜索问题
  - 状态：(x,y) 位置
  - 行动：北南东西
  - 转换模型：更新位置
  - 目标测试：(x, y) == 终点

MN 个状态 （|x|=M, |y|=N）

Pacman吃豆子问题
- 状态：((x, y)，豆子布尔(存在为真，没有为假))
- 行动：北南东西
- 转换模型：更新位置，和豆子布尔值
- 目标测试：豆子布尔全都是假

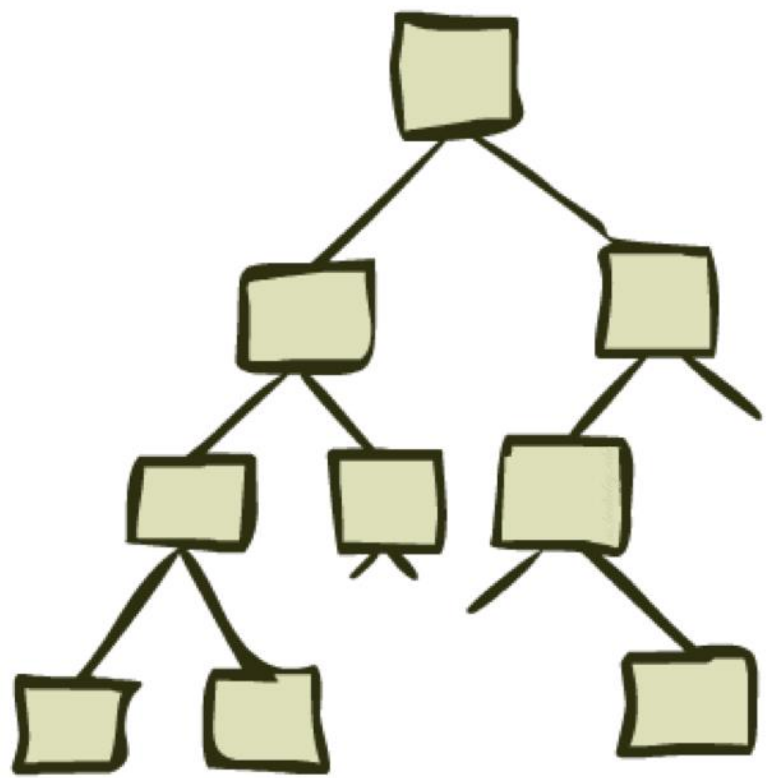
- $MN2^{MN}$ 个状态

更复杂的情况：安全通道



- 问题：吃掉所有豆子，同时保持所有怪物都在被威慑状态（吃到大的豆子妖怪会失去力量一段时间，此时pacman可以吃掉妖怪）
- 状态空间如何定义？
  - 智能体位置，豆子布尔值，能量丸（大豆子）布尔值，剩余威慑时间

状态空间图 和 搜索树



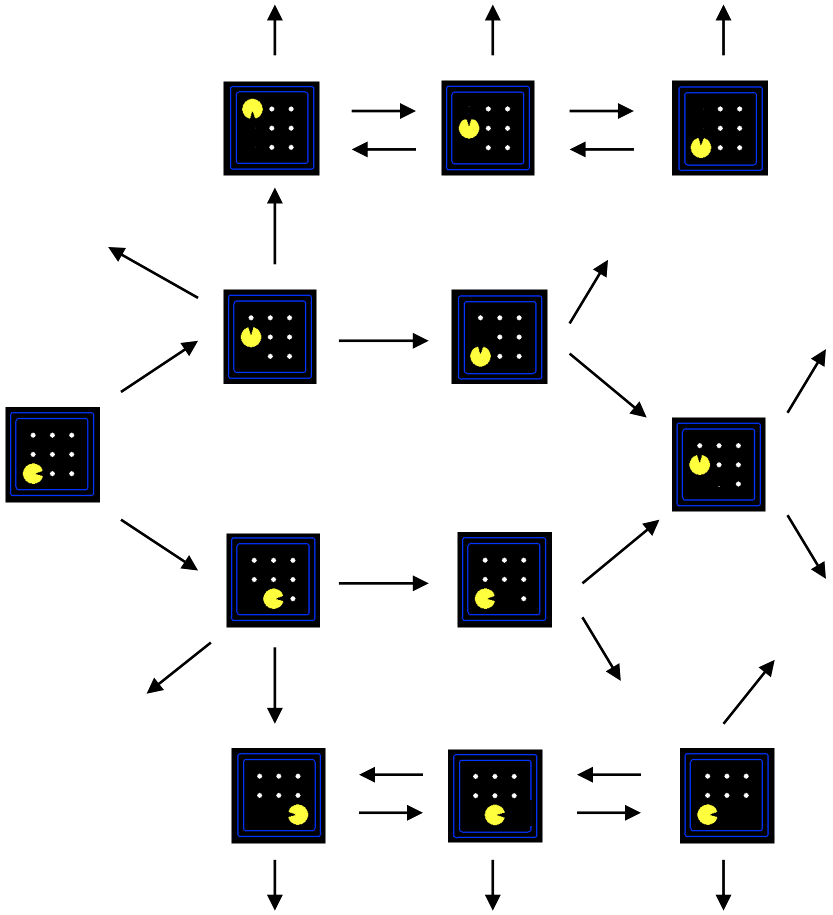
区别在哪里?

状态空间图

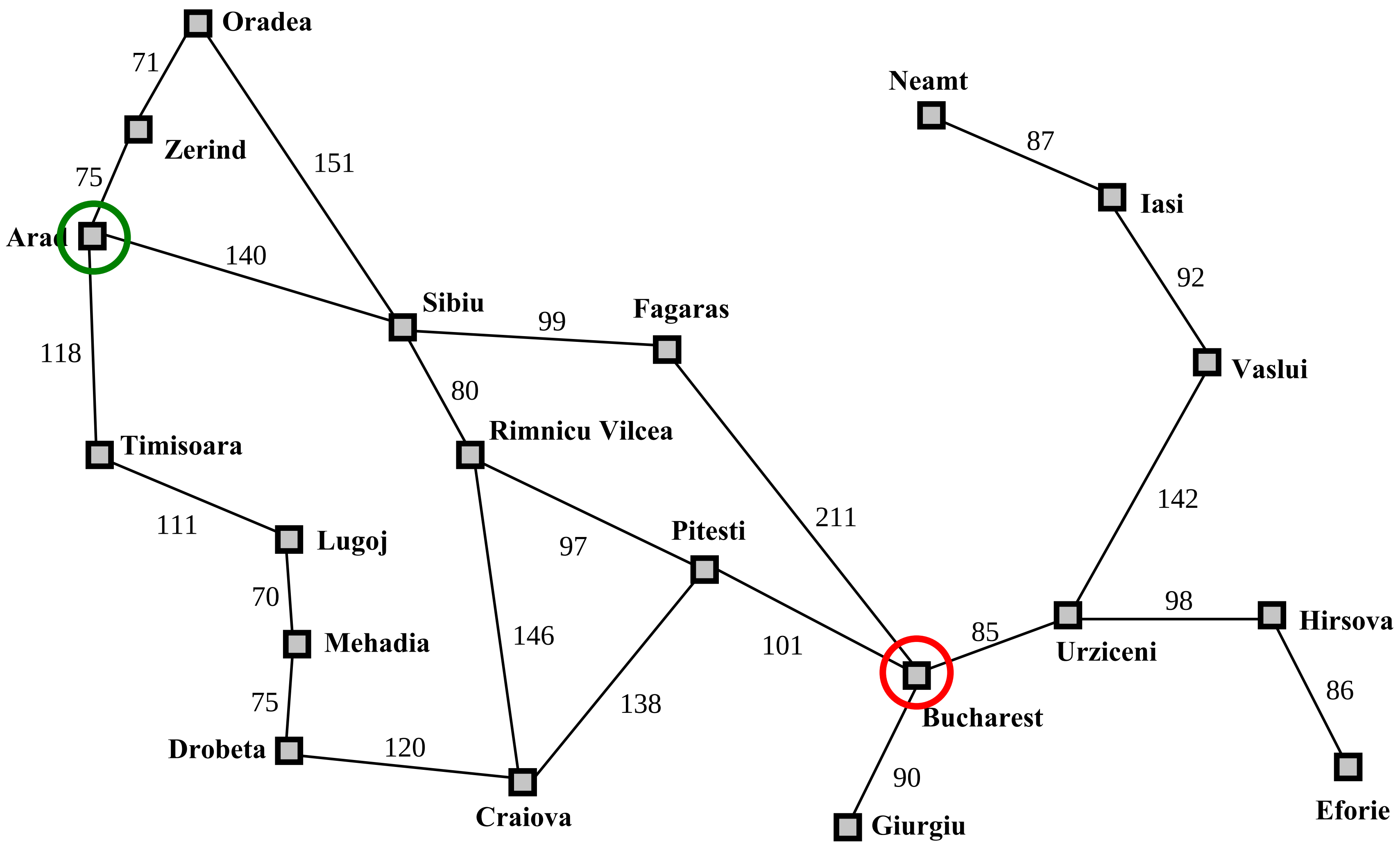
- 对搜索问题的数学表达
  - 节点是搜索问题中定义的状态
  - 边代表动作所导致的转换
  - 目标测试是当前节点是否是目标节点

- 每个状态只出现一次!

- 状态图有时很大难以完全构建，但它是一个有用的概念。

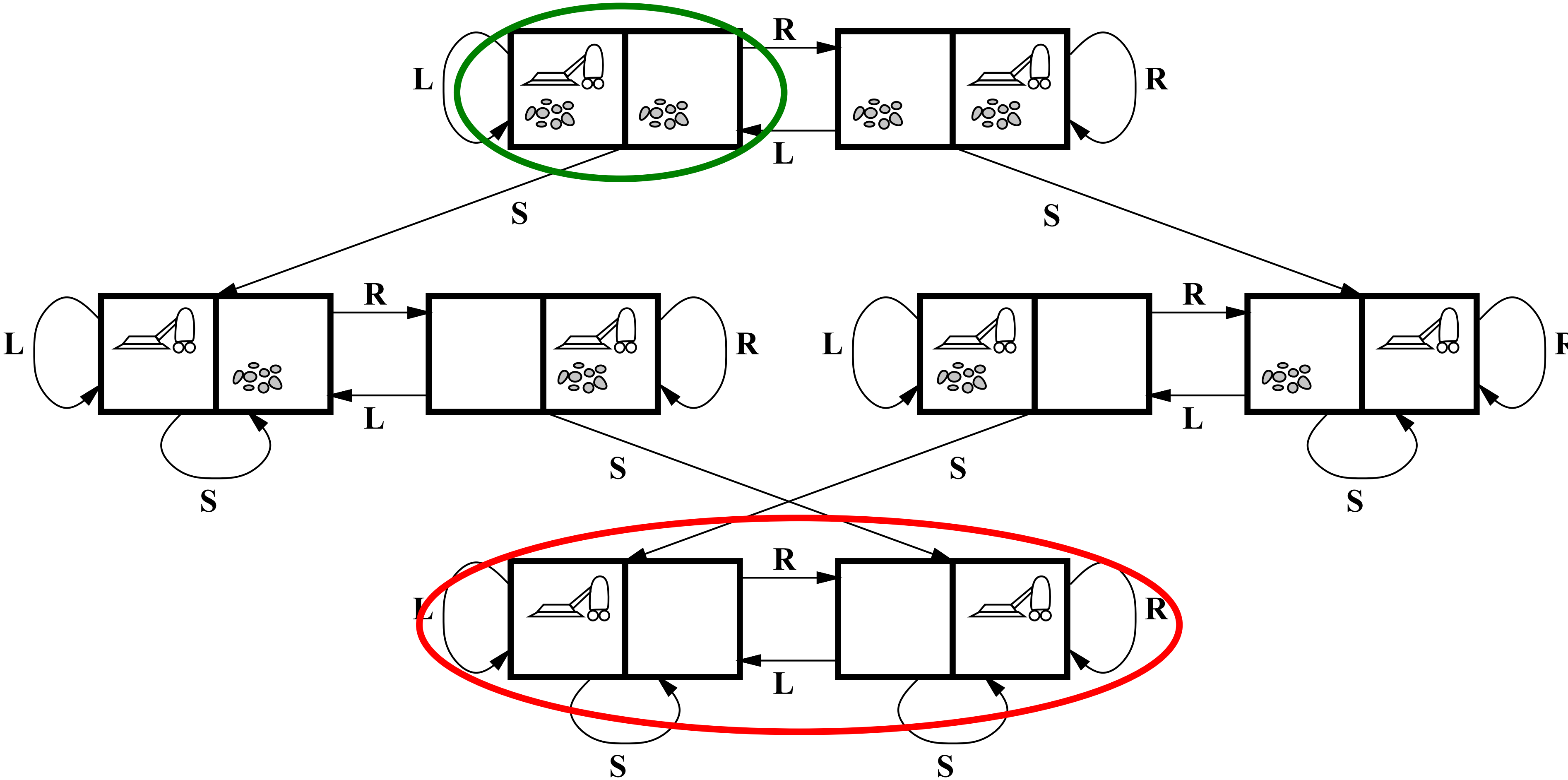


状态图举例

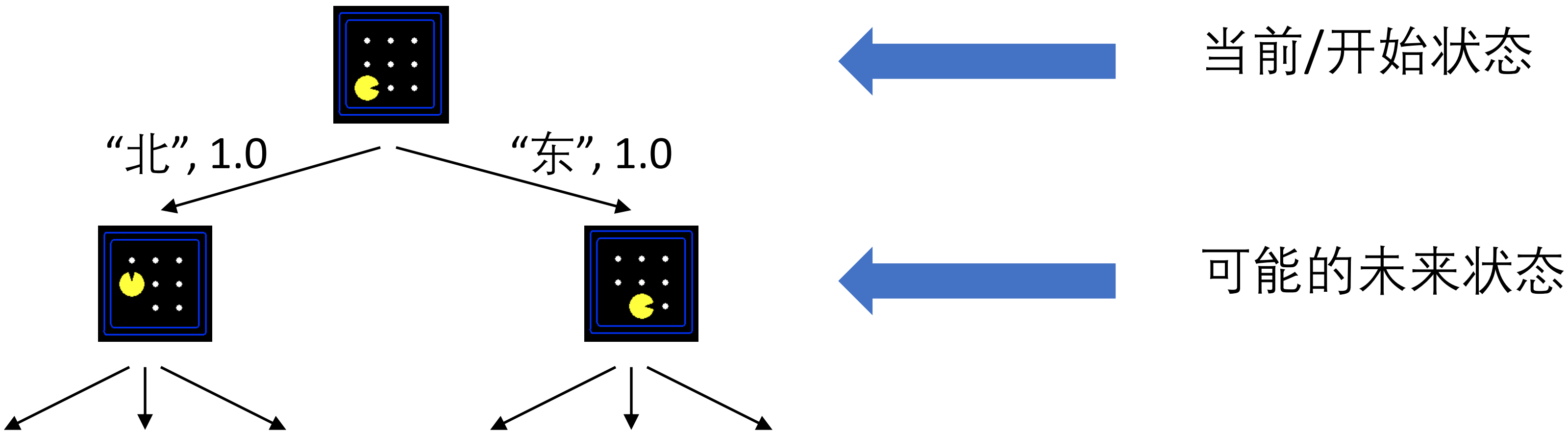


Oradea
71
Zerind
151
75
Arad
140
118
Sibiu
99
Fagaras
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
Drobeta
120
138
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
85
Urziceni
Bucharest
90
Giurgiu
86
Eforie

