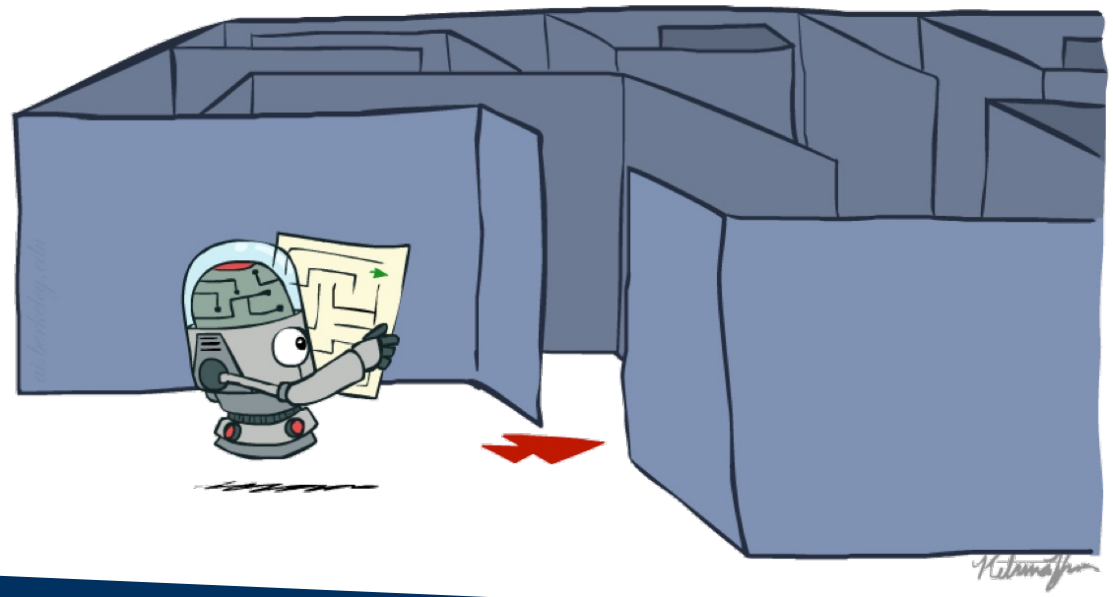
人工智能导论

主讲：王 博

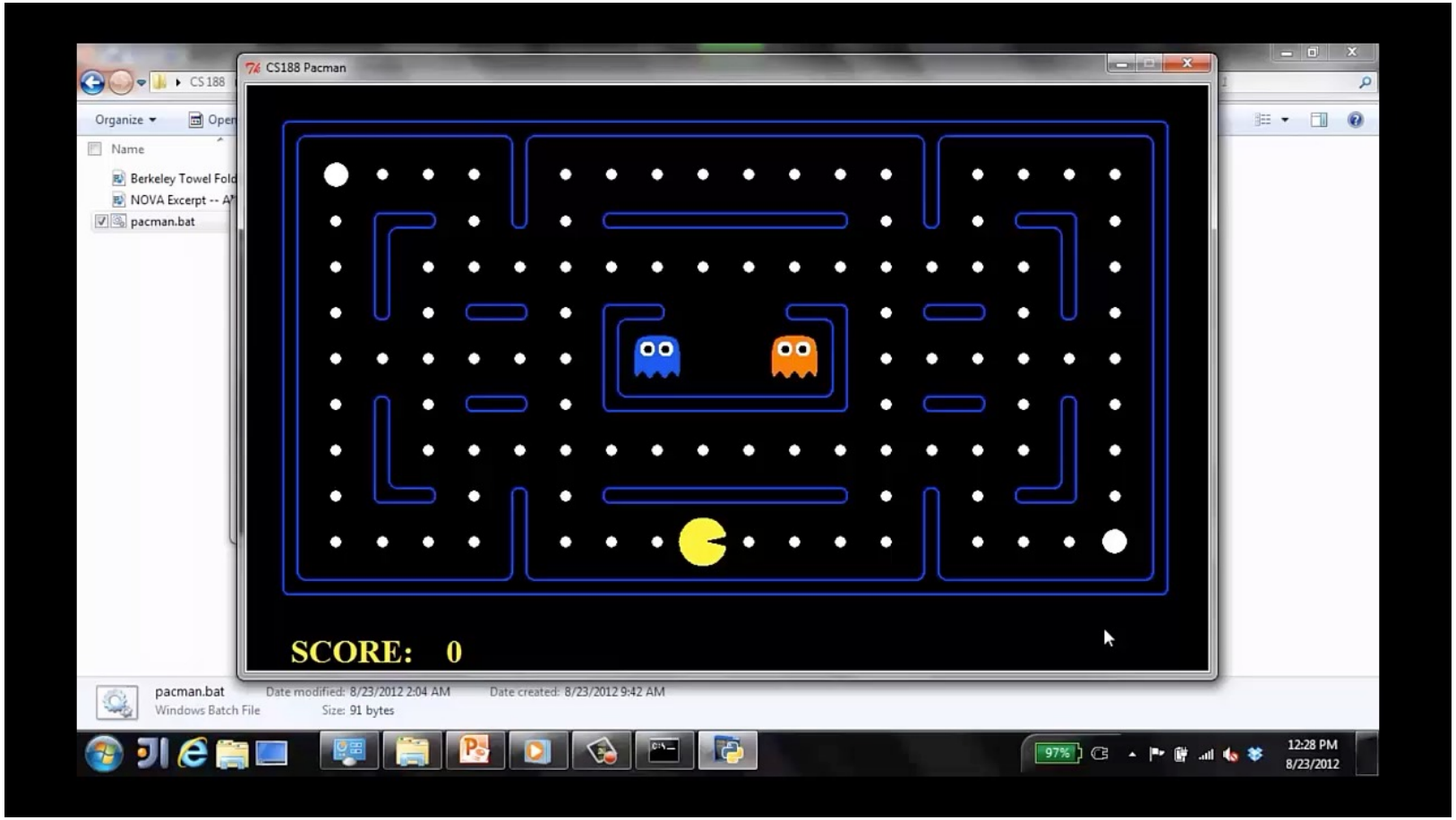
人工智能与自动化学院

第2章 搜索与机器学习

第1节 盲目式搜索与启发式搜索

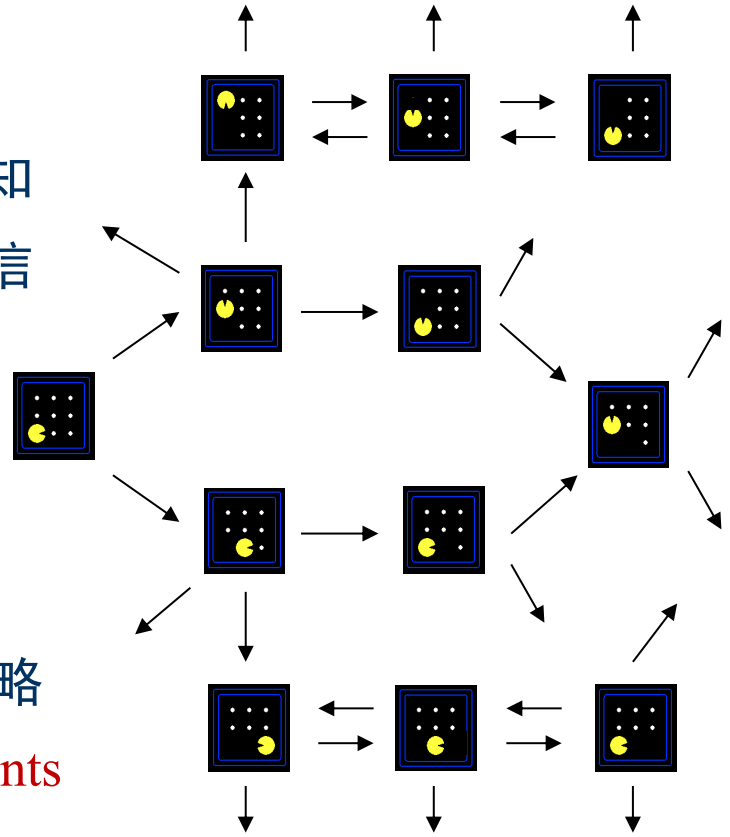


吃豆人Pacman


CS188 Pacman
SCORE: 0
pacman.bat
Windows Batch File
Date modified: 8/23/2012 2:04 AM
Date created: 8/23/2012 9:42 AM
Size: 91 bytes
12:28 PM
8/23/2012

图搜索技术

- 已知 vs. 未知
  - 游戏怎么玩，完全已知
  - 如何直接求解，数学信息不完整
- 将状态转换成图（树）
  - 能选择什么动作？
  - 下一个状态是？
- 能够在执行前找到最优策略
  - Search is useful for agents that plan ahead!



- 假设不考虑ghost！

图搜索技术

1. 先把问题的初始状态作为当前扩展节点对其进行扩展
2. 生成一组子节点，然后检查问题的目标状态是否出现在这些子节点中。
   - 若出现，则搜索成功，找到了问题的解；
3. 若没出现，则再按照某种搜索策略从已生成的子节点中选择一个节点 作为当前扩展节点。
4. 重复上述过程，直到目标状态出现在子节点中或者没有可供操作的节点为止。

搜索 – 目录

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

搜索的基本问题与主要过程

- 搜索中需要解决的基本问题：

  （1）是否一定能找到一个解 Completeness

  （2）找到的解是否是最佳解 Optimality

  （3）时间与空间复杂性如何 Time & space complexity

  （4）是否终止运行或是否会陷入一个死循环 Goal test

- 搜索的主要过程：

  （1）出发点：初始（正向）或目标（逆向）状态。当前状态；

  （2）扫描操作算子集，选择适当的操作，生成新状态

  （3）检查所生成的新状态是否满足结束状态，

  - 满足：得到问题的一个解，包含解答路径
  - 不满足，将新状态作为当前状态，返回第(2)步再进行搜索。

搜索方向

（1）数据驱动：从初始状态出发的正向搜索

- 正向搜索——从问题给出的条件出发。

（2）目的驱动：从目的状态出发的逆向搜索

- 逆向搜索——从想达到的目的入手，看哪些操作算子能产生该目的以及应用这些操作算子产生目的时需要哪些条件。

（3）双向搜索

- 双向搜索——从开始状态出发正向搜索，同时又从目的状态出发逆向搜索，直到两条路径在中间的某处汇合为止。

搜索策略

（1）盲目搜索：在不具有对特定问题的任何有关信息的条件下，按固定的步骤（依次或随机调用操作算子）进行的搜索。

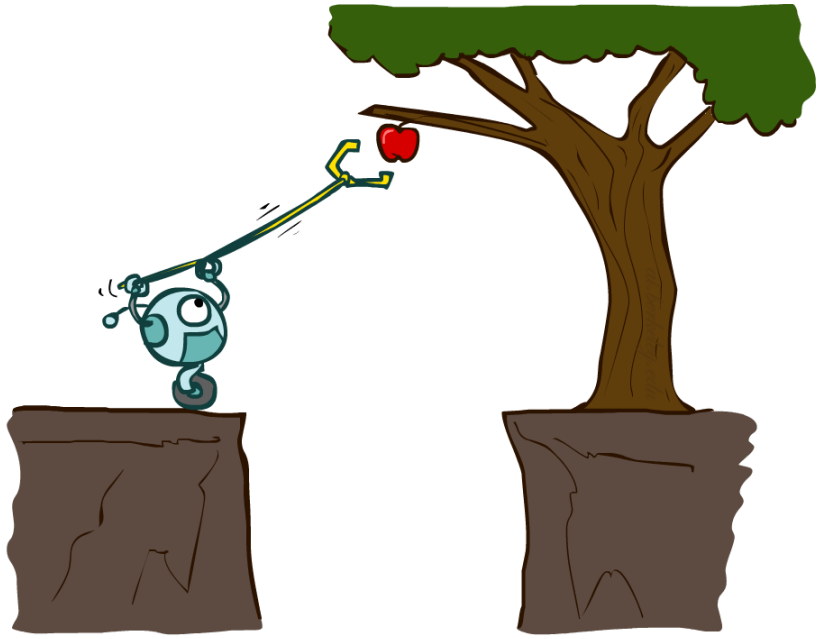
（2）启发式搜索：考虑特定问题领域可应用的知识，动态地确定调用操作算子的步骤，优先选择较适合的操作算子，尽量减少不必要的搜索，以求尽快地到达结束状态。

