

**信息网络综合专题**

­­应用层阅读报告

|  |  |
| --- | --- |
| 学 院： | 电子信息工程 |
| 班 级： | 通信1905班 |
| 姓 名： | 秦欣毓 |
| 学 号： | 19211144 |
| 指导教师： | 陈一帅、郭宇春 |

目录

[前言 1](#_Toc99103710)

[搜索算法的背景和定义 2](#_Toc99103711)

[无信息搜索算法 6](#_Toc99103712)

[宽度优先搜索BFS 6](#_Toc99103713)

[统一代价搜索UCS 7](#_Toc99103714)

[深度优先搜索DFS 8](#_Toc99103715)

[深度受限搜索DLS 9](#_Toc99103716)

[迭代加深搜索IDS 9](#_Toc99103717)

[双向搜索BS 10](#_Toc99103718)

[有信息搜索算法 11](#_Toc99103719)

[贪心算法Greedy best-first search 12](#_Toc99103720)

[A\* 搜索 13](#_Toc99103721)

[迭代最佳优先搜索RBFS 17](#_Toc99103722)

[启发式设计 18](#_Toc99103723)

[习题解答 21](#_Toc99103724)

## 前言

本文是对最近阅读《Artificial Intelligence: A Modern Approach》第三版所作的学习笔记，书中第二章Problem-solving与课上所学的Searching模块密切相关，我将对其进行较深入的学习，并进行概括总结。

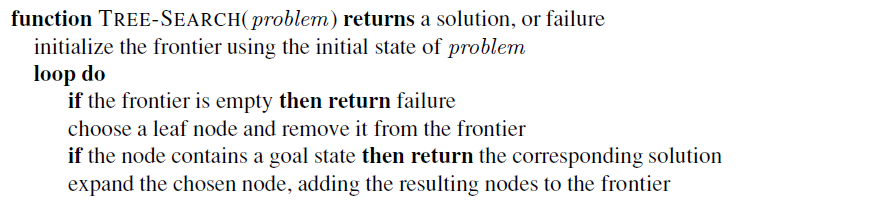
人工智能可以被建模成：学习、模型、推断。所谓的搜索往往指的是根据模型推断的过程。在Problem-solving章节中我们了解了人工智能中Agent如何通过一系列动作来达到目标，即搜索实现目标的方法，涉及了各种经典的搜索算法。其中内容分为四部分：Solving Problems by Searching、Beyond Classical Search、Adversarial Search、Constraint Satisfaction Problems。

Solving Problems by Searching中通过举例子的方式介绍了解决问题的Agent，在最简单的任务环境中，问题的解决方案始终是一个固定的动作序列。同时，描述了几种可用于解决问题的通用搜索算法，这些算法可分为无信息搜索算法，即除了问题的定义之外未提供任何关于问题的信息，还有有信息搜索算法。Beyond Classical Search 中放宽了假设，主要研究局部搜索算法。而Adversarial Search讨论在有其他Agent计划与我们对抗时，该如何预先规划的问题。最后，Constraint Satisfaction Problems基于约束满足问题提出了CSP搜索算法，即一组变量，每个变量都有自己的值。当每个变量都有自己的赋值同时满足所有关于变量的约束时，问题就得到了解决。这类问题就叫做约束满足问题。实际上就是就优化过的值域范围中遍历所有的可能性，然后找到满足条件的结果。下面我将对这四节的内容进行整合归纳：