状态图举例

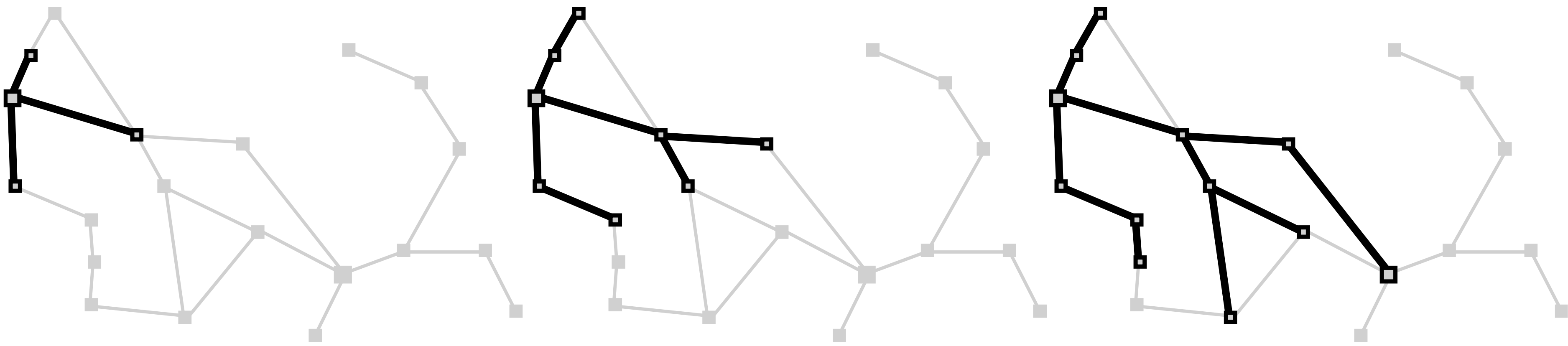


搜索树



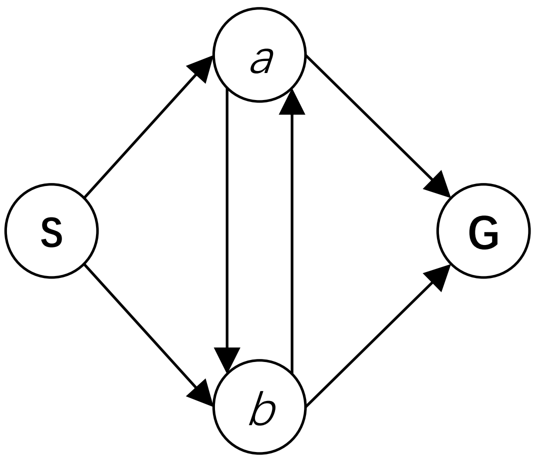
- 树上反映的是可能的行动规划及其结果。
- 开始状态是根节点。
- 子节点反映的是可能的动作结果。
- 节点代表状态，但一个状态可能出现多次，反映的是不同行动规划的结果。
- 对于大多数问题，我们实际上很难建立整个搜索树。

状态图 vs 搜索树

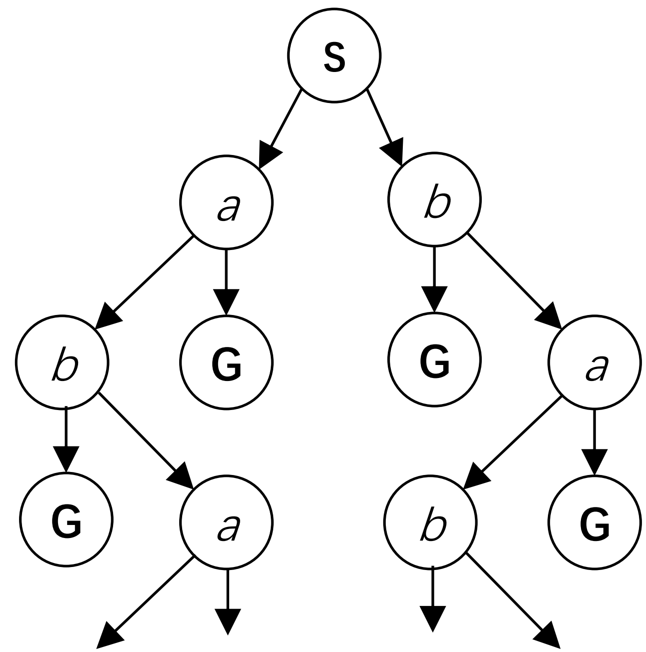


状态图 vs 搜索树

有一个4节点状态图：



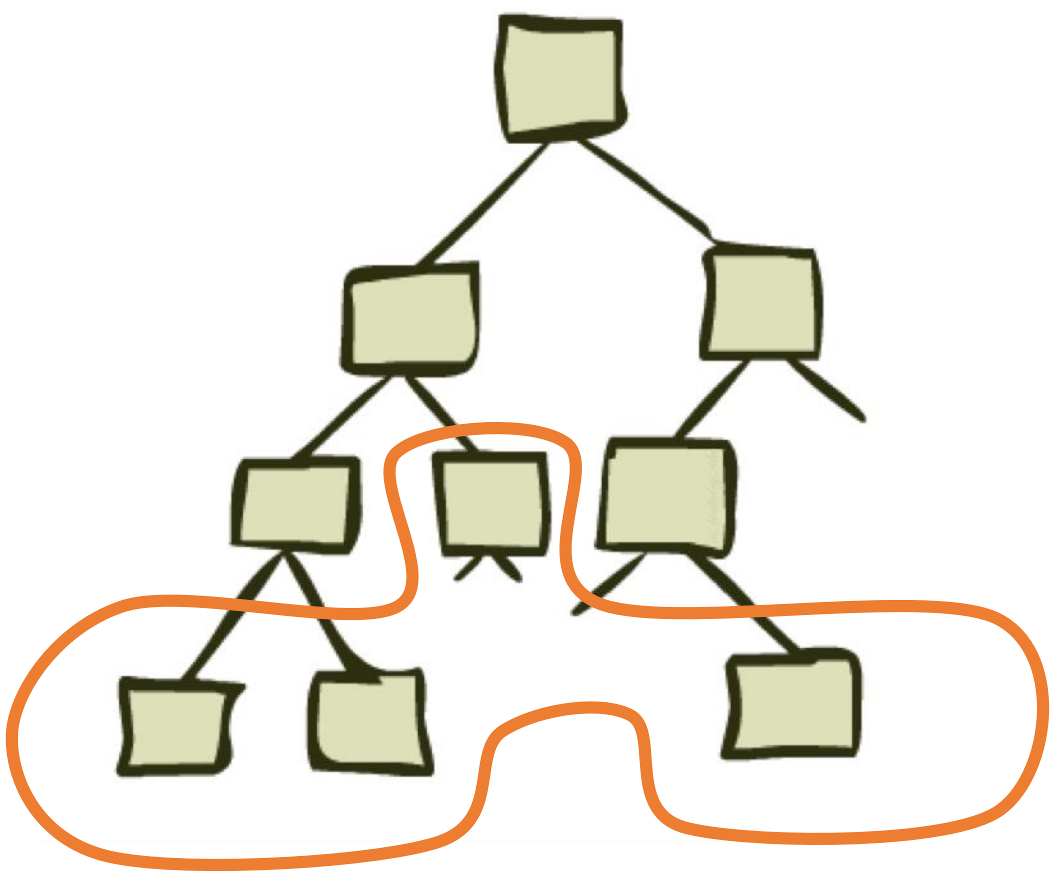
它的搜索树有多大（从S 状态开始）？



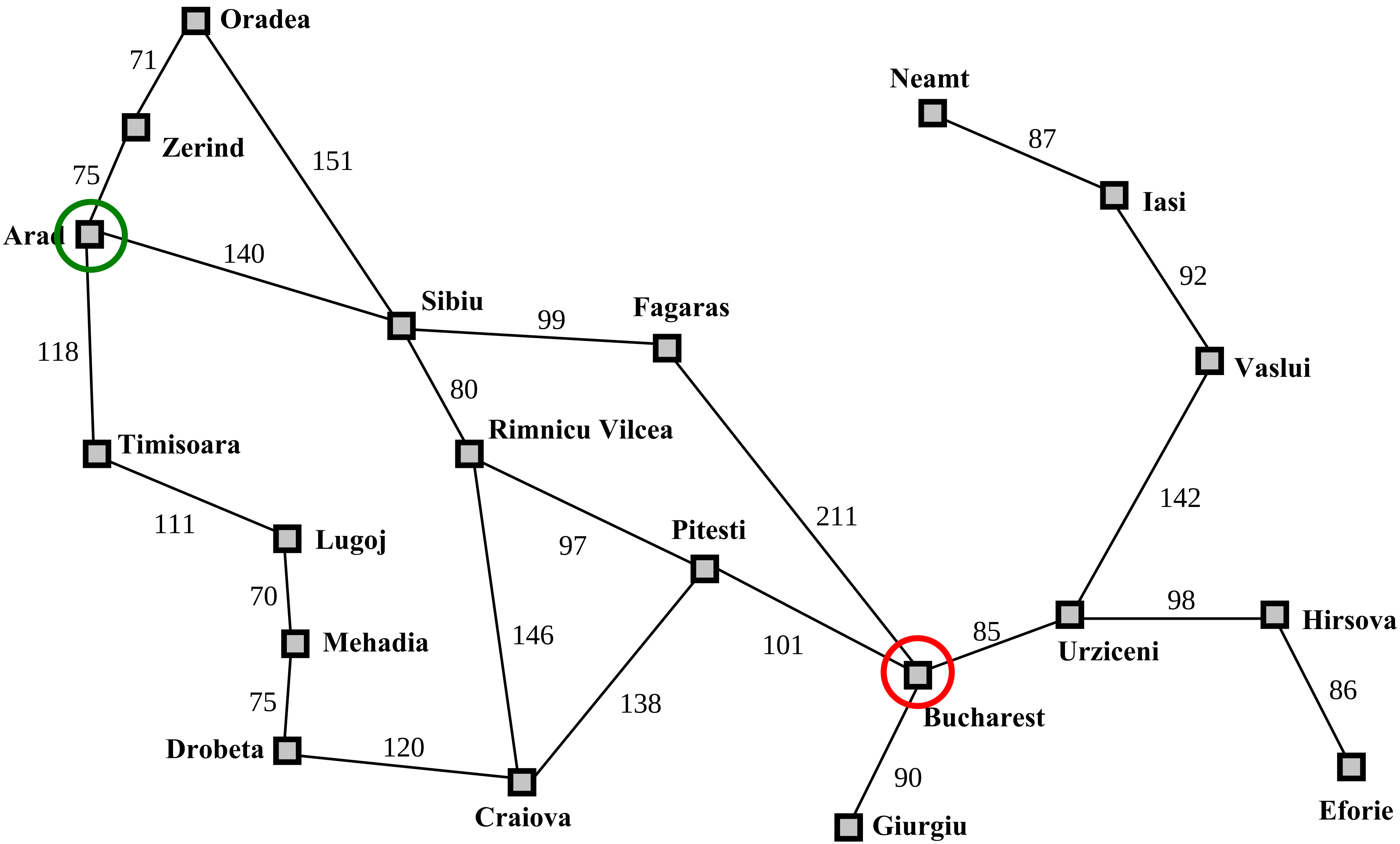
∞

- 存在许多重复的树枝结构!

树搜索

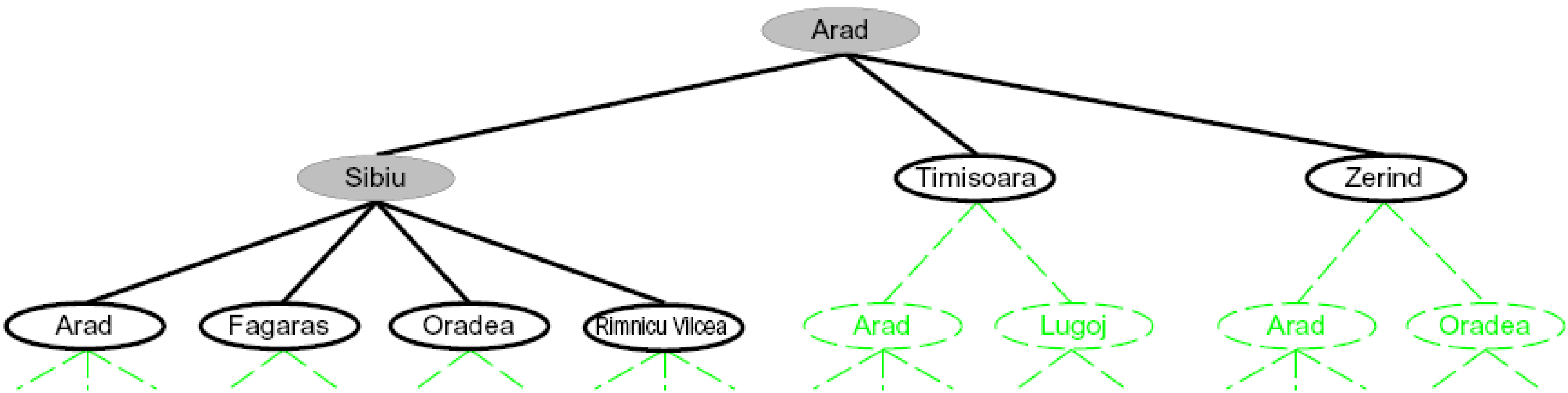


搜索问题：罗马尼亚旅行



Oradea
71
Zerind
151
75
Arad
140
118
Sibiu
99
Fagaras
80
Rimnicu Vilcea
Timisoara
111
Lugoj
97
Pitesti
211
70
Mehadia
146
101
75
138
Drobeta
120
Craiova
Neamt
87
Iasi
92
Vaslui
142
98
Hirsova
85
Urziceni
86
Bucharest
90
Giurgiu
Eforie

搜索树搜索



- 扩展树节点（寻找潜在的行动规划）
- 从搜索前沿（当前的所有叶节点）中考虑扩展；前沿代表了当前所有的规划部分
- 树节点扩展的越少越好
- 探索策略

通用的树搜索方法框架

function 树搜索(问题) **returns** 一个 解, 或 失败
    把问题的初始状态放入 搜索前沿
    loop do
        if 搜索前沿 为空 **then return** 失败
        选择一个叶 节点 并把它从 搜索前沿 中移除
        if 这个 节点 包含一个目标状态 **then return** 相应的解（从根节点到此节点的路径）
        扩展这个选择的 节点, 把扩展结果的 节点 加入 搜索前沿

- 关键元素:
  - 搜索前沿
  - 扩展操作
  - 探索(选择)策略
- 主要问题: 从哪些前沿叶节点开始探索扩展?