搜索 – 目录

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

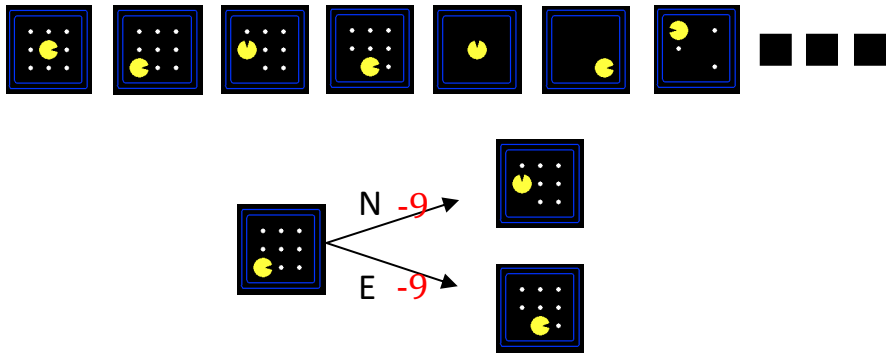
规划智能体

- Planning agents decide based on evaluating future action sequences
- Must have a model of how the world evolves in response to actions
- Usually have a definite goal
- Optimal: Achieve goal at least cost



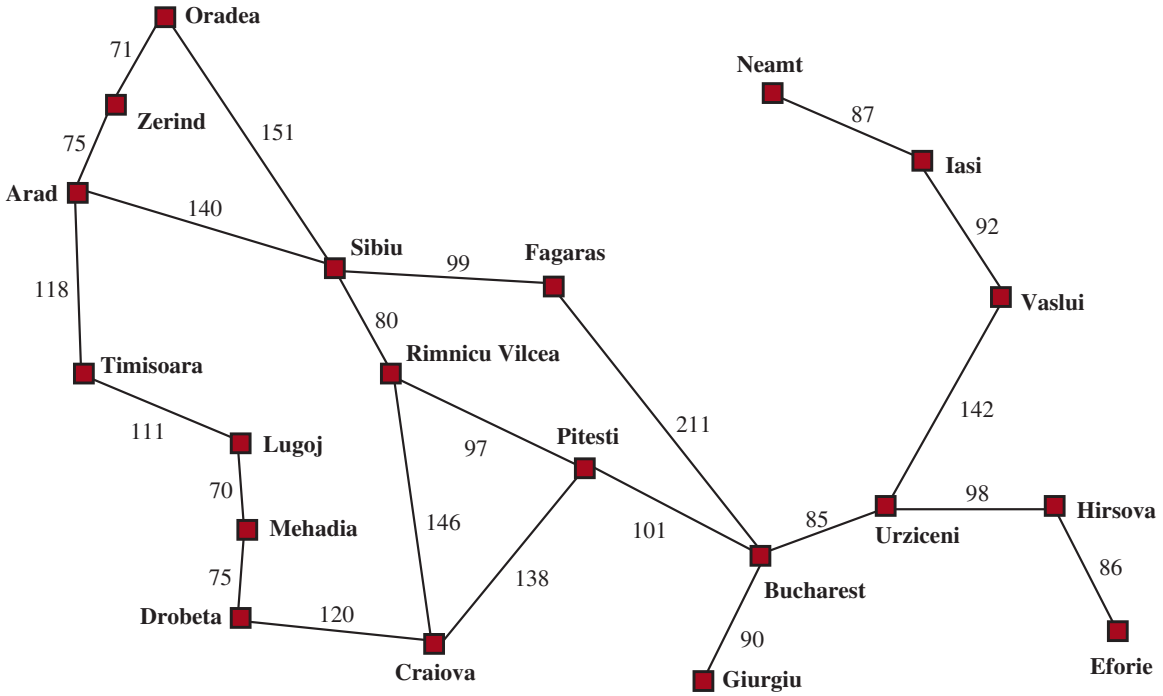
Search Problems

- 一个搜索问题包含:
  - 状态空间 S
  - 初始状态 s_0
  - 每个状态下的动作集合 $A(s)$
  - 转移模型 $Result(s,a)$
  - 目标验证 $G(s)$
    - 没有豆子剩下的 S
  - 动作代价 $c(s,a,s')$
    - +1 per step; -10 food; -500 win; +500 die; -200 eat ghost
- 一个解是一个能达到目标状态的动作序列
- 最优解有所有解中最低的cost



Example: Traveling in Romania

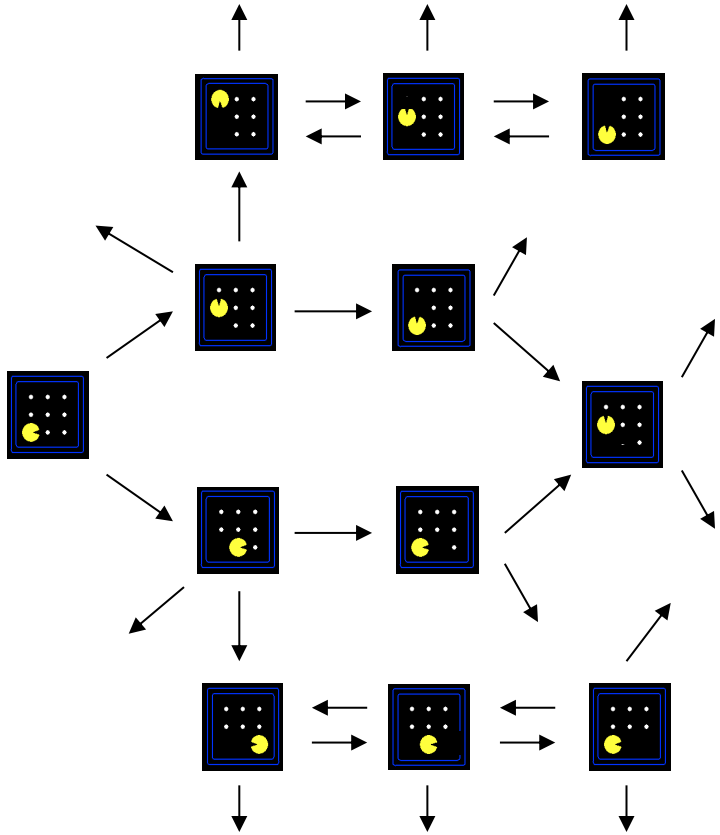
Example: Traveling in Romania



- 状态空间:
  - 访问的城市
- 初始状态
  - Arad
- 动作:
  - 旅行到相邻的城市
- 转移模型:
  - 到达相邻的城市
- 目标检验:
  - *s* = Bucharest?
- 动作代价:
  - 从 *s* to *s'* 的路径长度
- 解是什么?

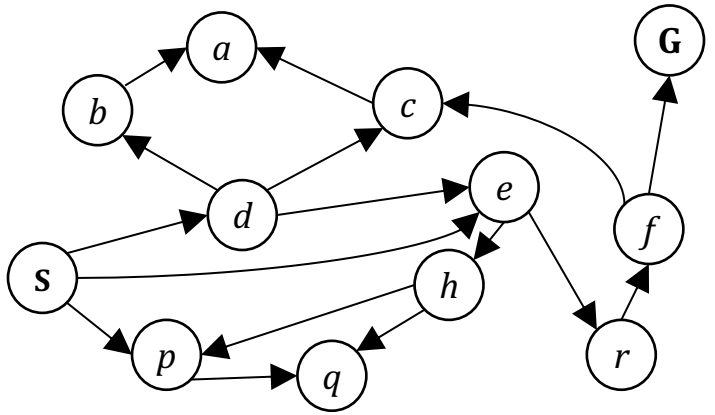
State Space Graphs

- 状态空间图: 搜索问题的数学表达
  - 节点 (抽象了) 环境的状态
  - 变表示状态转移 (标记出造成转移的动作)
  - 目标检验表示为目标节点 (可能唯一)
- 状态空间图中，每个状态只出现一次
- 实际上很难完整画出状态空间图
  - 占用过多存储空间
  - 思想非常有用！



State Space Graphs

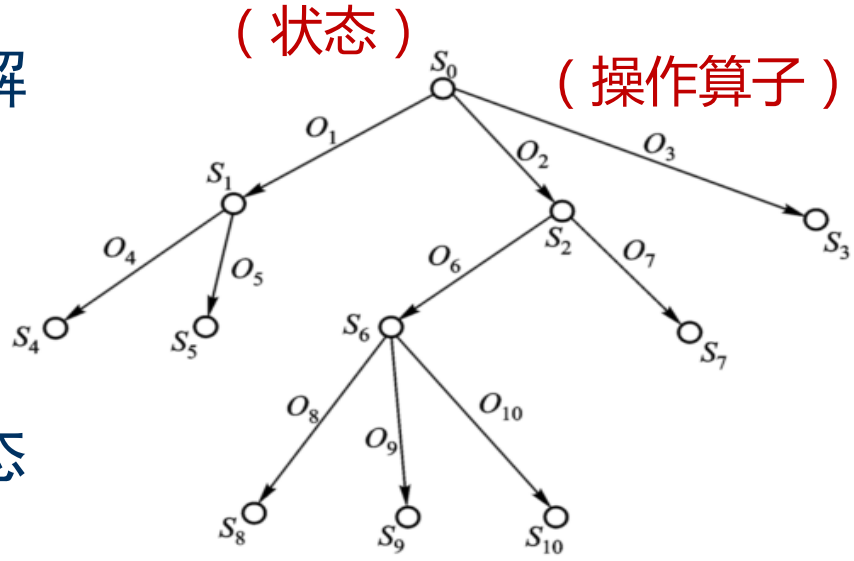
- 状态空间图: 搜索问题的数学表达
  - 节点 (抽象了) 环境的状态
  - 变表示状态转移 (标记出造成转移的动作)
  - 目标检验表示为目标节点 (可能唯一)
- 状态空间图中，每个状态只出现一次
- 实际上很难完整画出状态空间图
  - 占用过多存储空间
  - 思想非常有用！