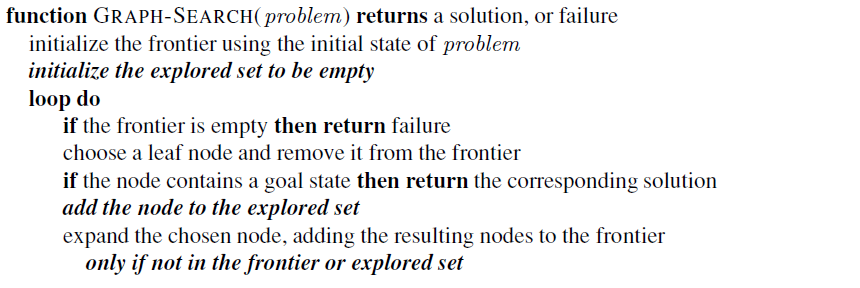
## 搜索算法的背景和定义

首先，搜索算法可以分为树搜索和图搜索。

树搜索：



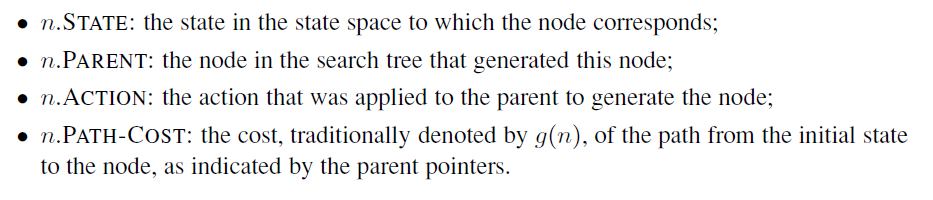
图搜索：



算法开始的时候把初始节点放到 frontier里，接着对相邻节点进行遍历扩展，如果frontier空了却没有找到最终的目标，说明目标路径不存在，算法返回failure，否则将节点frontier集合中的节点移出进行检测，若节点为目标节点则返回solution，即解决方法。若节点不是目标节点，则继续对节点进行扩展，循环搜索，添加到frontier集合中。

可以看出，树搜索和图搜索都有frontier集合，用来记录将要探索的节点，不同的是图搜索比树搜索多维护了一个explored集合，图搜索不允许重复访问节点，这个集合用来记录算法经过的节点，通过检查新的节点是否在这个队列中，图搜索避免了对节点的重复搜索，即{frontier} ∩ {explored} = ø。而树搜索允许重复访问节点。

其中对节点的描述如下：一个节点包括：

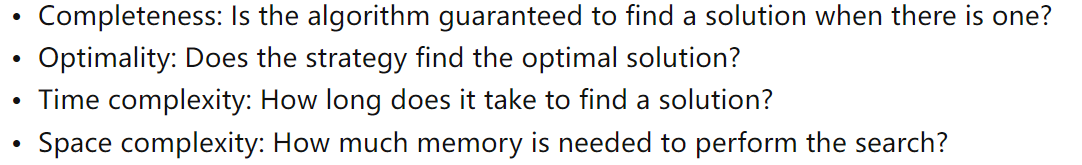


其中关于STATE，CS221给出了一个很好的解释：

A **state** is a summary of all the past actions **sufficient to choose future actions optimally**.

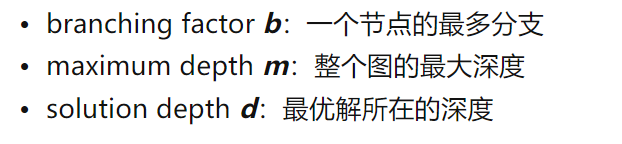
就好比在一盘棋局中某一时刻的棋手，他能根据之前下的棋对现有的棋局进行分析，对下一步棋子的落点做出合适的判断，而棋手有不同的行棋策略，这就对应不同的搜索算法。