一个节点的具体实现

节点的属性:

state, parent, action, path-cost

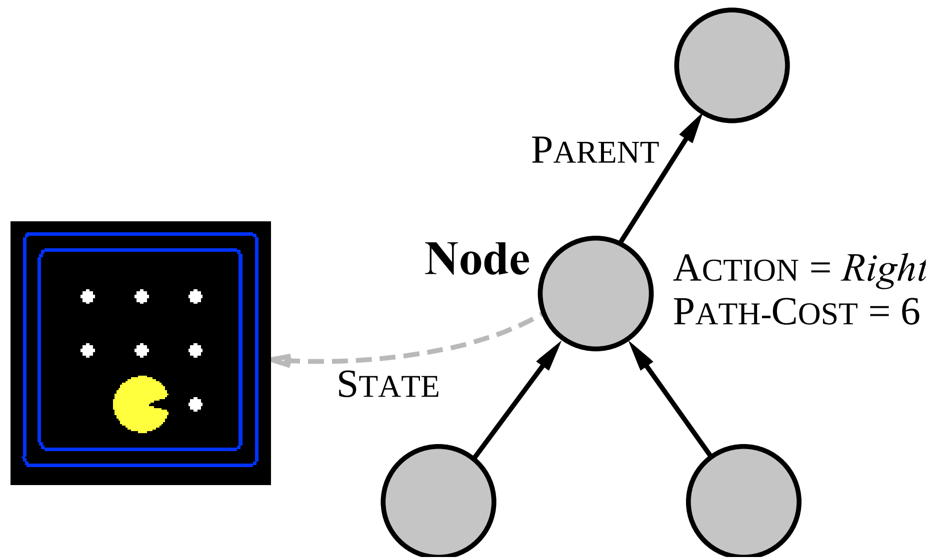
A child of node by action *a* has

state = result(node.state,*a*)

parent = node

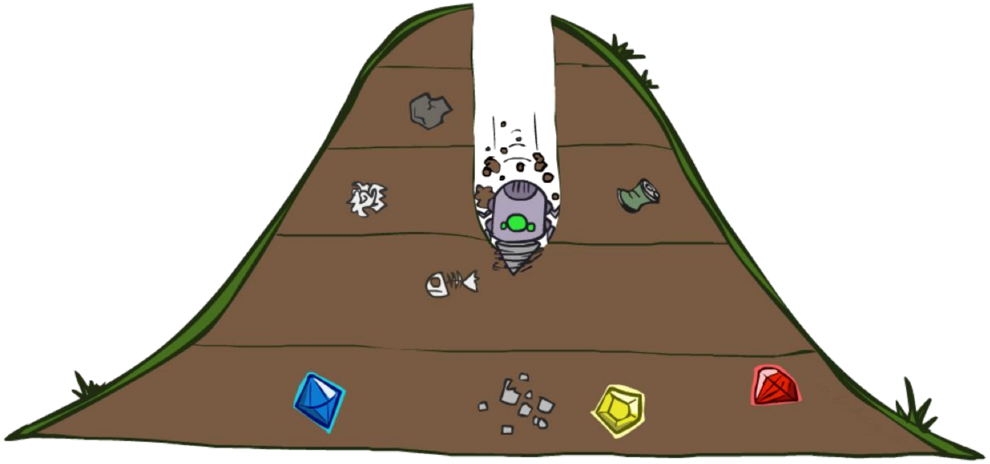
action = *a*

path-cost = node.path-cost + step-cost(node.state, *a*, self.state)



解的获取通过回溯父节点指针采集行动，从而获得一个行动序列即行动规划

深度优先搜索

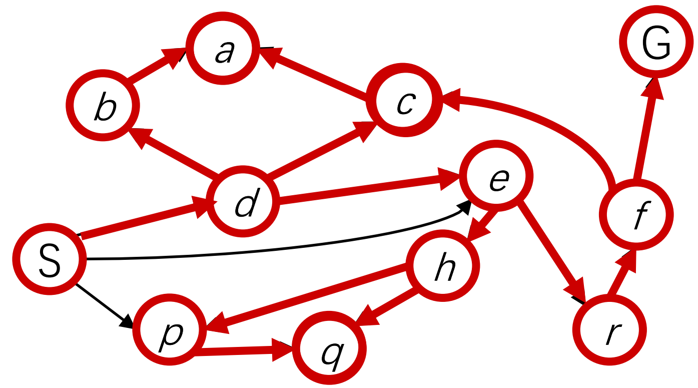


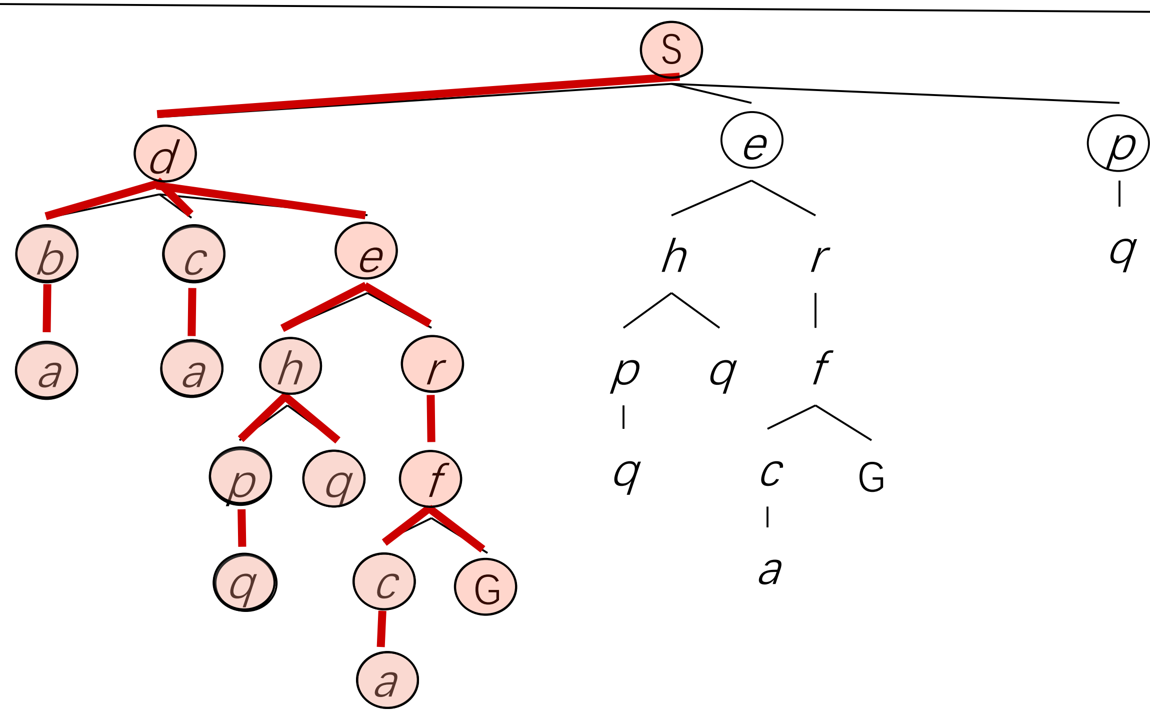
深度优先搜索

策略：总是最先扩展一个最深的节点

实现:

前沿是一个后进先出的栈

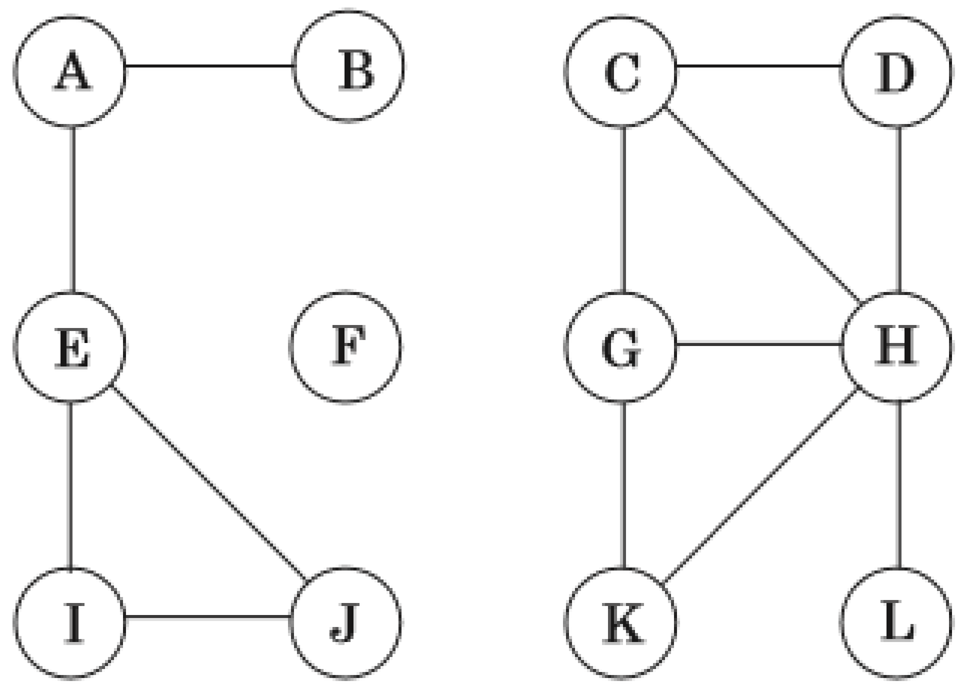


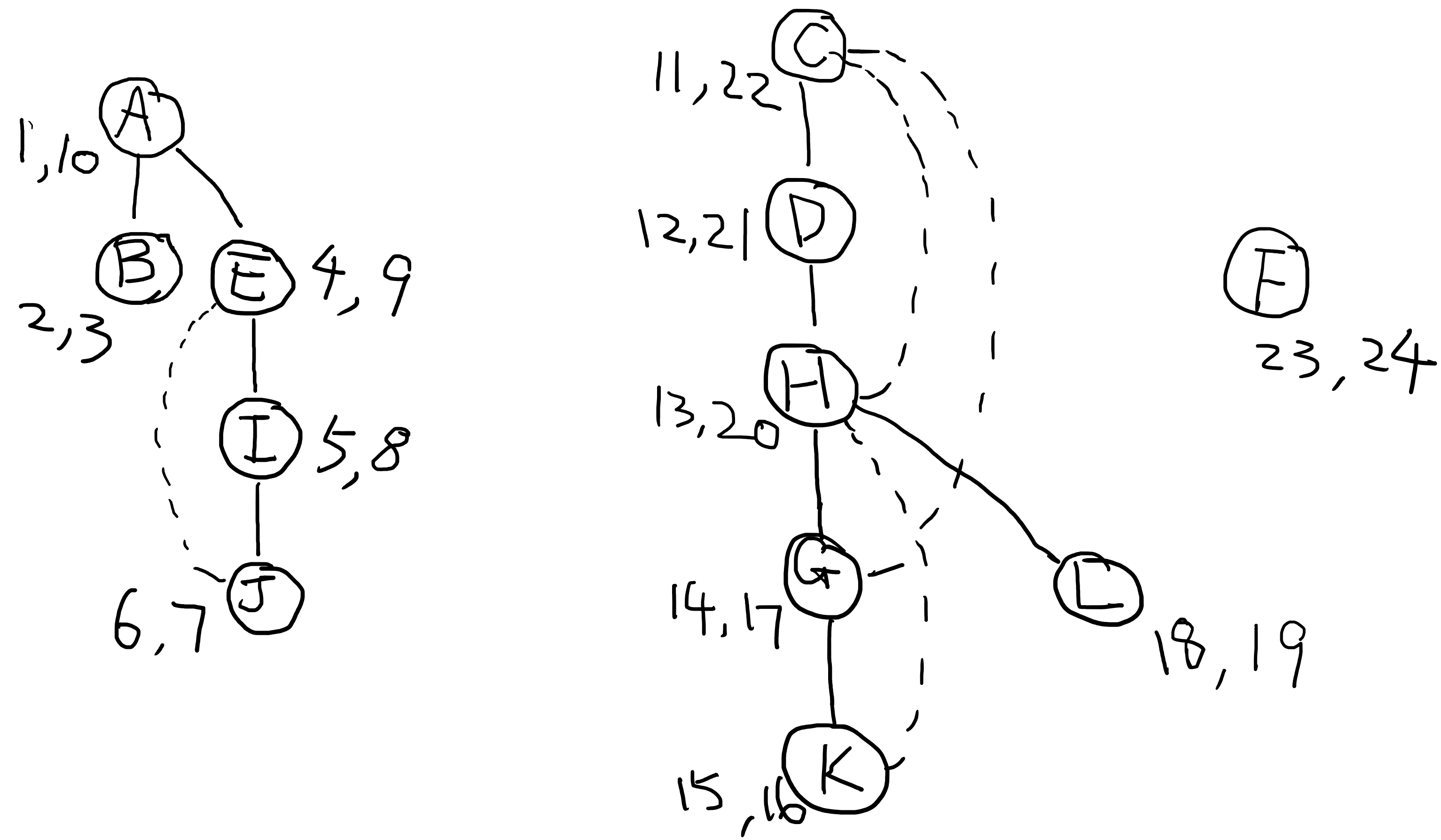


```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
   from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
   if not visited(u): explore(u)
postvisit(v)
```

```
procedure dfs(G)

for all v ∈ V:
   visited(v) = false

for all v ∈ V:
   if not visited(v): explore(v)
```