Tiny state space graph for a tiny search problem

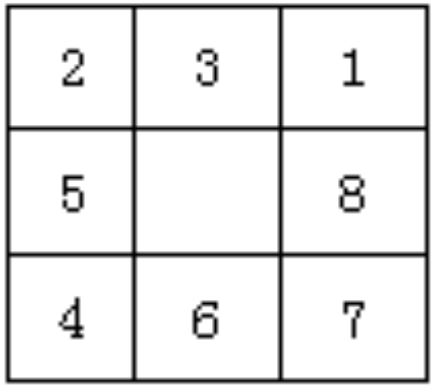
搜索树

- 用有向图描述搜索过程
- 结点表示状态、边表示求解步骤
- 初始状态为根结点
- 一种状态转换为另一种状态的操作算子序列←→图中寻找某一路径

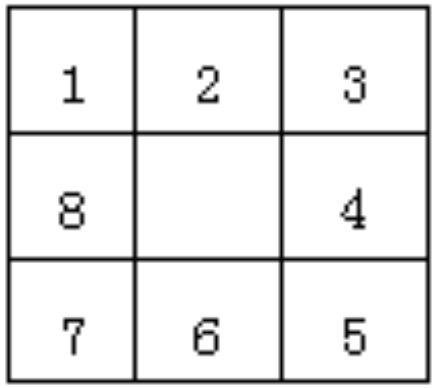


八数码问题的搜索树

- 例：八数码问题的状态空间



初始状态



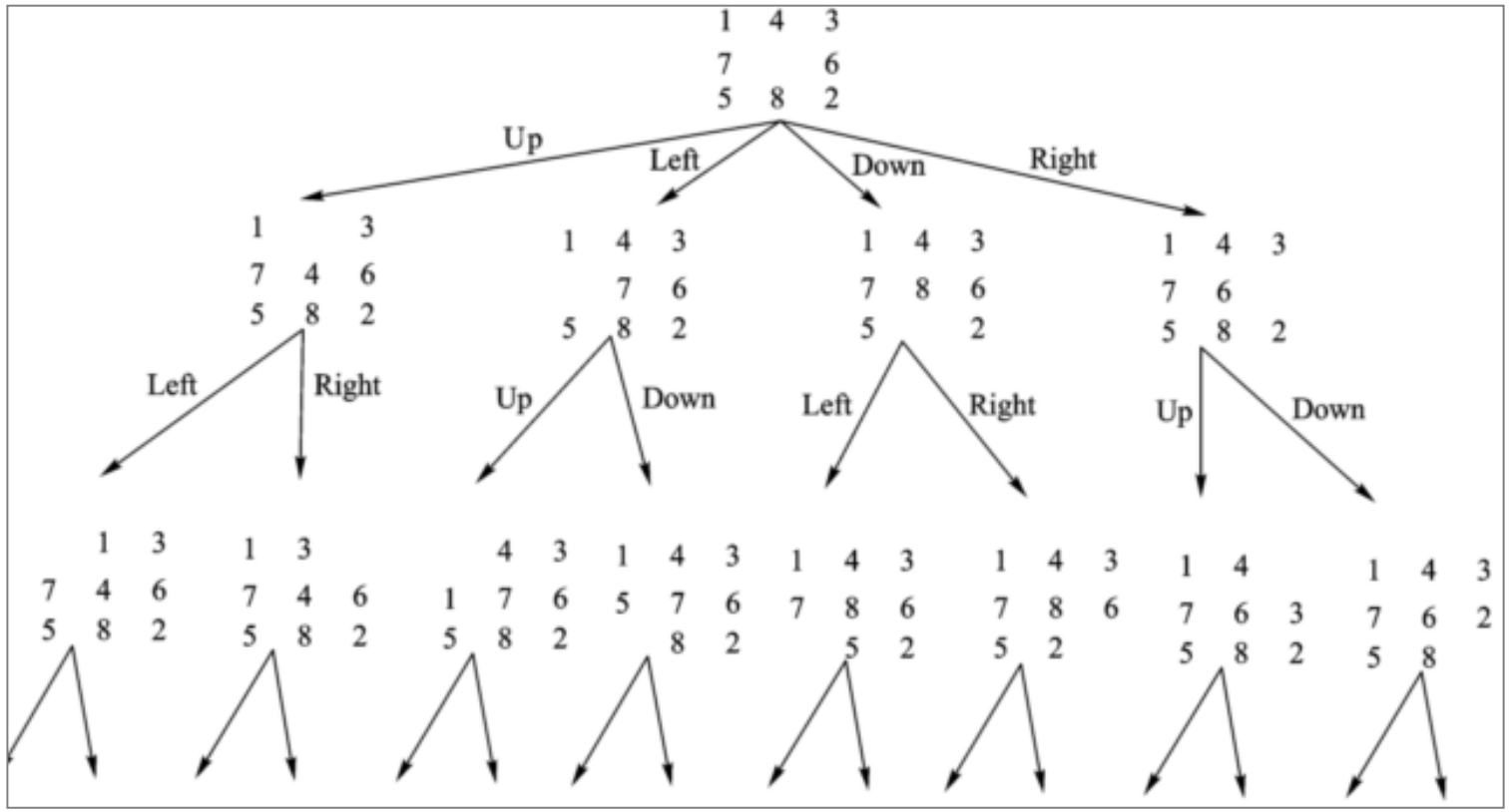
目标状态

状态集 S 所有摆法

操作算子：

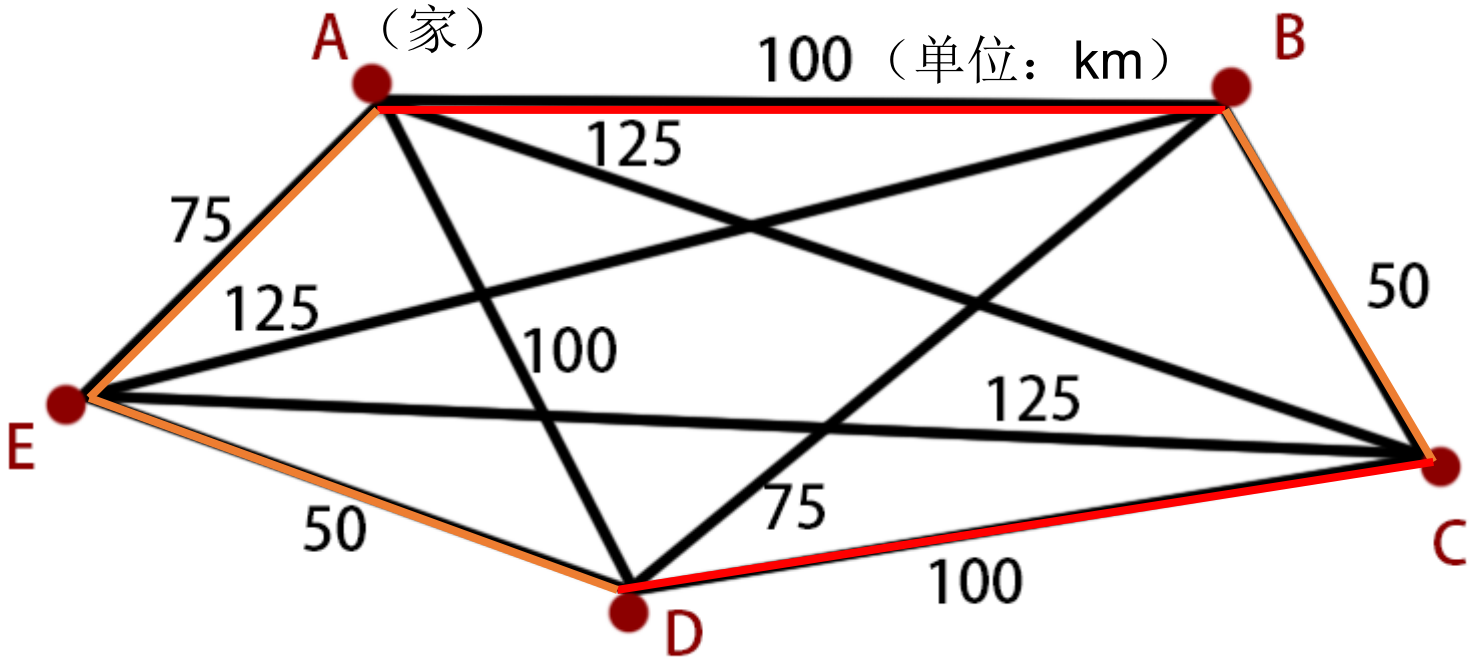
- 将空格向上移Up
- 将空格向左移Left
- 将空格向下移Down
- 将空格向右移Right