最后是描述算法好坏的指标：

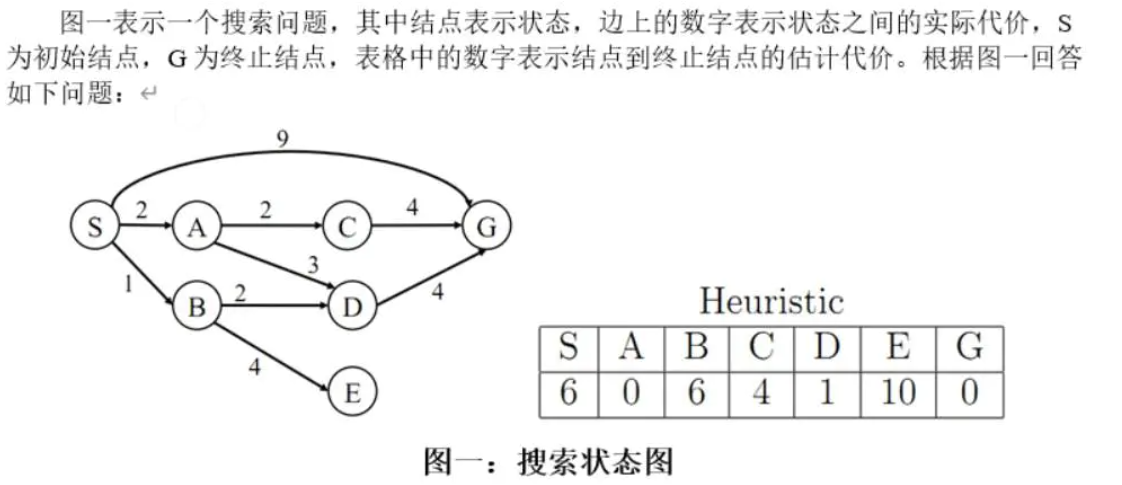


即完整性、最优性、时间复杂度、空间复杂度。

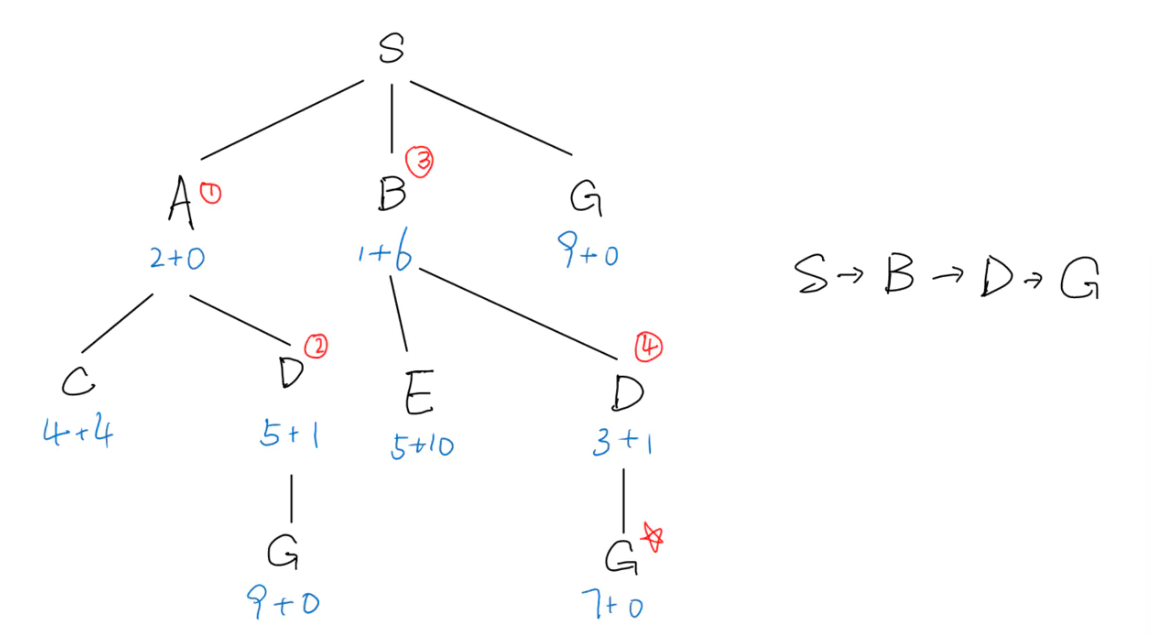
为了描述后两个性能指标，我们在图搜索上定义一些常数（这些常数由问题确定，与算法无关）



在对这部分知识的学习中，我对树搜索和图搜索两者的区别进行了进一步了解，两者的不同之处在于遍历模式的不同，图搜索需要跟踪访问状态，需要更多的内存，而树搜索由于可能对节点进行重复的访问，导致效率更低，但有时需要这种明显的低效率，以下面这道题为例子：