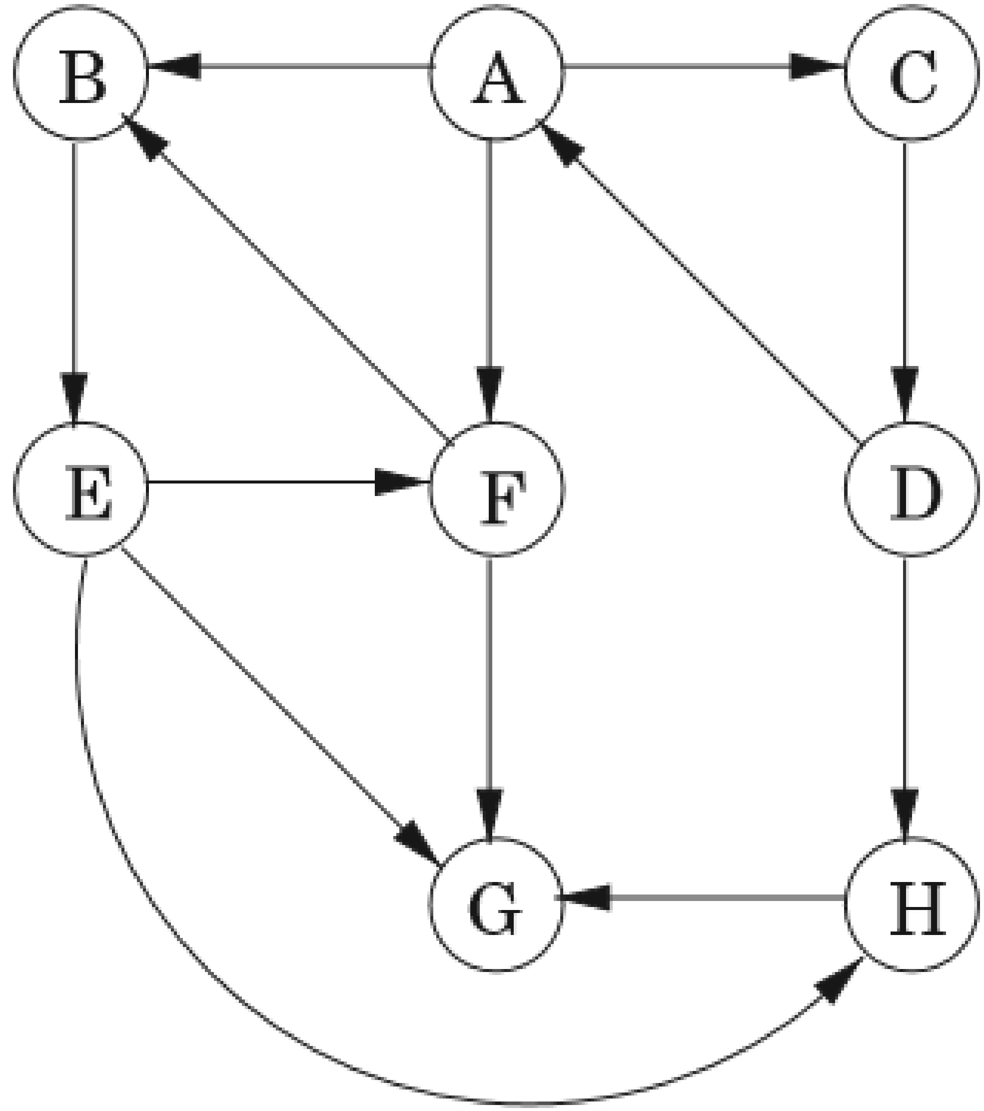
实例



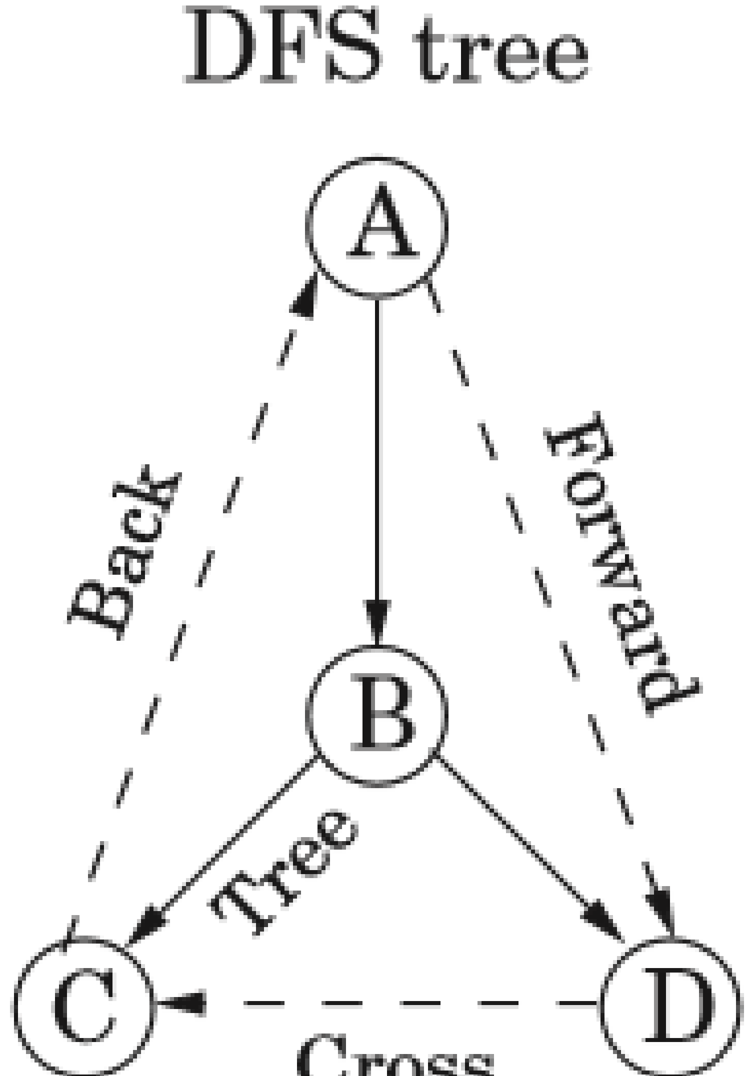


A
1,10
B
2,3
E
4,9
I
5,8
J
6,7
C
11,22
D
12,21
H
13,20
G
14,17
K
15,16
L
18,19
F
23,24

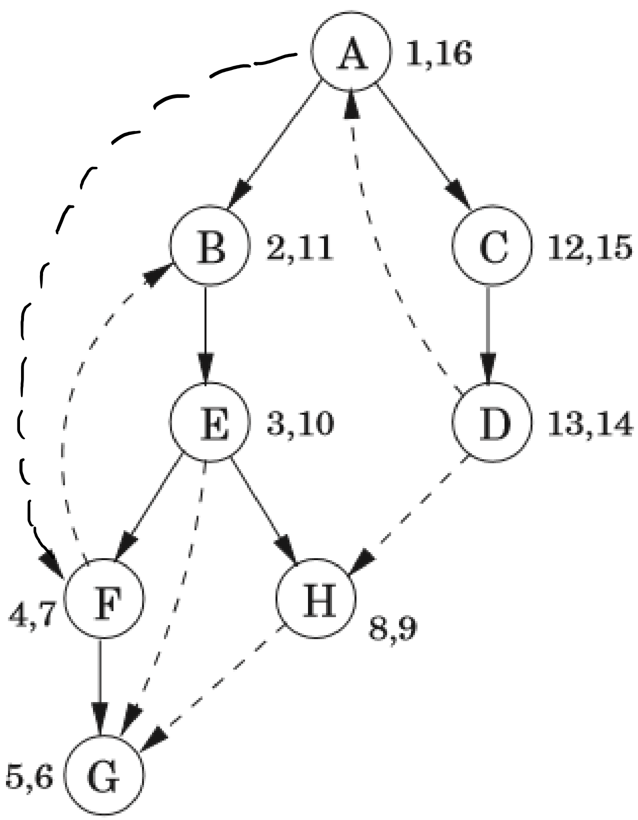
实例
(有向图)

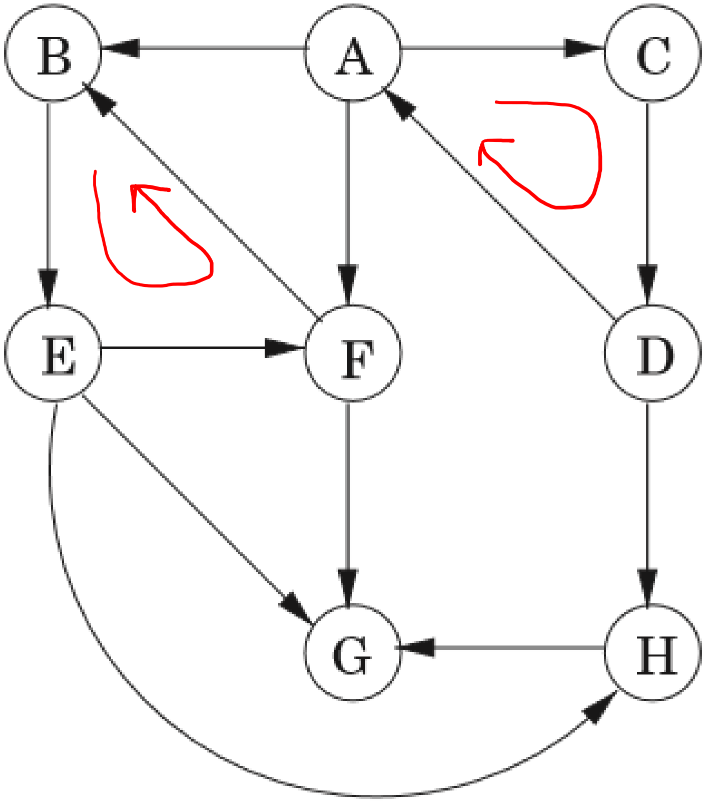


搜索树和非搜索树的边的分类



搜索树





时间复杂度

```
procedure explore(G, v)
Input: G = (V, E) is a graph; v ∈ V
Output: visited(u) is set to true for all nodes u reachable
    from v

visited(v) = true
previsit(v)
for each edge (v, u) ∈ E:
    if not visited(u): explore(u)
postvisit(v)
```

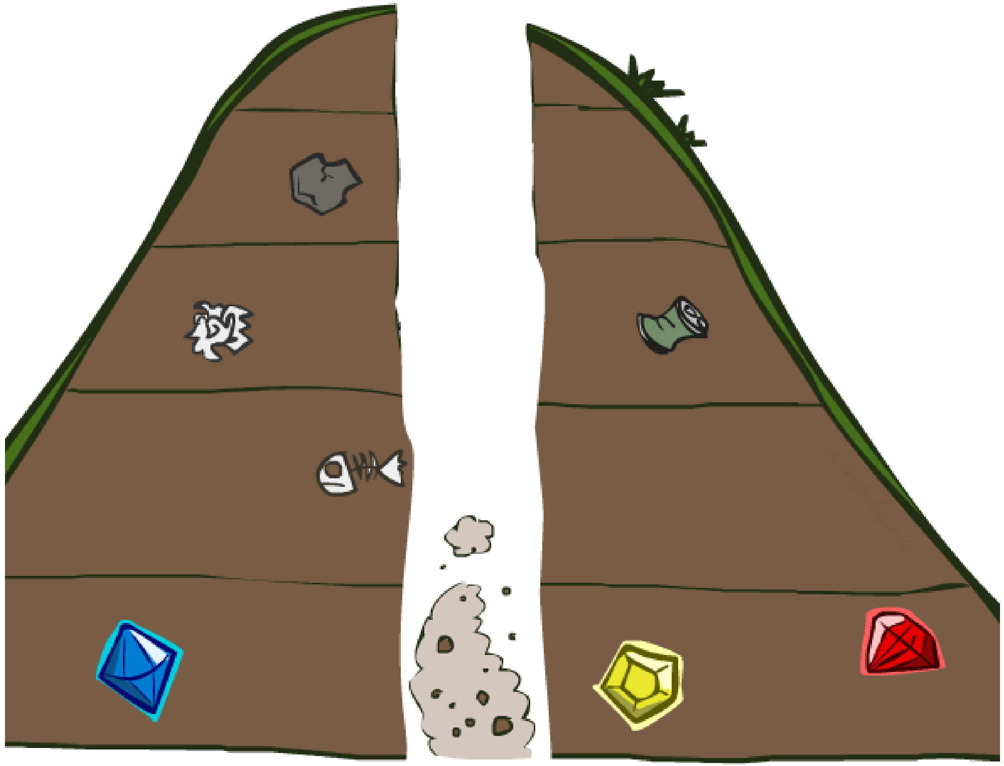
← 固定量工作

← $O(|E|)$

```
procedure dfs(G)

for all v ∈ V:
    visited(v) = false

for all v ∈ V:
    if not visited(v): explore(v)
```