八数码问题的搜索树



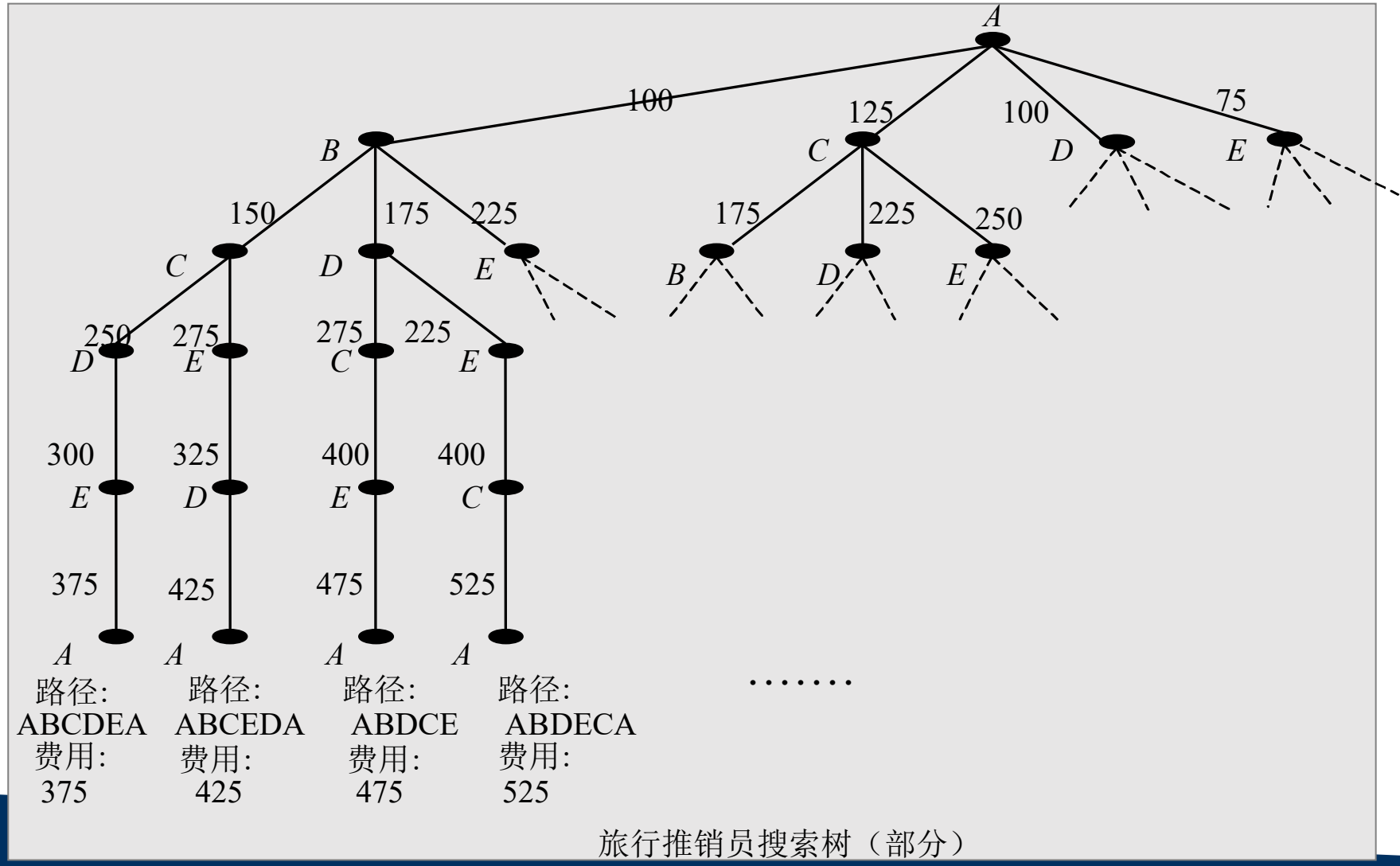
旅行商问题的搜索树

- 例： 旅行商问题（traveling salesman problem, TSP）



可能路径：费用为375的路径（A，B，C，D，E，A）

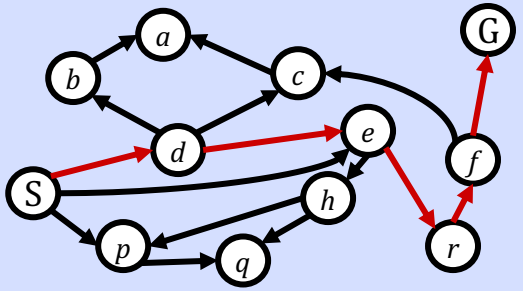
旅行商问题的搜索树



旅行推销员搜索树（部分）

状态空间图 vs. 搜索树

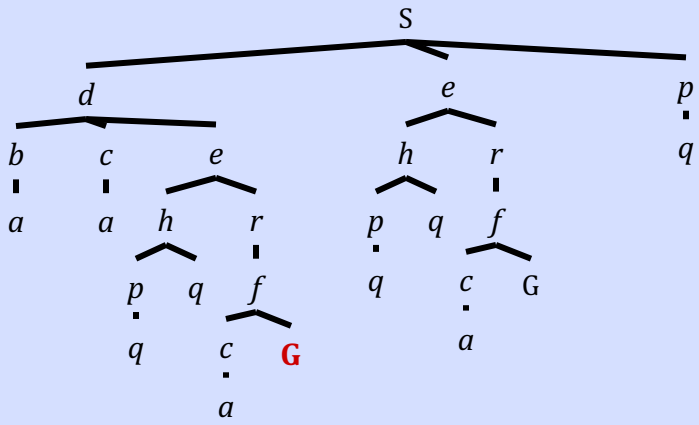
State Space Graph



Each NODE in in the search tree is an entire PATH in the state space graph.

We construct the tree on demand – and we construct as little as possible.

Search Tree



搜索 – 目录

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

深度优先搜索

- 深度优先搜索（Depth-first search, DFS）：首先扩展最新产生的节点，深度相等的节点按生成次序的盲目搜索。

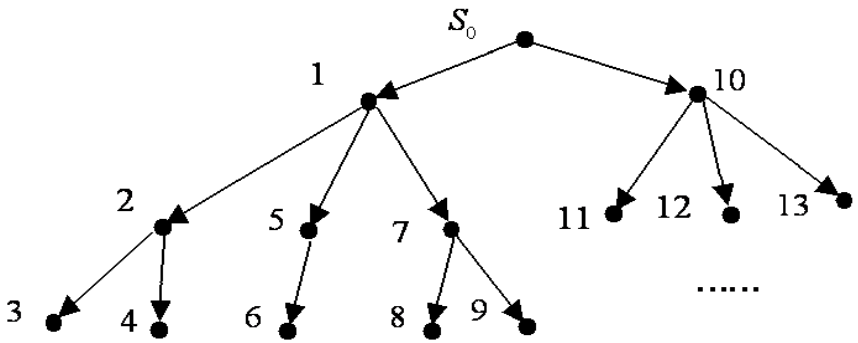
深度优先搜索

- 深度优先搜索（Depth-first search）：首先扩展最新产生的节点，深度相等的节点按生成次序的盲目搜索。

特点：

（1）扩展最深的节点使得搜索沿着状态空间某条单一的路径从起始节点向下进行；

（2）仅当搜索到达一个没有后裔的状态时，才考虑另一条替代的路径。



深度优先搜索

- 通常来讲，在search的过程中，会将所有的点分成几个部分，Explored, Frontier, Unexplored
- Explored 是那些已经找到从起始点到该点的最佳路径的那些点
- Frontier 是还在寻找最优路径的那些点
- Unexplored是那些我们都还没有预见到的点。
- 搜索过程是：把Unexplored的点放到Frontier集合里面，然后把Frontier集合里的点挪到Explored集合里面。当最终的节点挪到Explored里面之后，整个搜索过程就结束了。

深度优先搜索

算法：

- 防止搜索过程沿着无益的路径扩展下去，往往给出一个节点扩展的最大深度——深度界限；
- 将扩展的后继节点放在OPEN表的前端；
- 深度优先搜索算法的Open表是堆栈，先进后出（FILO）。

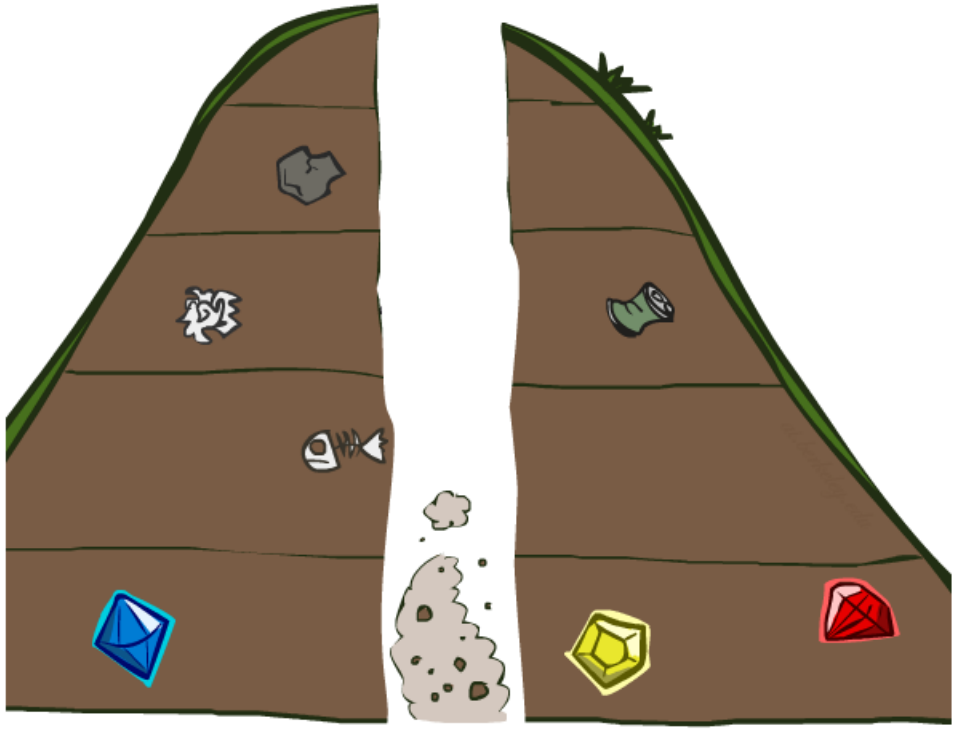
  - Open：已经生成出来但其子状态未被搜索的状态。
  - Closed：记录了已被生成扩展过的状态。

深度优先搜索

- 在深度优先搜索中，当搜索到某一个状态时，它所有的子状态以及子状态的后裔状态都必须先于该状态的兄弟状态被搜索。
- 为了保证找到解，应选择合适的深度限制值，或采取不断加大深度限制值的办法，反复搜索，直到找到解。
- 对任何状态而言，以后的搜索有可能找到另一条通向它的路径。如果路径的长度对解题很关键的话，当算法多次搜索到同一个状态时，它应该保留最短路径。

深度优先搜索

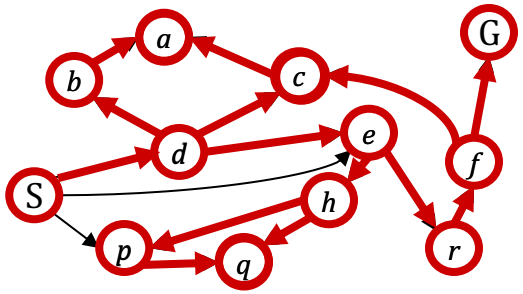
- 深度优先搜索并不能保证第一次搜索到的某个状态时的路径是到这个状态的最短路径。

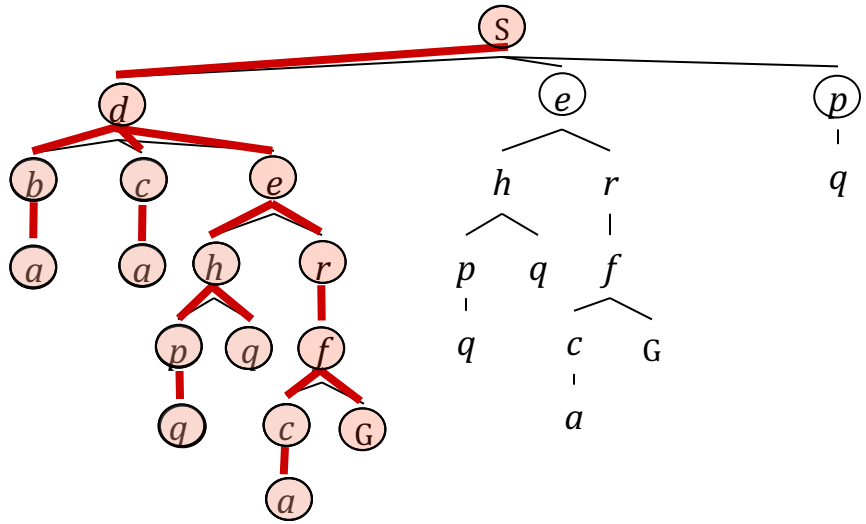


Depth-First Search

Strategy: expand a deepest node first

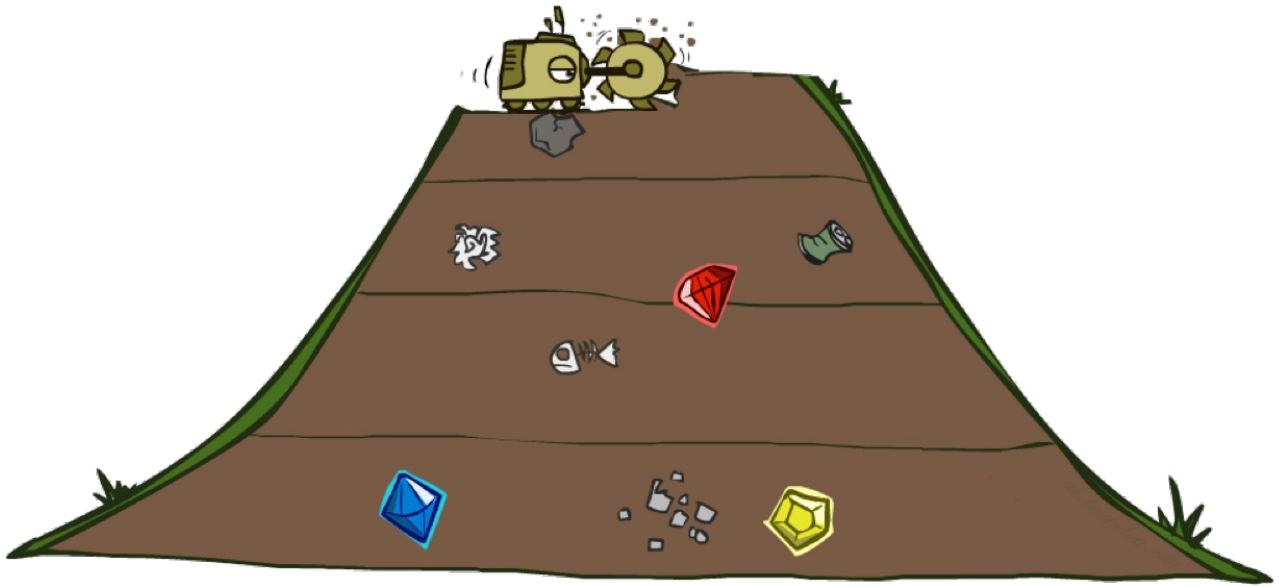
Implementation: Frontier is a LIFO stack





宽度优先搜索

- 宽度优先搜索（Breadth-first search，BFS）：以接近起始节点的程度（深度）为依据，进行逐层扩展的节点搜索方法。



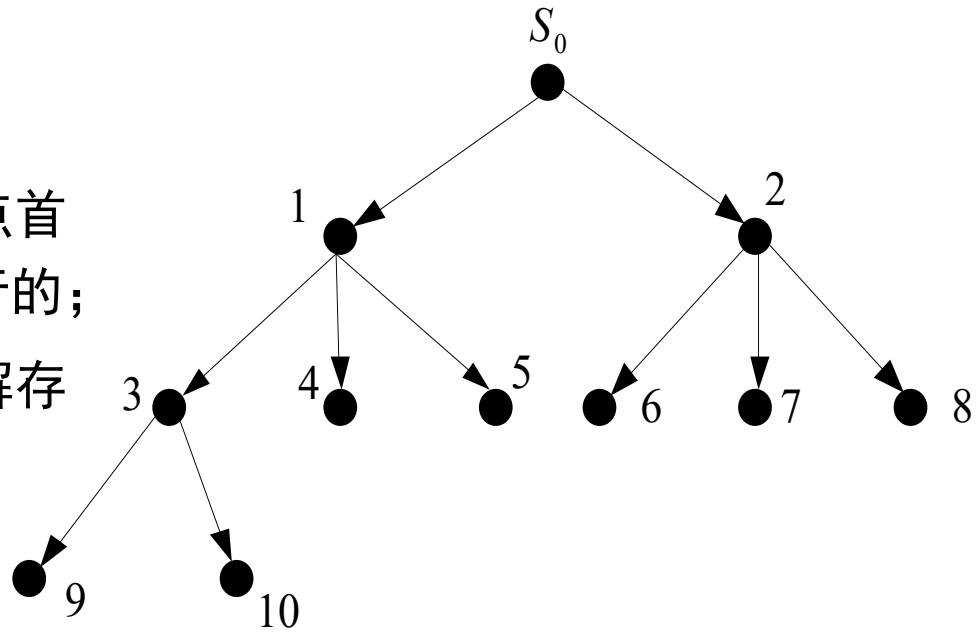
宽度优先搜索

- 宽度优先搜索（Breadth-first search，广度优先搜索）：以接近起始节点的程度（深度）为依据，进行逐层扩展的节点搜索方法。

特点：

（1）每次选择深度最浅的节点首先扩展，搜索是逐层进行的；

（2）一种高价搜索，但若有解存在，则必能找到它。

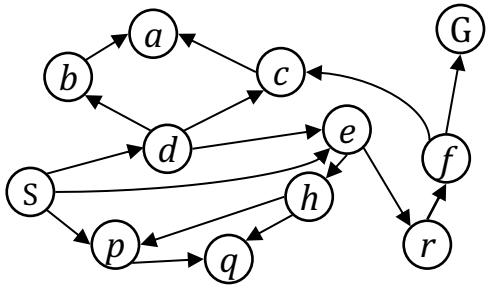


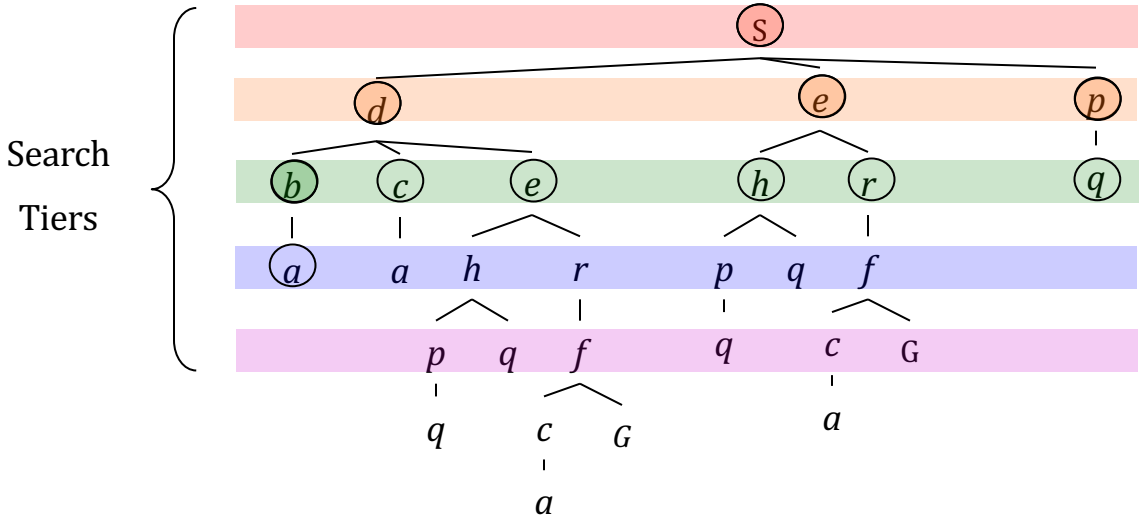
宽度优先搜索法中状态的搜索次序

Breadth-First Search

Strategy: expand a shallowest node first

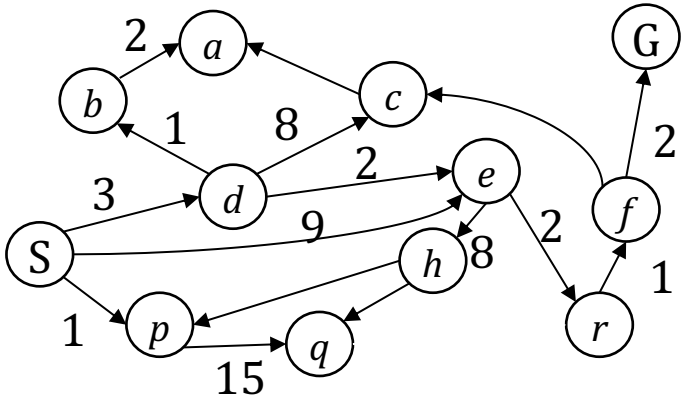
Implementation: Frontier is a FIFO queue





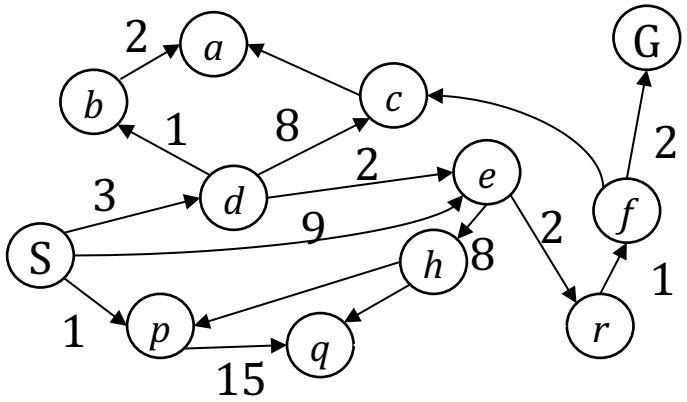
统一代价搜索

- 宽度优先搜索只能处理无权重图搜索问题
- 统一代价搜索（Uniform cost search， UCS）是对宽度优先的改进，可以解决有权重图搜索问题
- 节点的扩展选择以g(n)=从根节点到节点n的代价为依据，选择g(n)最小的节点扩展