每个 v 都会被 explored 一次 →

搜索算法的属性

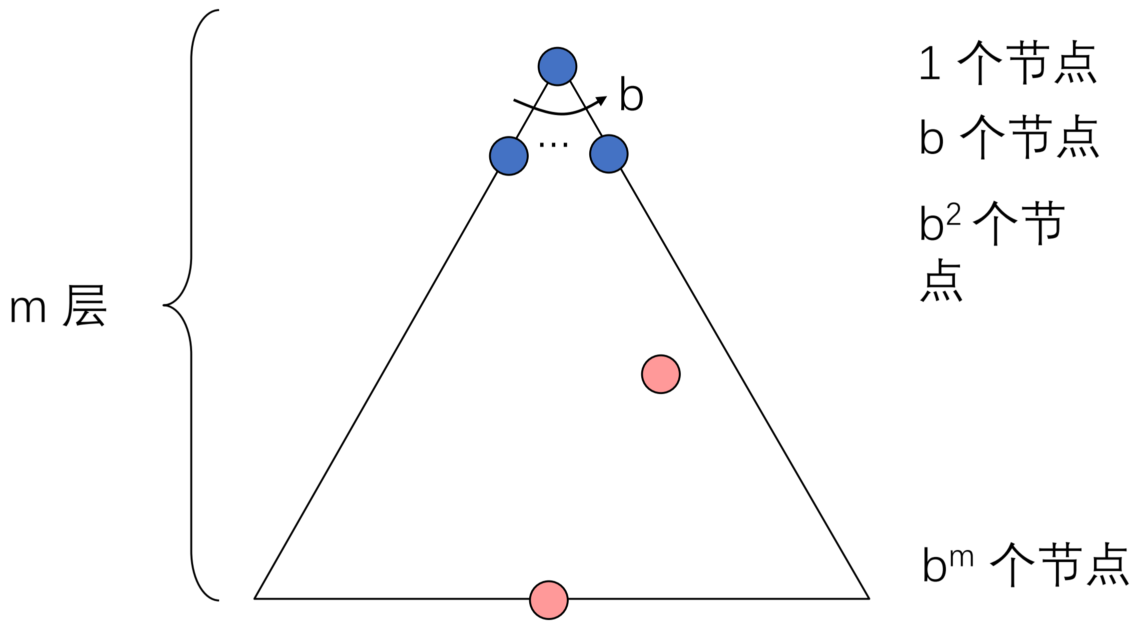


搜索算法的属性

- 完全性：保证能找到一个存在的解？
- 最优性：保证能找到最小路径成本的解？
- 时间复杂性？
- 空间复杂性？

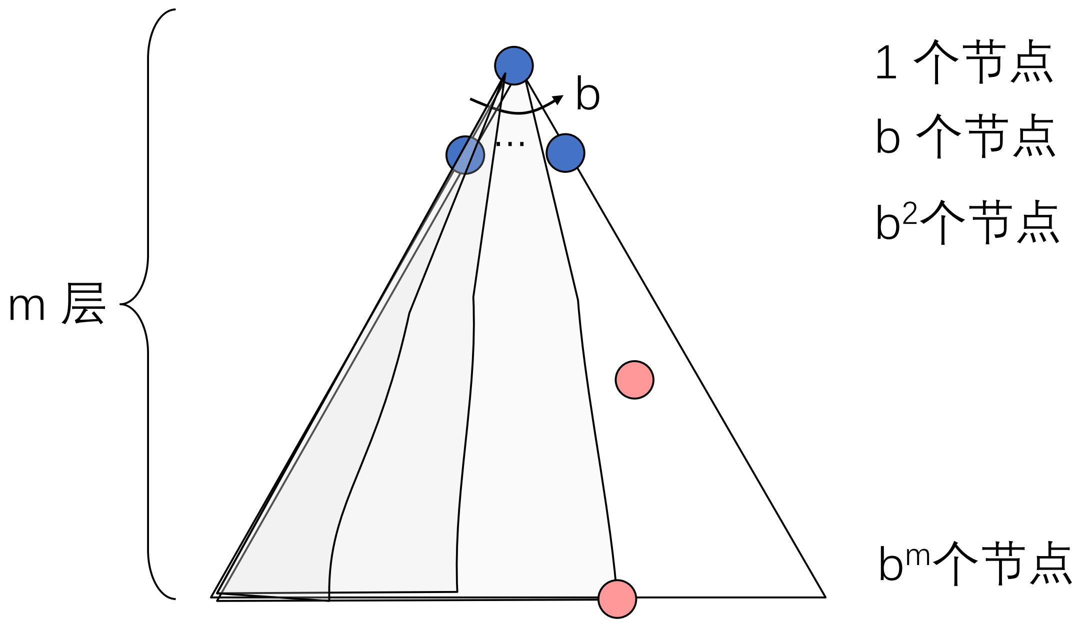
- 搜索树：
  - b ： 最大分支数
  - m： 最大深度
  - 解可能存在于不同深度

- 整个树的节点总数？
  - $1 + b + b^2 + \cdots. b^m = O(b^m)$

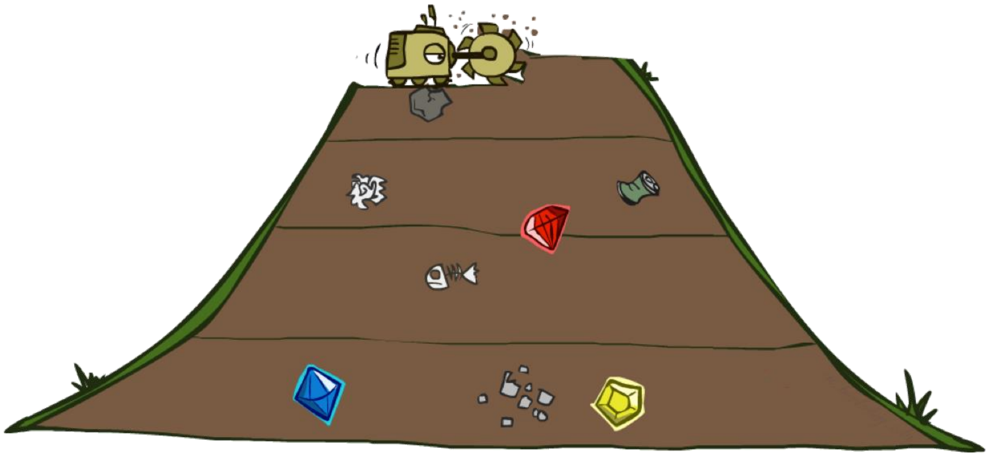


深度优先搜索（DFS）属性

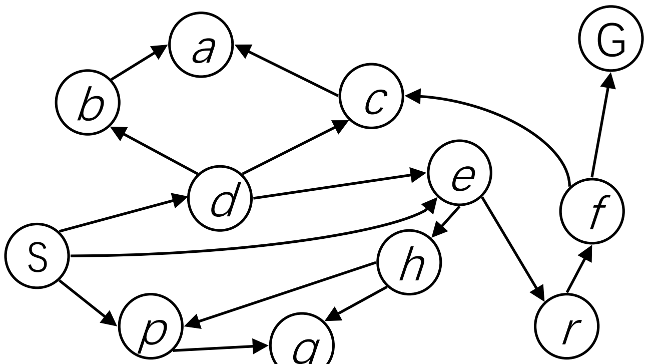
- 哪些节点被扩展?
  - 某些左边的节点
  - 可以产生整个树
  - 如果 m 是有限的，时间复杂度O(b^m)
- 存储前沿需多少空间?
  - **只存储一条路径从根到一个叶节点及沿路相关兄弟节点，所以 O(bm)**
- 完全性?
  - 不一定，因为m 可能是无穷。除非可以避免循环搜索
- 优化性?
  - 不优化。发现的可能是树最左边的一个次优解



广度优先搜索（BFS）



广度优先（Breadth-FS）搜索

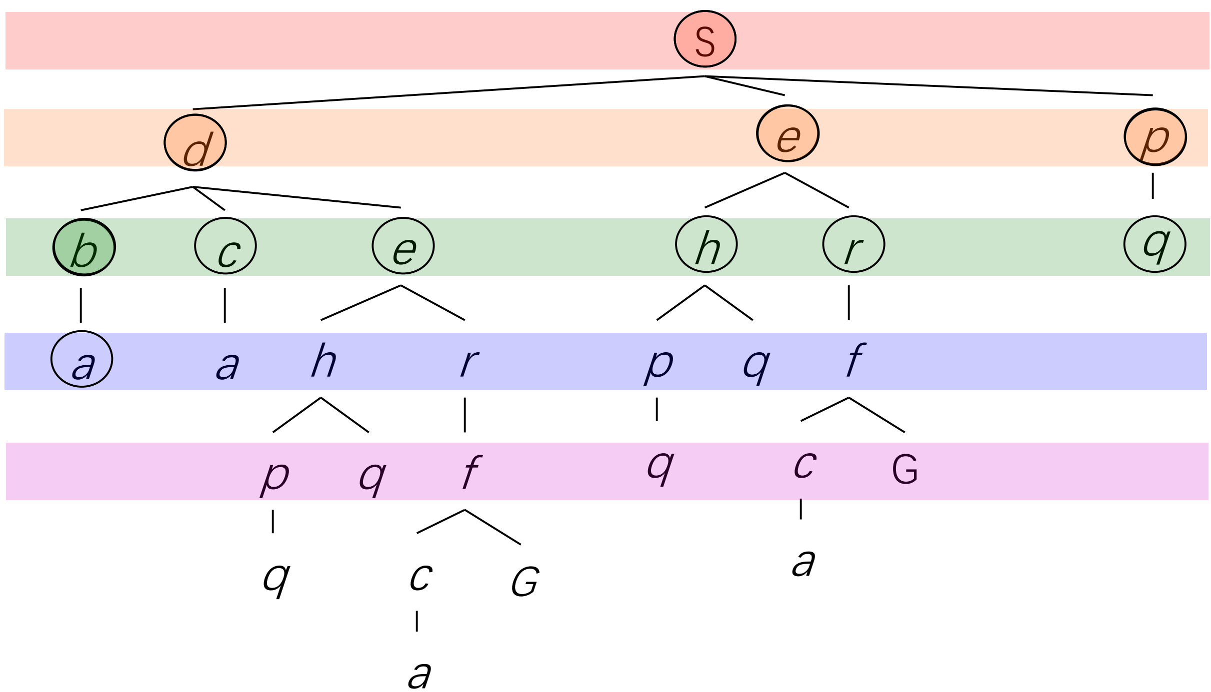


策略：优先扩展最浅的节点

实现:

先进先出的队列

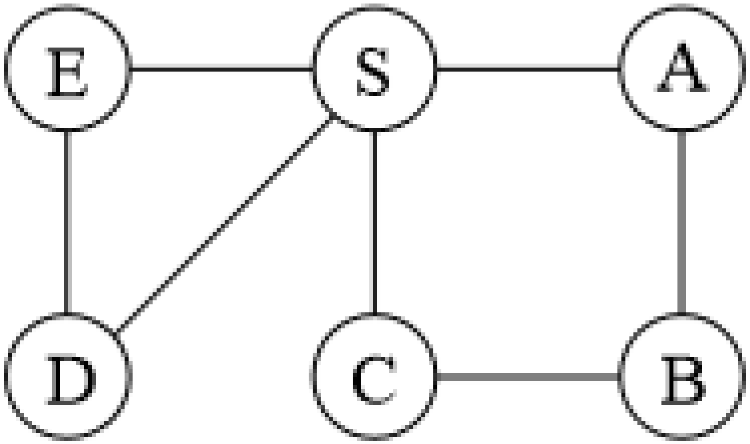
搜索层次



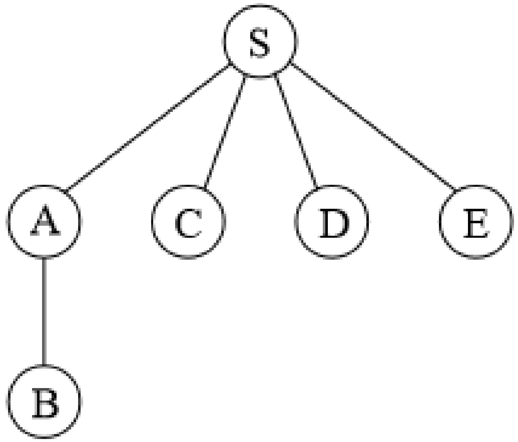
算法

```
procedure bfs(G, s)
Input:  Graph G = (V, E), directed or undirected; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
   u = eject(Q)
   for all edges (u, v) ∈ E:
      if dist(v) = ∞:
         inject(Q, v)
         dist(v) = dist(u) + 1
```

举例



Order of visitation	Queue contents after processing node
	$[S]$
S	$[A\ C\ D\ E]$
A	$[C\ D\ E\ B]$
C	$[D\ E\ B]$
D	$[E\ B]$
E	$[B]$
B	$[\,]$



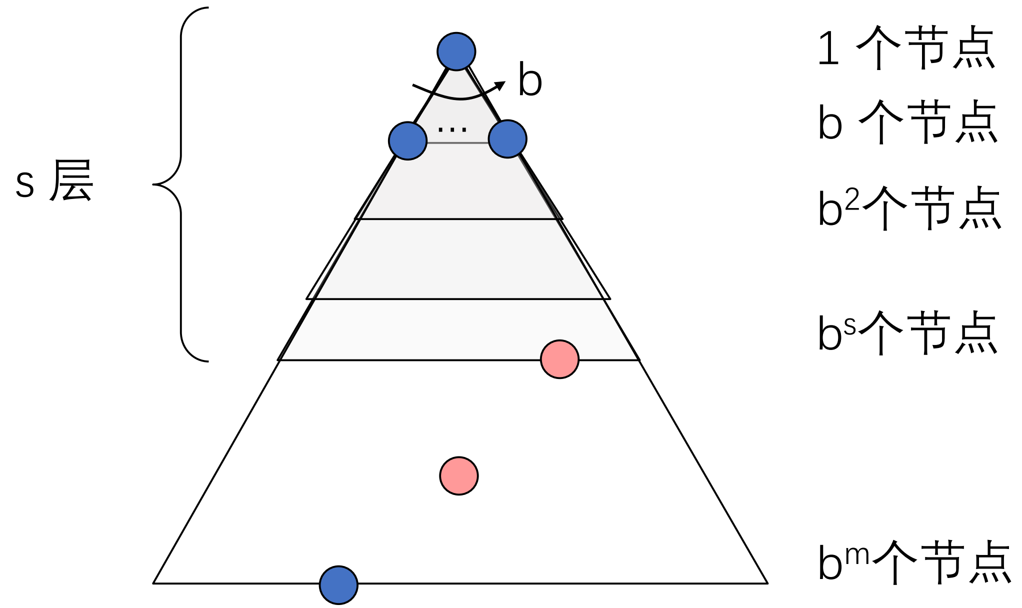
时间复杂度

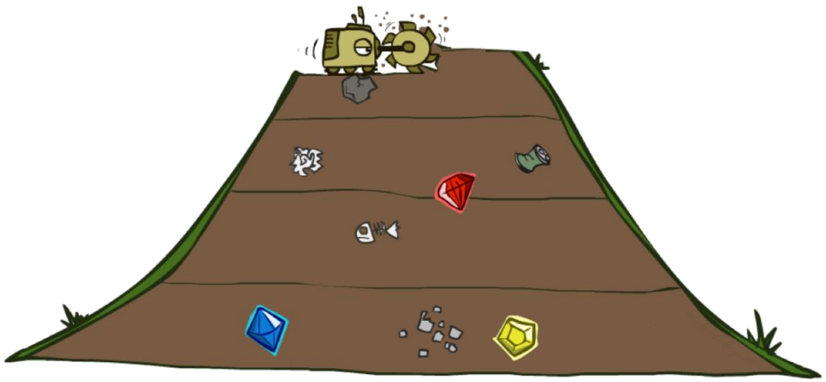
```
procedure bfs(G, s)
Input:  Graph G = (V, E), directed or undirected; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞

dist(s) = 0
Q = [s] (queue containing just s)
while Q is not empty:
    u = eject(Q)
    for all edges (u, v) ∈ E:
        if dist(v) = ∞:
            inject(Q, v)
            dist(v) = dist(u) + 1
```