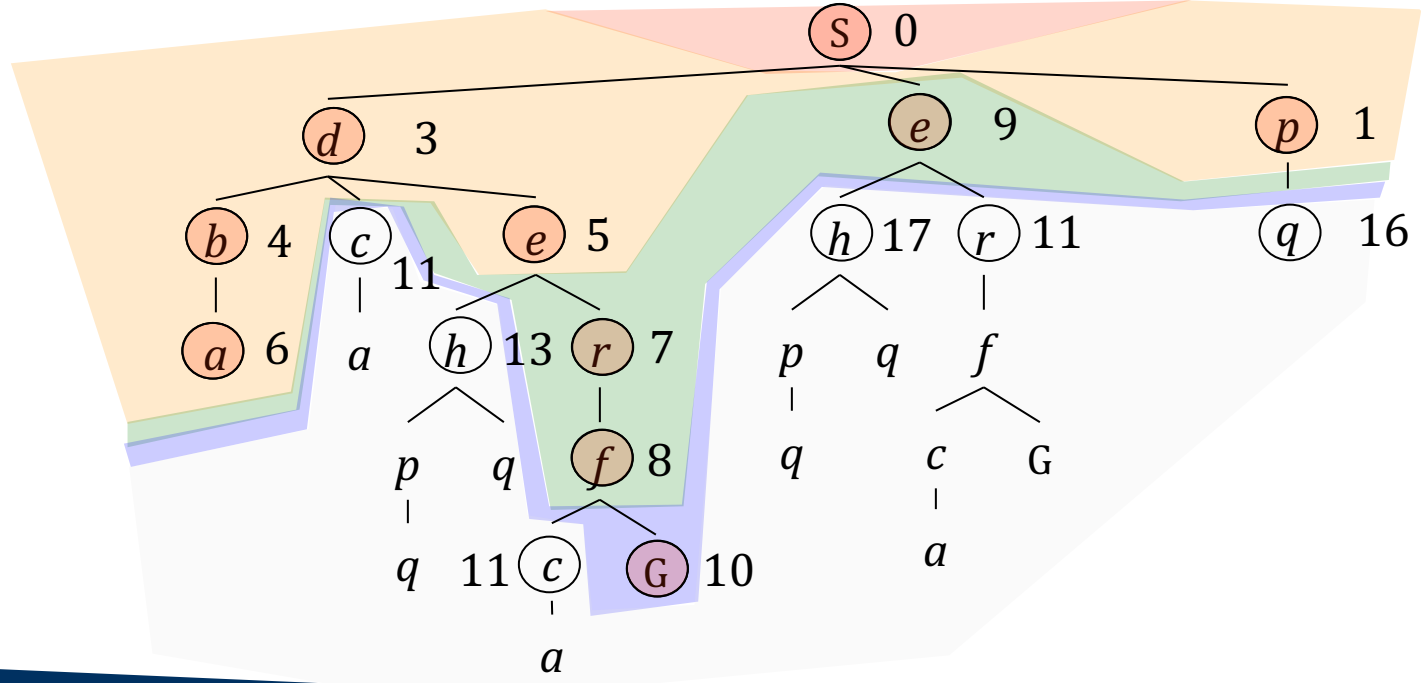


统一代价搜索

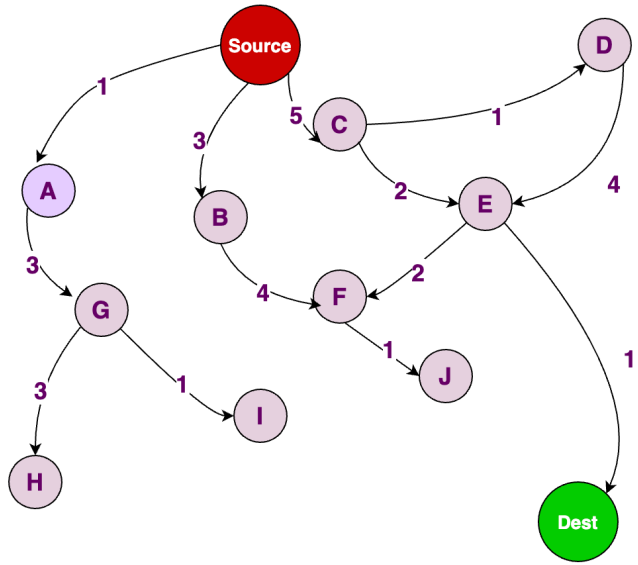
g(n)=从根节点到节点n的代价



Cost contours

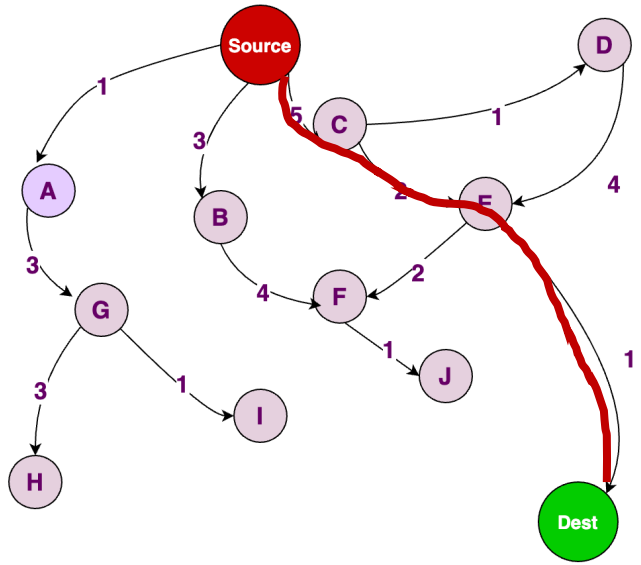


统一代价搜索-练习



The minimum distance between the source and destination nodes is 8.

统一代价搜索-答案

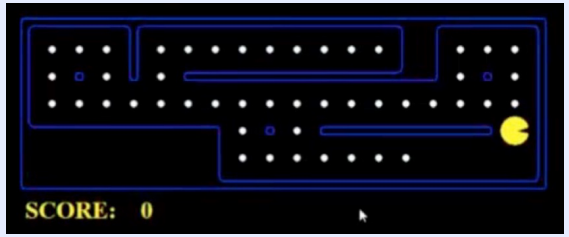


The minimum distance between the source and destination nodes is 8.

	BFS	UCS Dijkstra
Main Concept	Visit nodes level by level based on the closest to the source	In each step, visit the node with the lowest cost
Optimality	Gives an optimal solution for unweighted graphs or weighted ones with equal weights	Gives an optimal solution for both weighted and unweighted graphs
Queue Type	Simple queue	Priority queue
Time Complexity	$O(V+E)$	$O(V+E(logV))$

如何建立问题？

环境状态 包含了所有环境中的细节信息



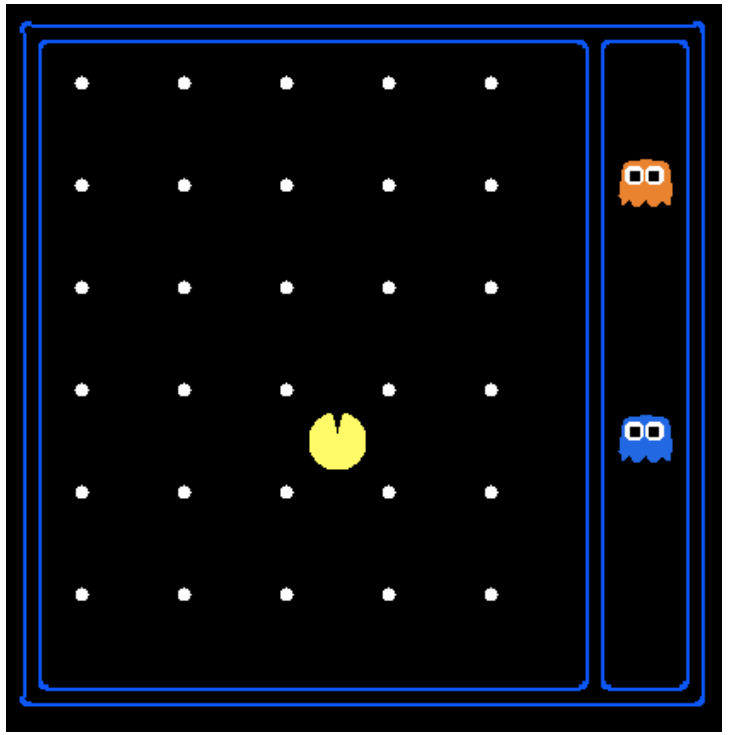
搜索状态 只保留了进行规划所需的信息 (对环境的抽象)

- Problem: Pathing
  - States: (x,y) location
  - Actions: NSEW
  - Transition: update x,y value
  - Goal test: is (x,y)=destination
- Problem: Eat-All-Dots
  - States: {(x,y), dot Booleans}
  - Actions: NSEW
  - Transition: update x,y and possibly a dot Boolean
  - Goal test: dots all false

状态空间大小

- 环境状态World state:
  - Agent positions: 120
  - Food count: 30
  - Ghost positions: 12
  - Agent facing: NSEW

- 状态维数：
  - 世界状态?
    $120x(2^{30})x(12^2)x4$
  - 路径规划所需状态?
    120
  - 吃光所有豆豆所需的状态?
    $120x(2^{30})$



搜索 – 目录

- 搜索的基本概念
- 状态空间表示方法
- 盲目搜索策略
- 启发式搜索策略

启发式策略

- 启发式信息：用来简化搜索过程有关具体问题领域的特性信息叫做启发信息
- 启发式图搜索策略（利用启发信息的搜索方法）的特点：重排OPEN表，选择最有希望的节点加以扩展
- 适合用启发式策略的两种基本情况：

  （1）由于存在问题陈述和数据获取的模糊性，可能会使代求解问题没有一个确定的解；

  （2）虽然一个问题可能有确定解，但是其状态空间特别大，搜索中生成扩展的状态数会随着搜索的深度呈指数级增长。

启发信息和估价函数

在具体求解中，能够利用与该问题有关的信息来简化搜索过程，称此类信息为启发信息。

启发式搜索：利用启发信息的搜索过程。

- 按运用的方法分类：
  - （1）陈述性启发信息：用于更准确、更精炼地描述状态
  - （2）过程性启发信息：用于构造操作算子
  - （3）控制性启发信息：表示控制策略的知识
- 按作用分类：
  - （1）用于扩展节点的选择，即用于决定应先扩展哪一个节点，以免盲目扩展
  - （2）用于生成节点的选择，即用于决定要生成哪些后继节点，以免盲目生成过多无用的节点
  - （3）用于删除节点的选择，即用于决定删除哪些无用节点，以免造成进一步的时空浪费

启发信息和估价函数

- 估价函数（evaluation function）：估算节点“希望”程度的量度。
- 估价函数值 $f(n)$ ：从初始节点经过n节点到达目标节点的路径的最小代价估计值，其一般形式是

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

- $g(n)$：从初始节点 S_0 到节点 n 的实际代价；
- $h(n)$：从节点 n 到目标节点 S_g 的最优路径的估计代价，称为启发函数。

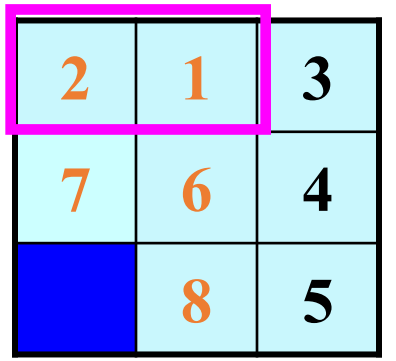
$h(n)$比重大：降低搜索工作量，但可能导致找不到最优解；

$h(n)$比重小：一般导致工作量加大，极限情况下变为盲目搜索，但可能可以找到最优解。

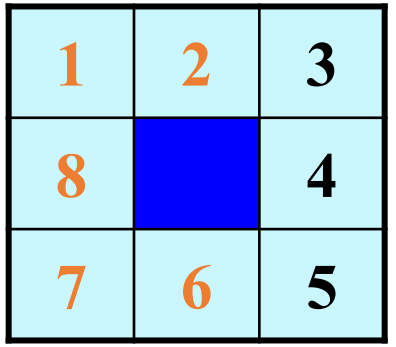
启发信息和估价函数

例 八数码问题的启发函数：

- 启发函数1：取一棋局与目标棋局相比，其位置不符的数码数目，例如$h(S_0) = 5$；
- 启发函数2：各数码移到目标位置所需移动的距离的总和，例如$h(S_0) = 6$；
- 启发函数3：对每一对逆转数码乘以一个倍数，例如3倍，则$h(S_0) = 3$；
- 启发函数4：将位置不符数码数目的总和与3倍数码逆转数目相加，例如$h(S_0) = 8$。