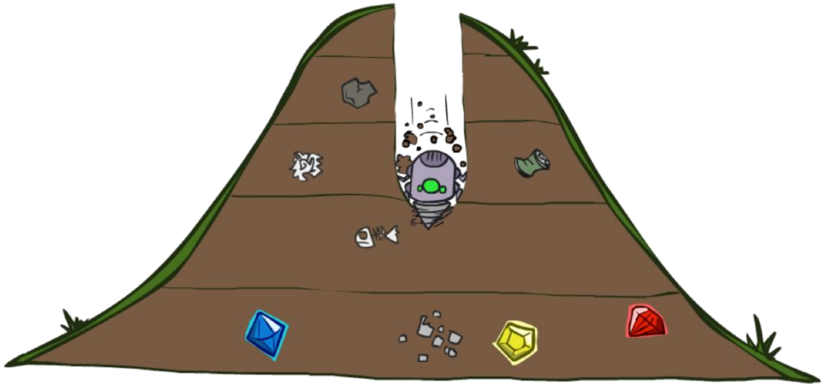
$O(|V|+|E|)$

广度优先搜索属性

- 哪些节点被扩展?
  - 处理所有最浅层解以上的所有节点
  - 如果最浅解的深度为 s
  - 搜索时间 $O(b^s)$
- 前沿探索需要的存储空间
  - 大概是最后一层，所以 $O(b^s)$
- 完全性?
  - 如果一个解存在，s 一定是有限的，所以是完全的!
- 优化性?
  - 是，如果所有步骤成本都是1时





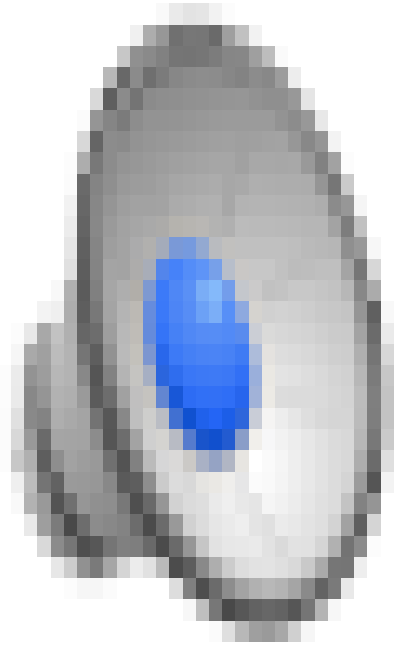


问题： DFS vs BFS

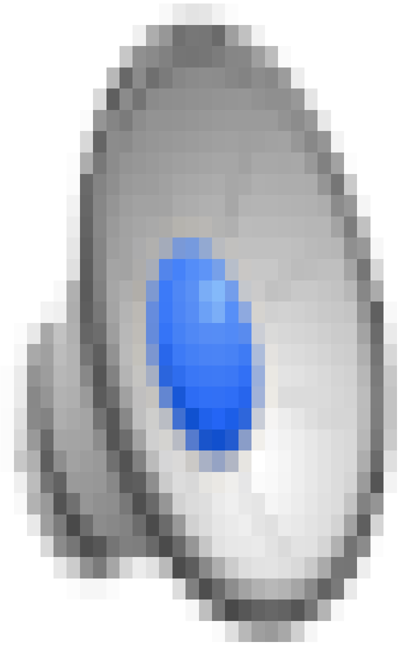
问题：深度优先搜索 vs 广度优先搜索

- 什么时候？广度优先搜索 优于 深度优先搜索
- 什么时候？深度优先搜索 优于 广度优先搜索

演示视屏：迷宫 DFS/BFS （1）

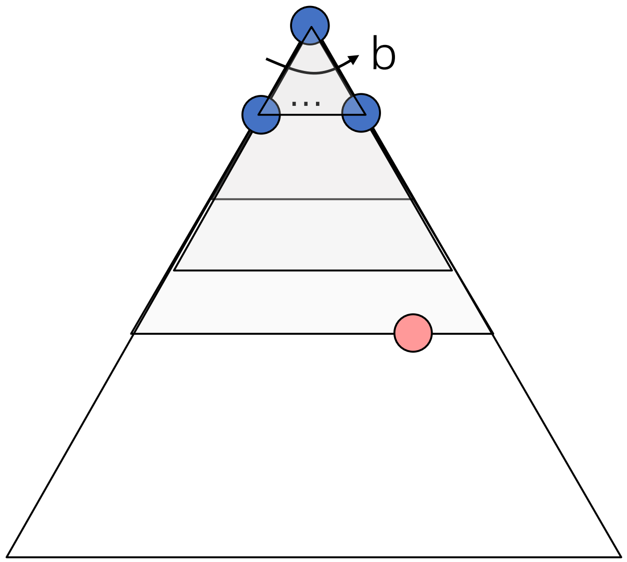


演示视屏：迷宫 DFS/BFS （2）

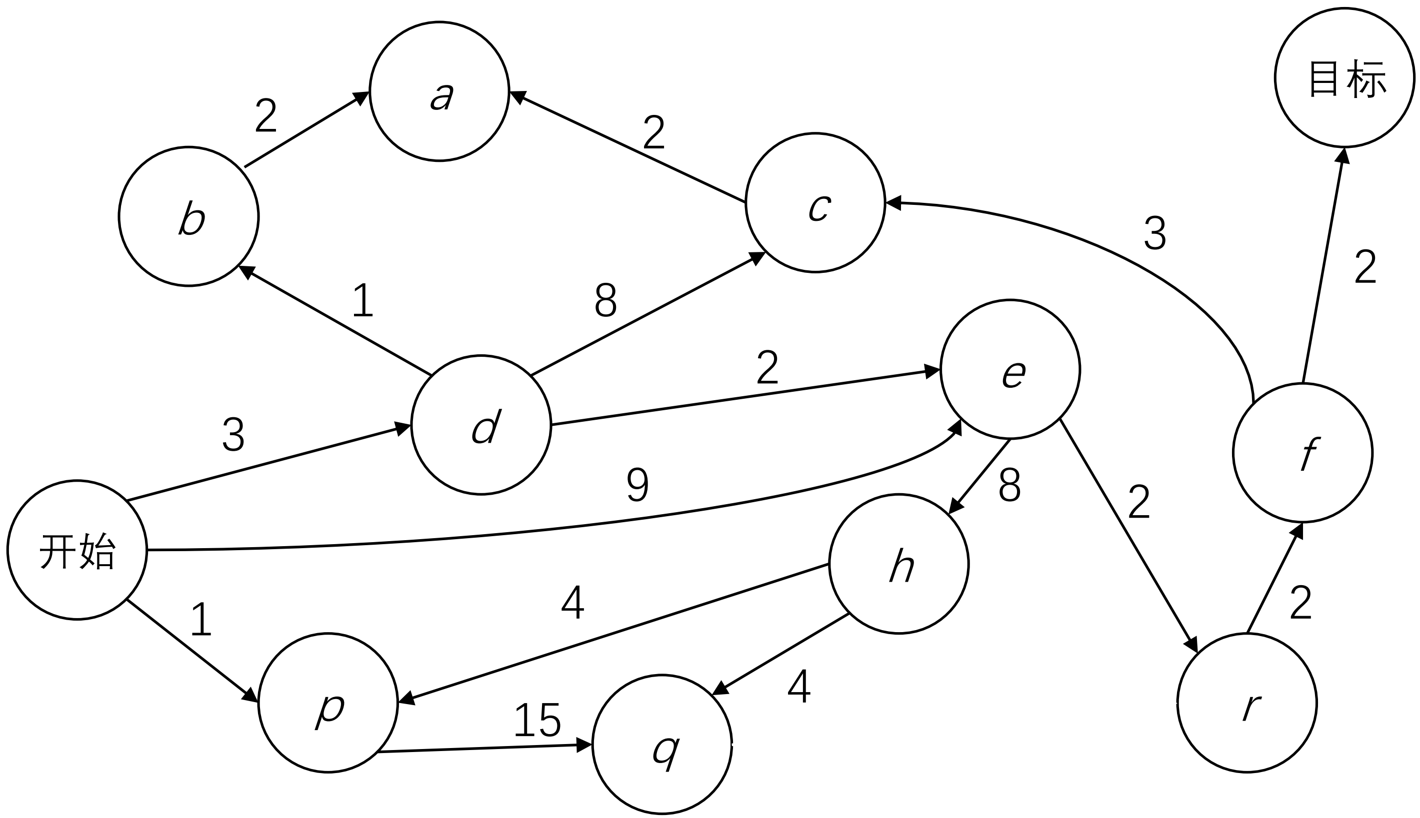


迭代加深

- 想法: 结合DFS的空间优势， 和BFS的时间/搜寻浅解的优势
  - 运行一次深度限制为1的DFS,如果没有解， …
  - 运行一次深度限制为2的DFS,如果没有解, 继续
  - 运行一次深度限制为3的DFS,如果没有解， …
- 难道重复搜索不浪费吗?
  - 多数重复搜索集中在浅层，节点数相对少，所以还可以。

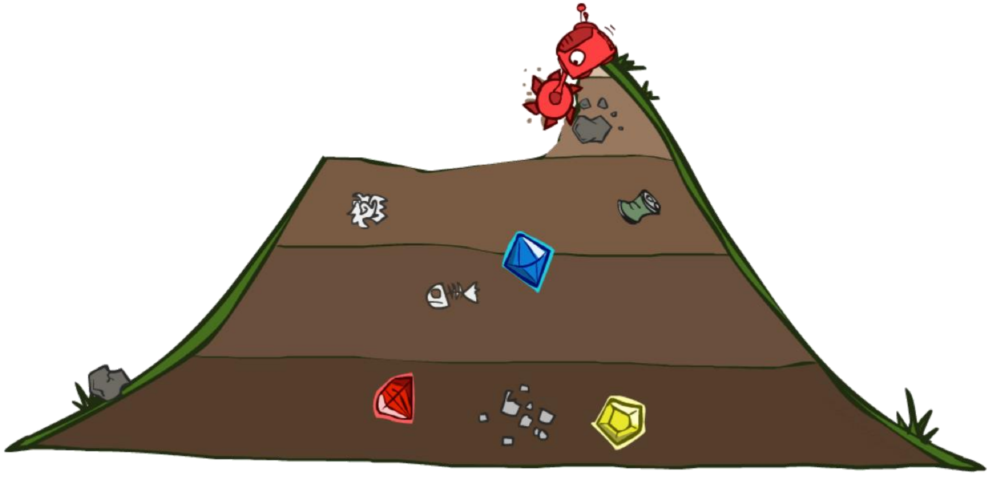


寻找最小步骤成本路径



- BFS找到的是最少行动数量的路径，而不是最小步骤成本路径。

基于成本的统一搜索（Uniform Cost Search）

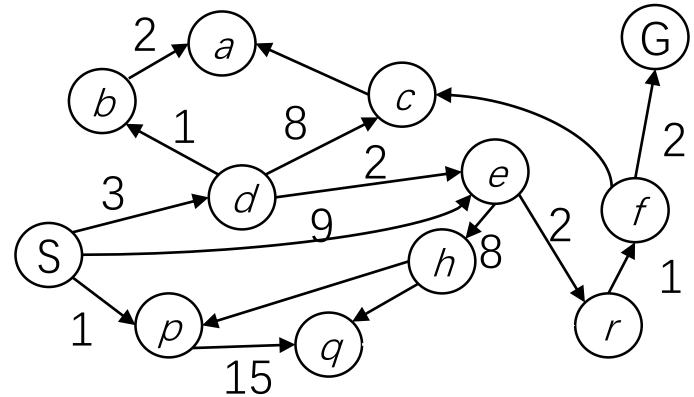


基于成本的统一搜索

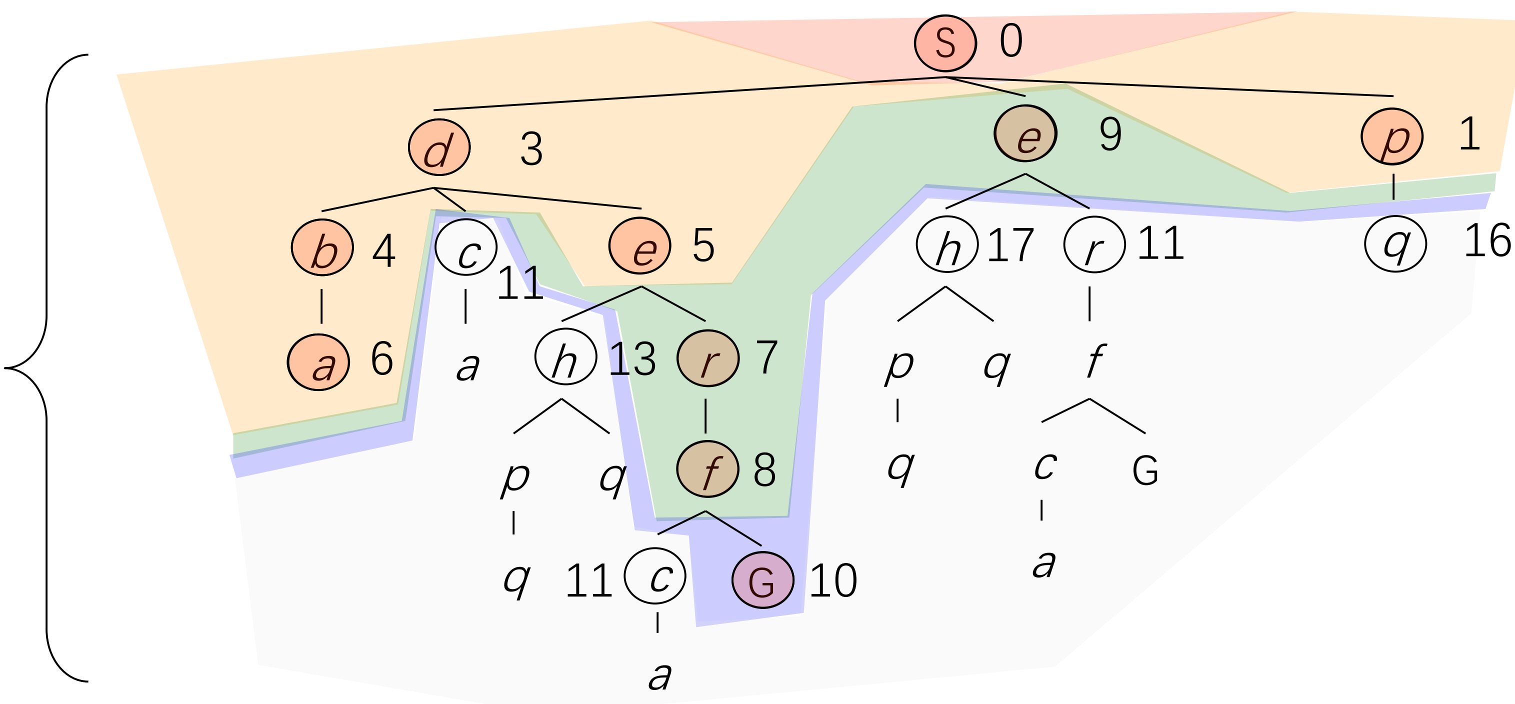
策略：首先扩展一个成本最低的节点

前沿用优先级队列存储

(优先级：累计成本)

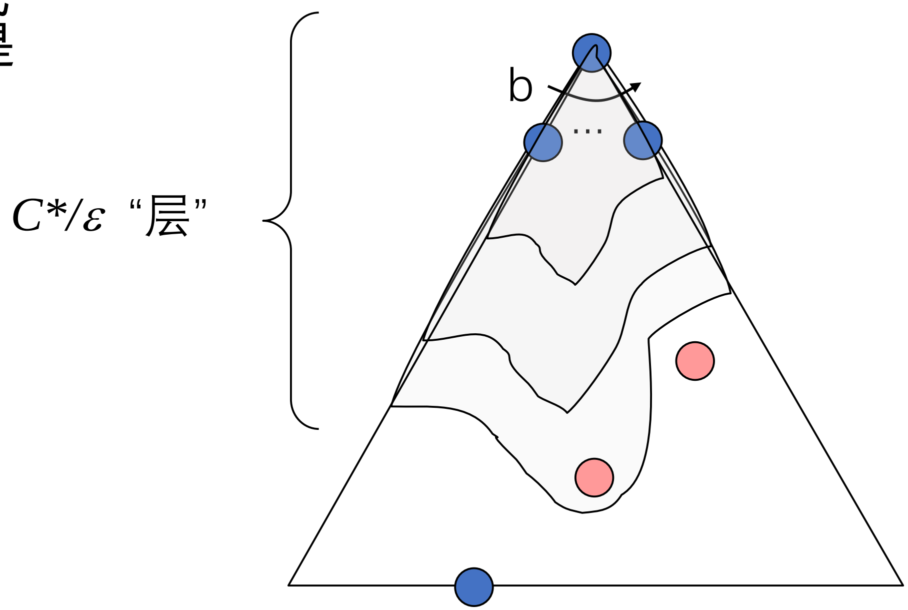


成本
轮廓



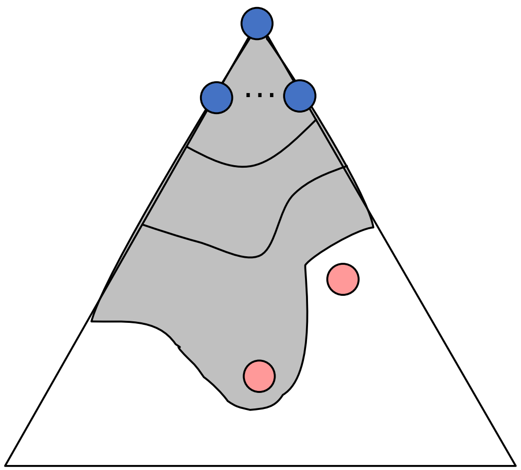
基于成本的统一搜索（UCS）属性

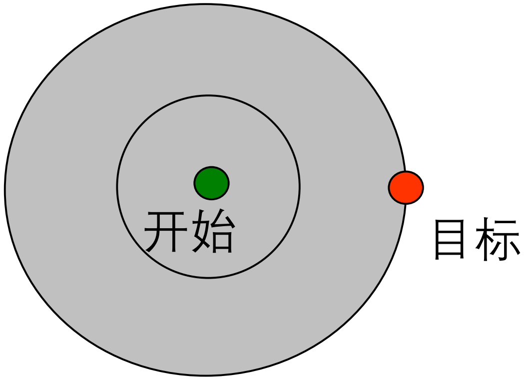
- 哪些节点被扩展?
  - 处理所有累计成本小于最低成本解的所有节点
  - **如果那个解的成本是 C*， 并且步骤成本至少是 ε，那么其“有效深度” 大概是** C*/ε
  - 时间花费 O($b^{C^*/\varepsilon}$) (有效深度指数)
- 前沿节点占用多少空间?
  - 大概总是上一层, 所以是 O($b^{C^*/\varepsilon}$)
- 完全性?
  - 假设最优解的成本有限，步骤代价都为正数，则是!
- 优化性?
  - 是。



存在不足

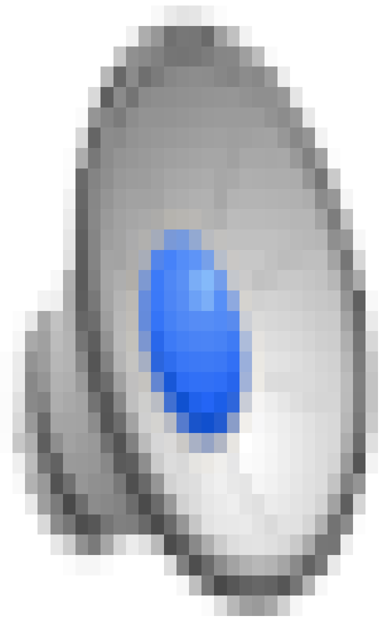
- 探索逐步提高的成本轮廓范围内的节点
- 优势: 完全性和优化性
- 不足:
- 在每个“方向”上都探索
- 没有关于目标位置的信息
- 可以补救（以后会讲）



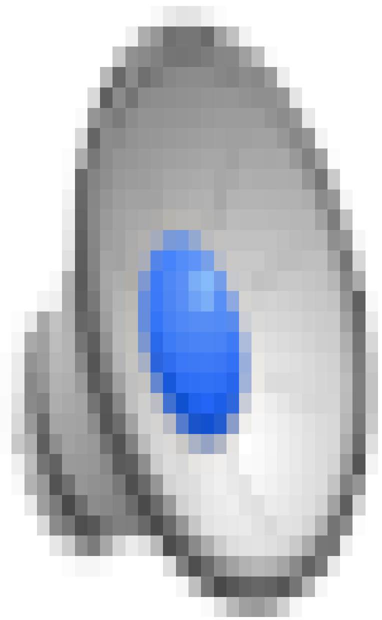


演示视频： 空白UCS

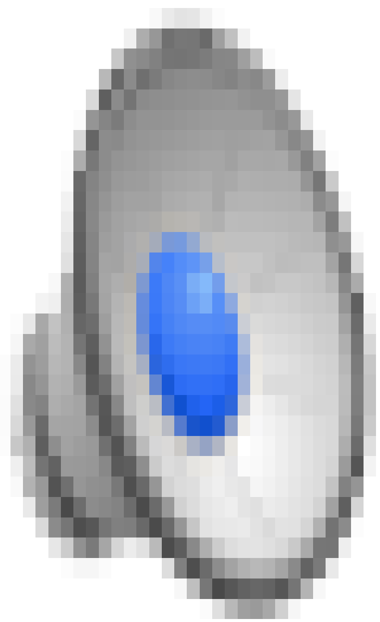
演示视频：有深/浅水的迷宫 --- DFS, BFS, or UCS？(1)



演示视频： 有深/浅水的迷宫 --- DFS, BFS, or UCS？ (2)



演示视频：有深/浅水的迷宫 --- DFS, BFS, or UCS？(3)



Dijkstra's Shortest-path Algorithm

```
procedure dijkstra(G, l, s)
Input:  Graph G = (V, E), directed or undirected;
        positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
   dist(u) = ∞
   prev(u) = nil
dist(s) = 0


H = makequeue (V)  (using dist-values as keys)
while H is not empty:
   u = deletemin(H)
   for all edges (u, v) ∈ E:
       if dist(v) > dist(u) + l(u, v):
          dist(v) = dist(u) + l(u, v)
          prev(v) = u
          decreasekey(H, v)
```

时间复杂度分析

```
procedure dijkstra(G, l, s)
Input:  Graph G = (V, E), directed or undirected;
        positive edge lengths {l_e : e ∈ E}; vertex s ∈ V
Output: For all vertices u reachable from s, dist(u) is set
        to the distance from s to u.

for all u ∈ V:
    dist(u) = ∞
    prev(u) = nil
dist(s) = 0

H = makequeue(V)  (using dist-values as keys)
while H is not empty:
    u = deletemin(H)
    for all edges (u, v) ∈ E:
        if dist(v) > dist(u) + l(u, v):
            dist(v) = dist(u) + l(u, v)
            prev(v) = u
            decreasekey(H, v)
```

$|V| = n, |E| = m$

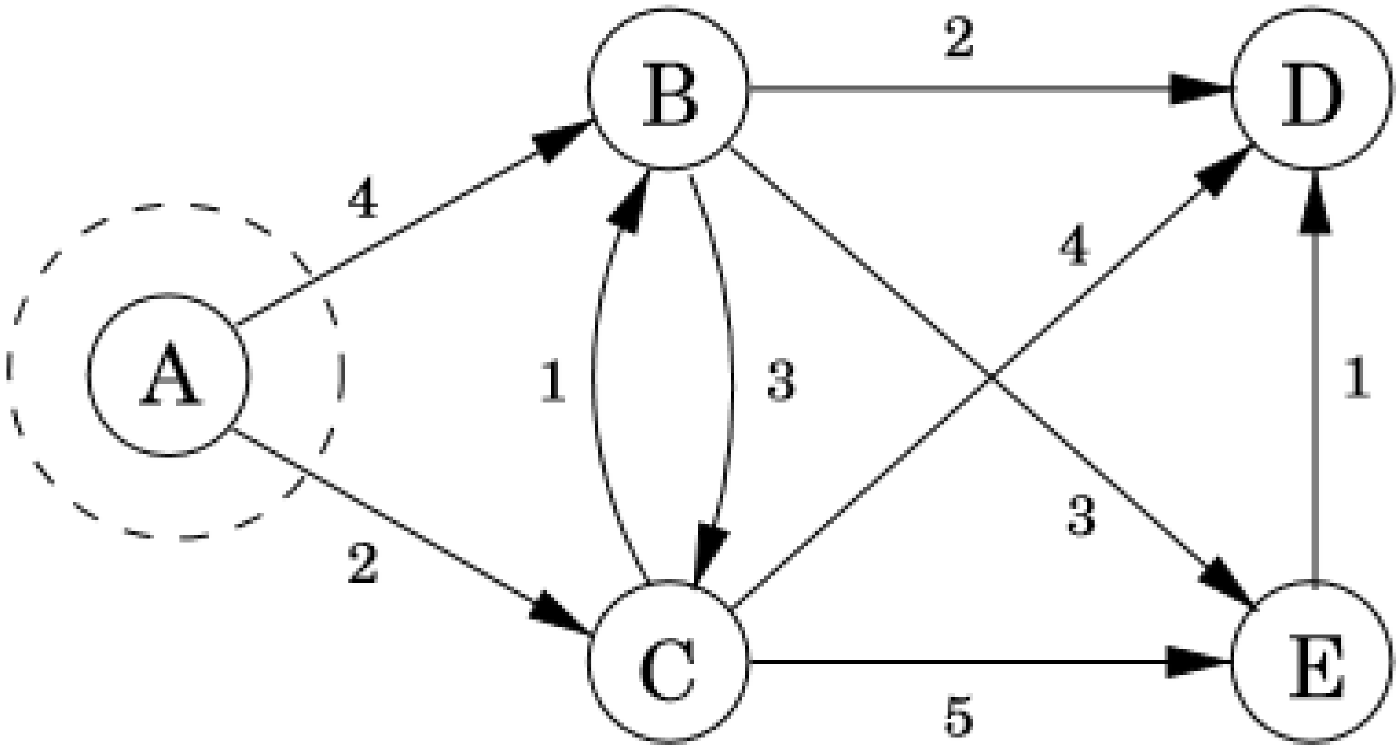
$O(n)$ (for all $u \in V$)

$O(n)$ (makequeue)

$O(1)$ (deletemin)

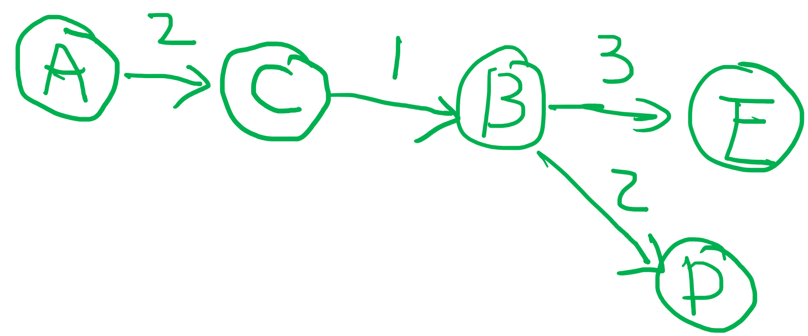
$O(\log n)$ (decreasekey)

Example



step:

dist(·)	0	1	2	3	4	5
A	0	0	0	0	0	0
B	∞	4	3	3	3	3
C	∞	2	2	2	2	2
D	∞	∞	6	5	5	5
E	∞	∞	7	6	6	6
deletemin	A	C	B	D	E	ϕ



最短路径树(从A开始的)

Dijkstra's Algorithm (Conceptual view)

Dijkstra's Algorithm:

Input: Graph G, with each edge e having a length $len(e)$, and a start node s.

Initialize: tree $= \{s\}$, no edges. Label s as having distance 0 to itself.

Invariant: nodes in the tree are labeled with the correct distance to s. 保持不变的.

Repeat:

1. For each neighbor x of the tree, compute an (over)-estimate ? of its distance to s:

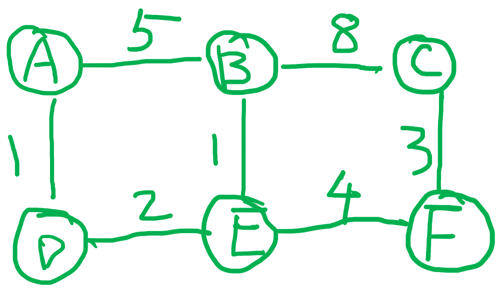
$$\text{distance}(x) = \min_{e=(v,x):v\in\text{tree}} [\text{distance}(v) + len(e)] \quad (1)$$

$len(e) \geq 0$?