初始棋局

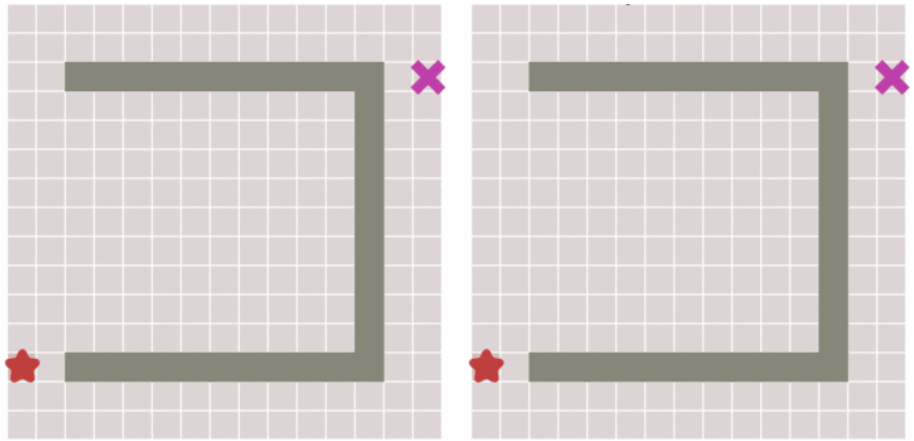


目标棋局

最佳优先搜索

- 最佳优先搜索算法（Best-First-Search）是一种启发式搜索算法，基于宽度优先搜索，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历（估价函数 $f(n)=h(n)$），直到找到目标节点或者遍历完所有点，算法结束。
- BFS算法不能保证找到的路径是一条最短路径，但是其计算过程相对于UCS算法会更快。

Dijkstra



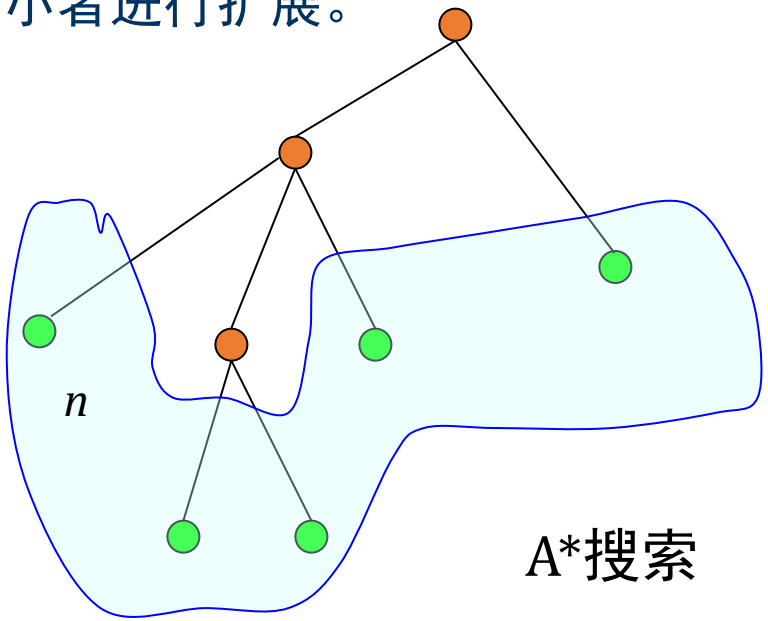
BFS

A*搜索

- A* 搜索算法：使用了估价函数 f 的最佳优先搜索。
  - 估价函数 $f(n) = g(n) + h(n)$
  - 如何寻找并设计启发函数$h(n)$，然后以$f(n)$的大小来排列OPEN表中待扩展状态的次序，每次选择 $f(n)$ 值最小者进行扩展。

$g(n)$：状态n的实际代价，例如搜索的深度；

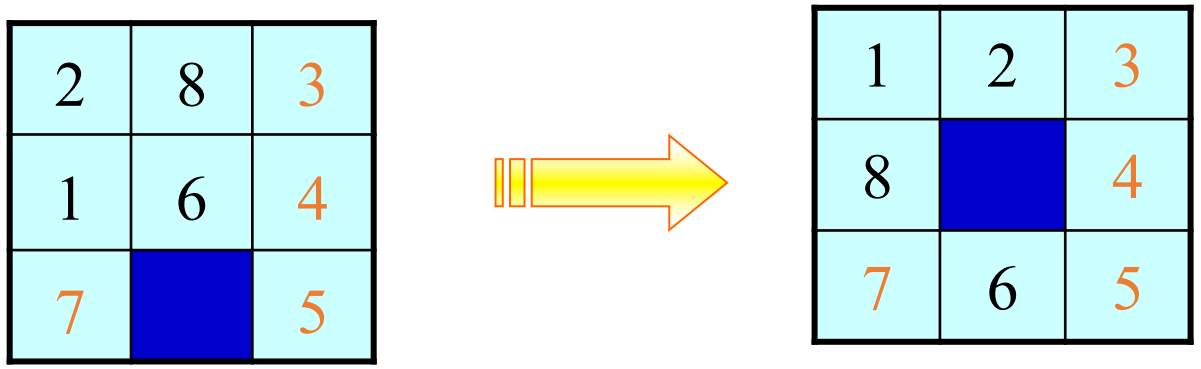
$h(n)$：对状态n与目标“接近程度”的某种启发式估计。



A*搜索

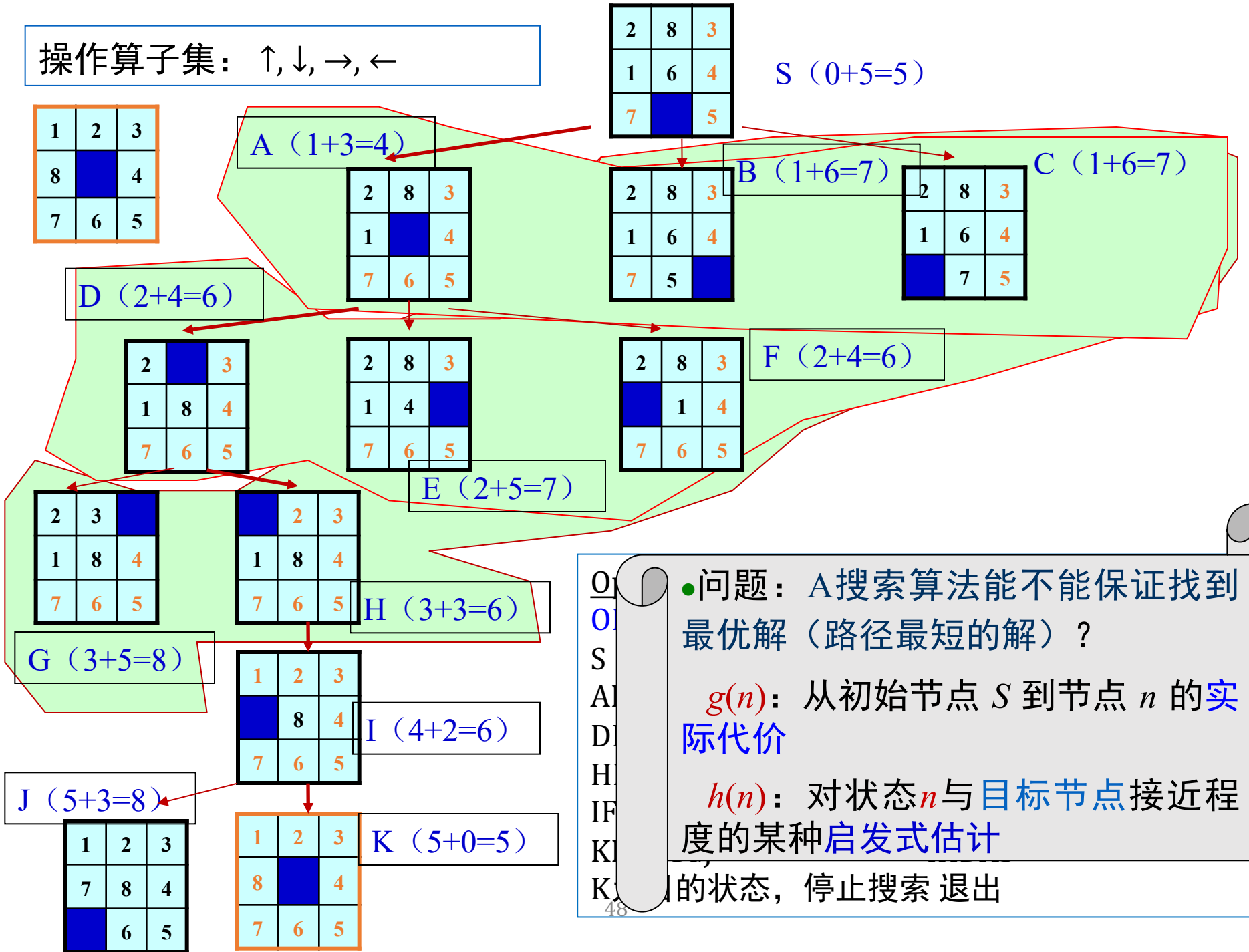
例：利用A*搜索算法求解八数码问题，问最少移动多少次就可达到目标状态？

- 估价函数定义为 $f(n) = g(n) + h(n)$
- $g(n)$：节点 n 的深度，如 $g(S_0)=0$。
- $h(n)$：节点 n 与目标棋局不相同的位数（包括空格），简称“不在位数”，如 $h(S_0)=5$。