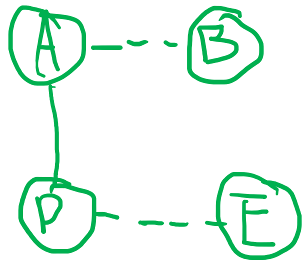
In other words, by our invariant, this is the length of the shortest path to x whose only edge not in the tree is the very last edge.

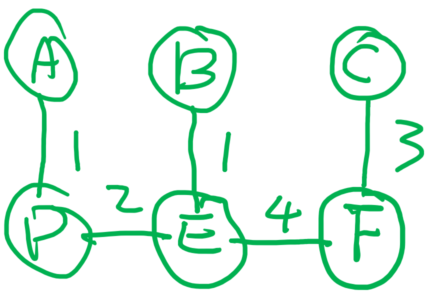
2. Insert the node x of minimum distance into tree, connecting it via the argmin (the edge e used to get distance(x) in the expression (1)).

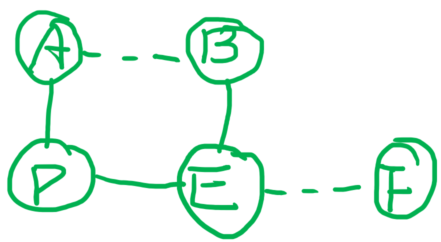
greedy



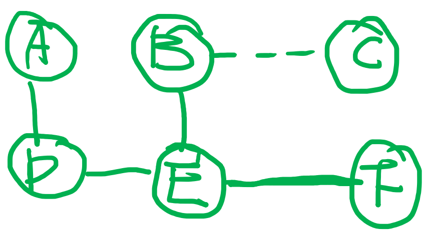
dist	A	B	C	D	E	F
	0	∞	∞	∞	∞	∞
	0	5	∞	1	∞	∞
	0	5	∞	1	3	∞
	0	4	∞	1	3	7
	0	4	12	1	3	7
	0	4	10	1	3	7



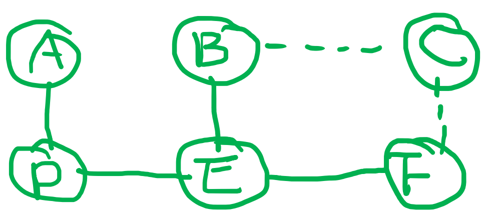




$\min\begin{cases} 5 \leftarrow A-B \\ 4 \leftarrow E-B \end{cases}$ $\quad \arg\min = (E, B)$



B−C E−F

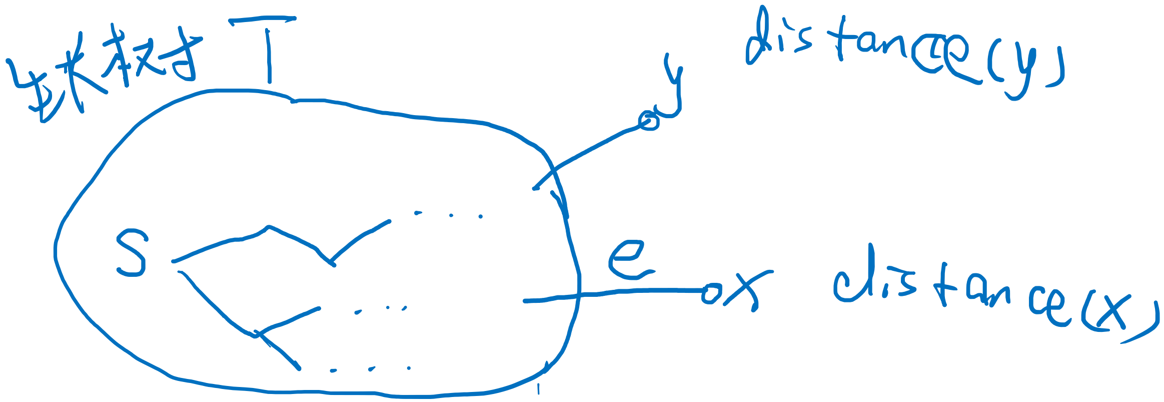


$\min\begin{cases} B-C : 4+8=12 \\ F-C : 7+3=10 \checkmark \end{cases}$

$\arg\min = (F, C)$

为什么Dijkstra’s algorithm 生成树是一棵最短路径树？

Correctness Proof



P：表示从 s 到 x 的最短路径

e 是算法选择的边加入到 T

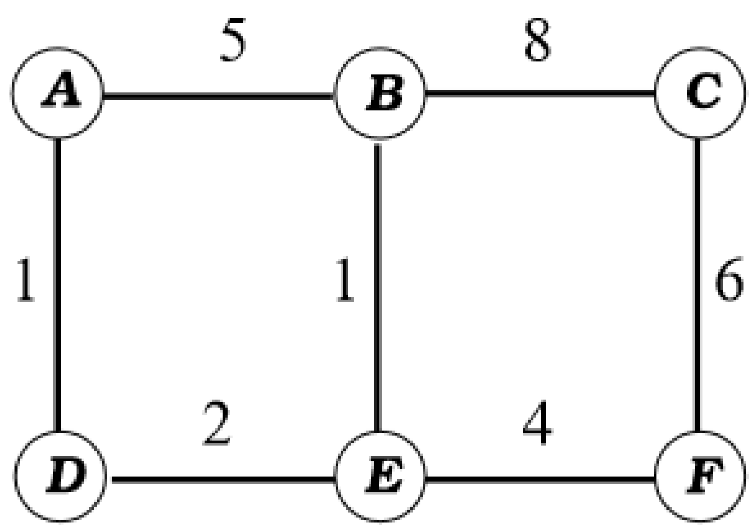
要证明：P 的最后一条边是 e.

反证法：假设 y 是在 P 上，但不在 T 里，则

length(P) > distance(y) > distance(x)，矛盾！

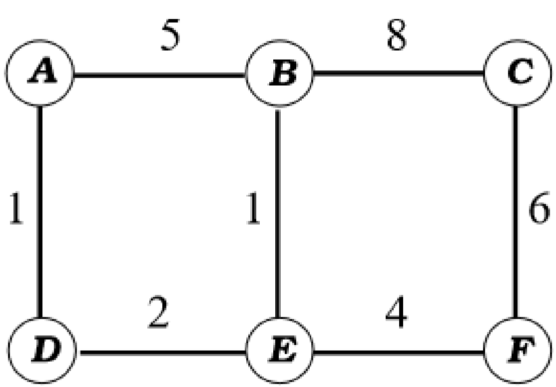
最小生成树

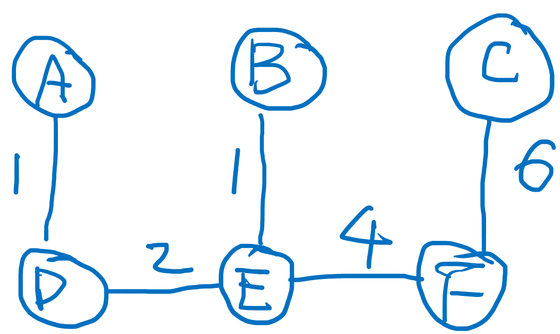
- 什么是生成树?
- 什么是最小生成树?



(Dijkstra-)Prim's algorithm

1. Pick some arbitrary start node s. Initialize tree $T = \{s\}$.
2. Repeatedly add the shortest edge incident to T (the shortest edge having one vertex in T and one vertex not in T) until the tree spans all the nodes.





Prim's 算法能够正确找到一棵最小生成树?

归纳法证明。

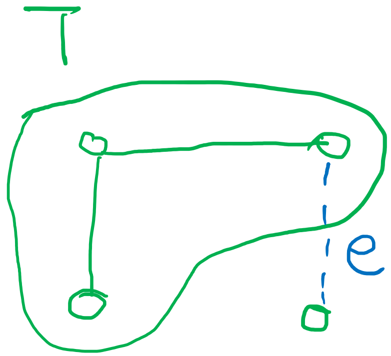
假设：算法构建中的生成树T是图G某个最小生成树M的子树。

- 开始时，T只包含开始节点，假设自然成立。
- 在算法执行中，边e被算法选出来，

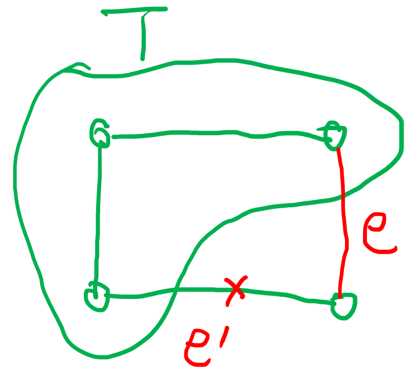
  现在需要证明的是加入e后的新树也满足假设。

if $e \in M$, then $T \cup \{e\}$ 满足假设.

if $e \notin M$, 添加 e 到 M 中, 产生环路



M 是一个最小生成树.



$len(e) < len(e')$

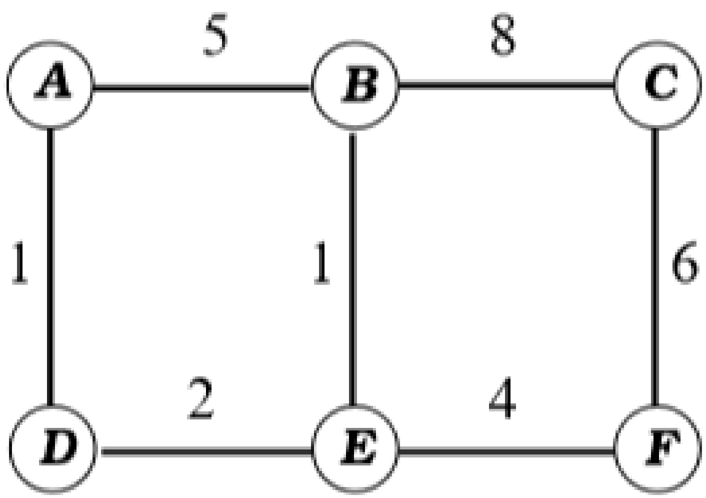
算法选择 e, 而不是 e'

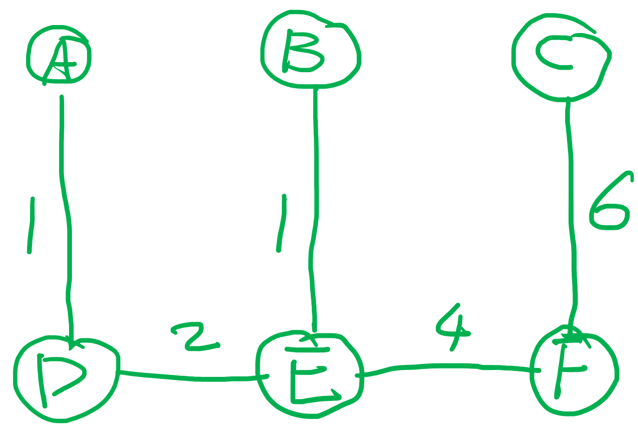
所以, 在 M 中增添 e, 去掉 e', 结果仍是一棵生成树 M', 而且 $size(M') \leq size(M)$

而且 M' 包含 $T \cup \{e\}$, 假设成立.

归纳证明完成。

1. 排序所有边，根据边长从小到大。
2. 初始森林是在图中去除所有边，仅保留所有节点。
3. 按照边长从小到大的顺序，把边加入到森林里，合并树并使得其不产生环路，重复直至该森林中的所有树合并为一棵树为止。





Kruskal’s Algorithm

Kruskal’s Algorithm:

Sort edges by length and examine them from shortest to longest. Put each edge into the current forest (a forest is just a set of trees) if it doesn’t form a cycle with the edges chosen so far.

Kruskal's 算法能够正确找到一棵最小生成树。

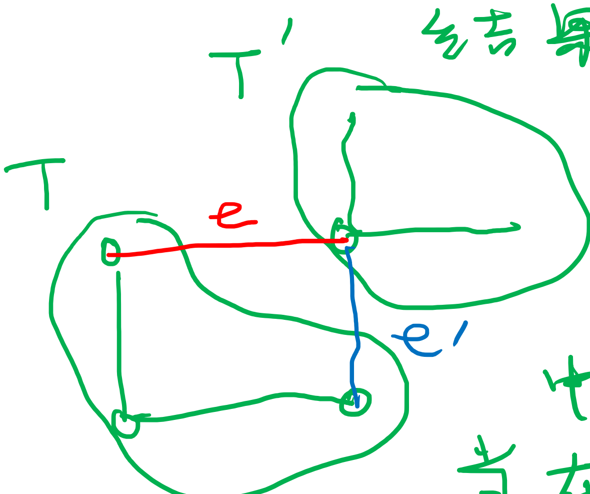
为什么？

贪婪算法，始终维持一个不变的事实：森林F与图G中的某个MST是一致的，即F中的边都在这个MST里。

我们需要证明以上这个假设成立。

归纳证明法。在初始F时，该假设自然是成立。

归纳过程中，令M是一棵MST，此时F与其一致，此时算法选择边e边加入F，如果$e \in M$，结果自然得证，否则：



此时加入e到M中必然形成一个环路，同时必然可以找到边e'连接同样的两棵树T和T'，由算法中的操作可知，$len(e) \leq len(e')$，所以当在M中添加e，并去掉e'后的结果仍是一棵MST（M'），而且其包含$F \cup \{e\}$，假设得证。

$size(M') \leq size(M)$

运行时间分析

- 1. 对所有边进行排序，根据边长
- 2. 对于要加入的每一条边需测试是否它连接的是两个不同的树
  - 如何判断一个边的两个顶点是否在同一个连接图（树）里?

"Union-Find" data structure, low-order to the sorting step.

for m tests, and n-1 mergers of components:

$O(m \lg^* n)$

$\lg^* n$ —— $\log_2$ 执行的次数，直到 n 变为 1。

运行时间

$$
\begin{aligned}
\lg^*(2) &= 1 \\
\lg^*(2^2 = 4) &= 2 \\
\lg^*(2^4 = 16) &= 3 \\
\lg^*(2^{16} = 65536) &= 4 \\
\lg^*(2^{65536}) &= 5.
\end{aligned}
$$