初始状态 S_0　　目标状态

操作算子集：↑, ↓, →, ←



- 问题：A搜索算法能不能保证找到最优解（路径最短的解）？

$g(n)$：从初始节点 S 到节点 n 的实际代价

$h(n)$：对状态n与目标节点接近程度的某种启发式估计

的状态，停止搜索 退出

A*搜索

- A*搜索算法的特点：
  - A*是完备的
  - A*是最优的，如果下列条件成立：

（1）是否一定能找到一个解 Completeness

（2）找到的解是否是最佳解 Optimality

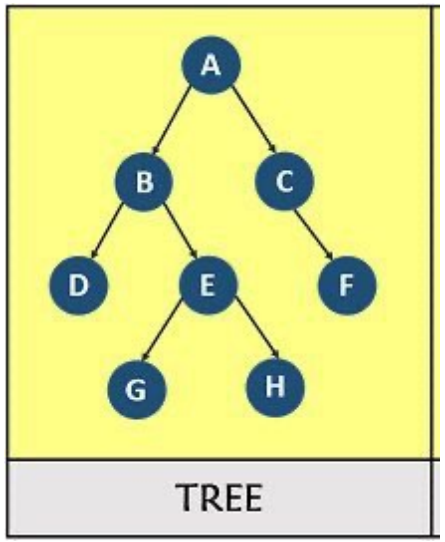
（3）时间与空间复杂性如何 Time & space complexity

（4）是否终止运行或是否会陷入一个死循环 Goal test

A*搜索的最优性条件

- 启发函数的可采纳性（Admissibility)：

$$h(n) \leq h^*(n)$$



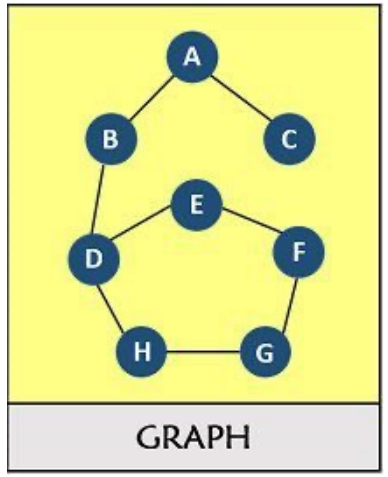
对树搜索总是成立

上例中的八数码问题中定义的$h(n)$表示了“不在位”的数码数，满足上述条件，因此所得解路径为最优解路径。

A*搜索的最优性条件

- 启发函数的单调性（Consistency / Monotonicity）：

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



$c(n, a, n')$是从节点n到节点n'的实际代价。

图搜索中A*最优性的条件

一般来说，大多数可采纳的启发式函数都是单调的。

补充-启发函数的信息性

- 信息性：
  - 在两个A*启发策略的h_1和h_2中，如果对搜索空间中的任一状态n都有$h_1(n) \leq h_2(n)$，就称策略h_2比h_1具有更多的信息性。
  - 如果某一搜索策略的$h(n)$越大，则A*算法搜索的信息性越多，所搜索的状态越少。
  - 但更多的信息性可能需要更多的计算时间，也有可能抵消减少搜索空间所带来的益处。

博弈中的搜索

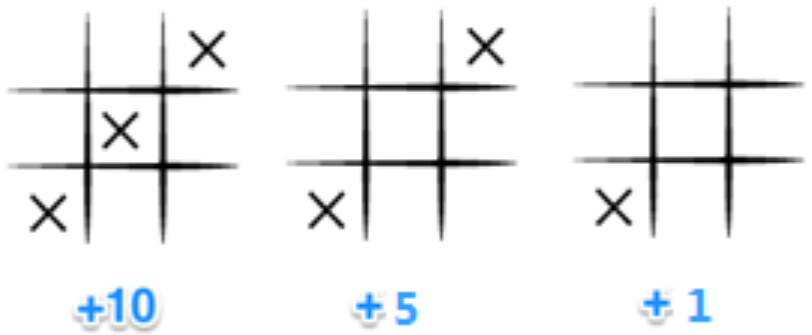
- 例：一字棋
  - 在九宫棋盘上，从空棋盘开始，双方轮流在棋盘上摆各自的棋子×或〇（每次一枚），谁先取得三子一线（一行、一列或一条对角线）的结果就取胜。

- × 和 〇 能够在棋盘中摆成的各种不同的棋局就是问题空间中的不同状态。
- 9个位置上摆放{空， ×， 〇}有 3^9 种棋局。
- 可能的走法： $9\times8\times7\times\cdots\times1$。

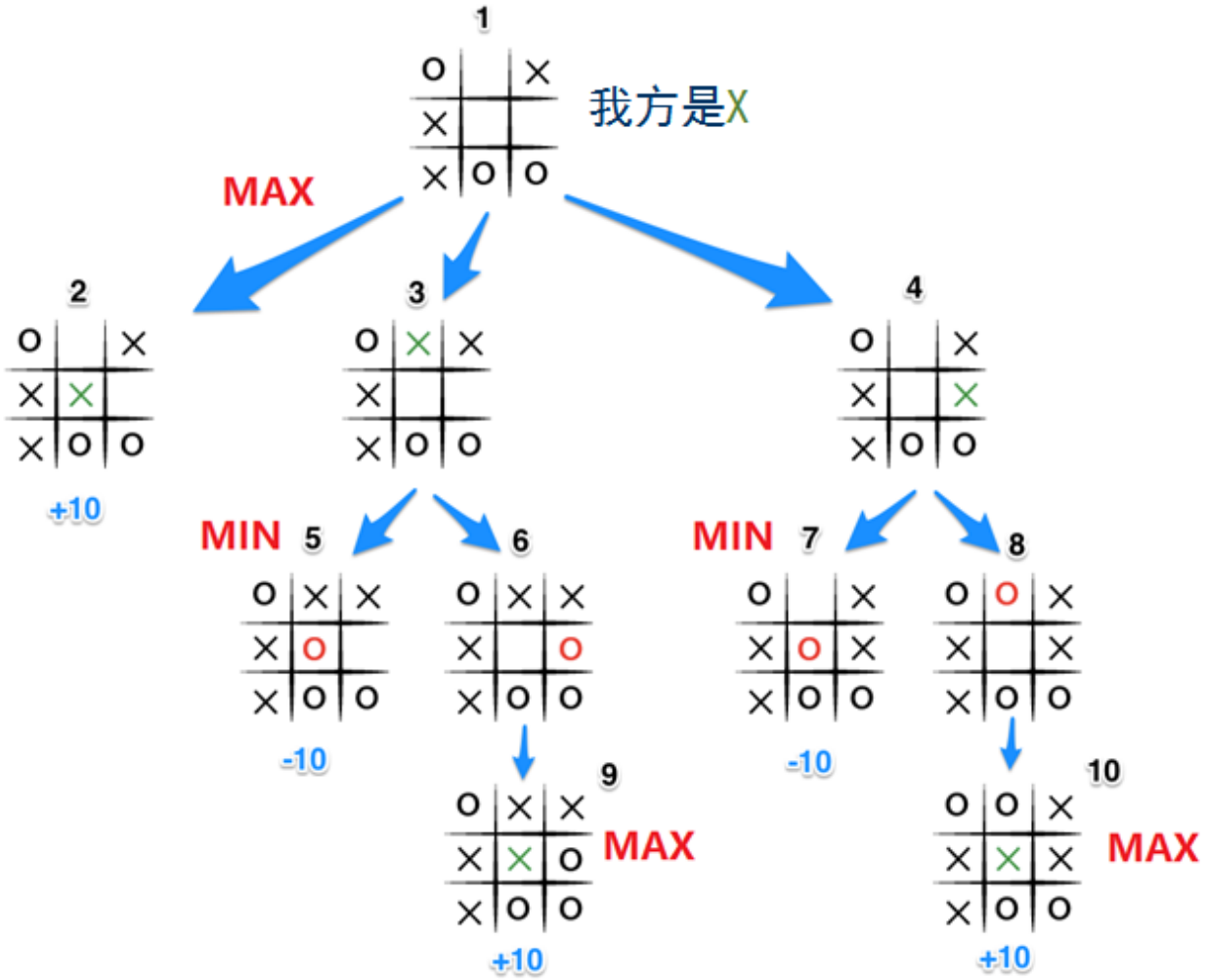
博弈中的搜索

- 博弈树：深度优先遍历下子情况
- MIN-MAX策略：每种棋盘情况有一个估价函数。当我方(MAX方)行动时，必须走棋使估价函数最大的一步，即MAX；而敌方（MIN)行动时，必然会走棋使估价函数最小的一步，即MIN。

- 设计估价函数：
  - 有三个相同的棋子
  - 有两个相同的棋子+一个空格
  - 有一个相同的棋子+两个空格

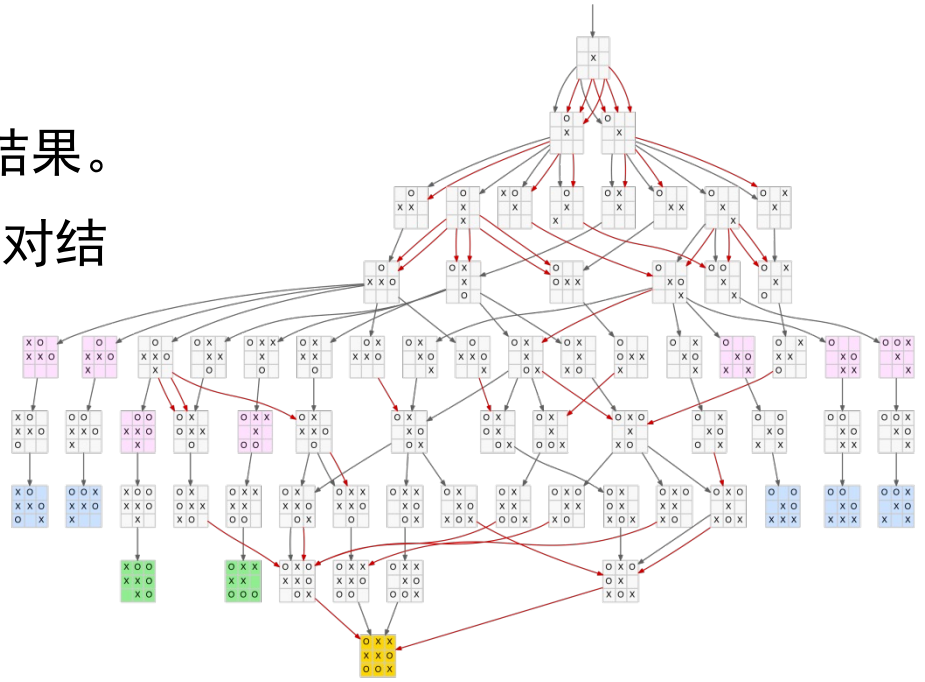


博弈中的搜索



博弈中的搜索

- 博弈树搜索时，先假设双方足够聪明，搜索尝试走完所有可能，然后得到行动时如何走棋的步骤。
- 实际是先生成了一整棵博弈树，枚举当前局面后的每一种下法，计算每个局面后续的赢棋概率，选择概率最高的走法。
  - 如果搜索树极大，则无法在有效时间内返回结果。
  - 使用剪枝算法来减少搜索结点（α-β剪枝）或对结点进行采样（蒙特卡洛树搜索）



博弈中的搜索

- AlphaGo/Zero
  - 搜索树太广、太深
  - 使用蒙特卡洛树搜索

基本的MCTS有4个步骤：
Selection, Expansion, Simulation, Back-propagation

MCTS：不穷举所有组合，找到最优或次优位置