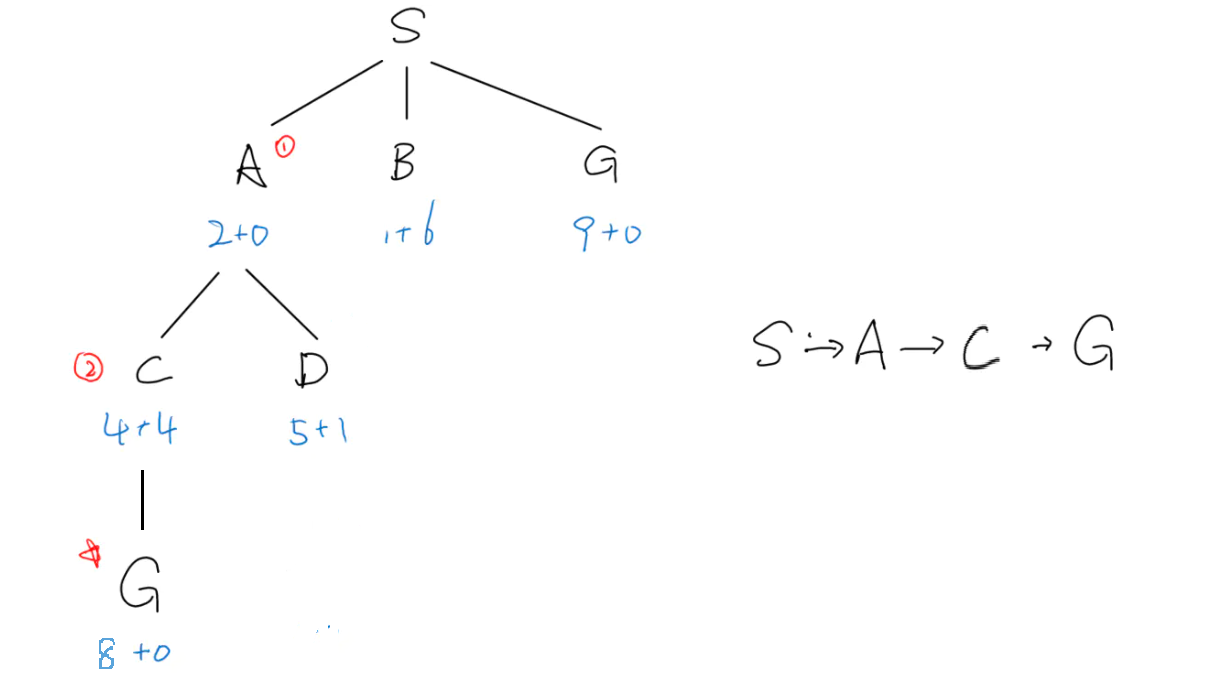


问题1：给出A\*的树搜索路径，要求给出每一步的扩展结点和代价值。



|  |  |
| --- | --- |
| 扩展路径 | 拓展但未经过的路径 |
| S | S-A(2+0); S-B(1+6); S-G(9+0) |
| S-A | S-A-D(5+1); S-B(1+6); S-A-C(4+4); S-G(9+0) |
| S-A-D | S-B(1+6); S-A-C(4+4); S-G(9+0); S-A-D-G(9+0) |
| S-B | S-B-D(3+1); S-A-C(4+4); S-G(9+0); S-A-D-G(9+0); S-B-E(5+10) |
| S-B-D | S-B-D-G(7+0); S-A-C(4+4); S-G(9+0); S-A-D-G(9+0); S-B-E(5+10) |
| S-B-D-G | S-A-C(4+4); S-G(9+0); S-A-D-G(9+0); S-B-E(5+10) |

问题2：给出A\*的图搜索路径，要求给出每一步的扩展结点和和代价值。

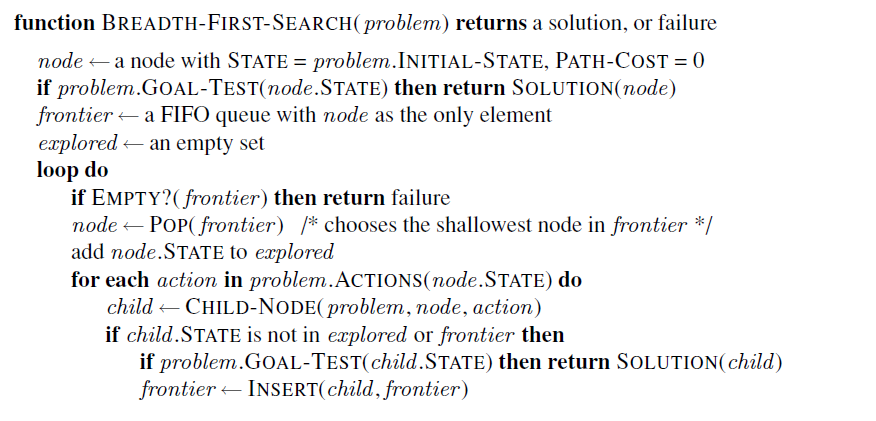


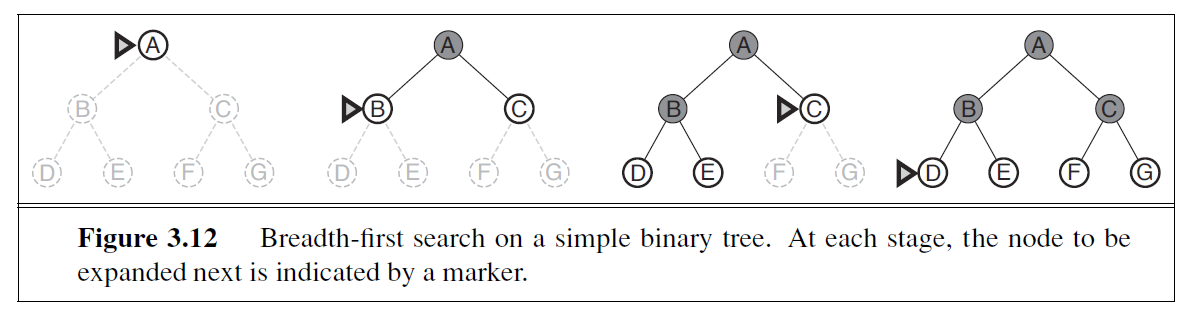
|  |  |  |
| --- | --- | --- |
| 扩展路径 | Explored集合 | 拓展但未经过的路径 |
| S | S | S-A(2+0); S-B(1+6); S-G(9+0) |
| S-A | S A | S-A-D(5+1); S-B(1+6); S-A-C(4+4); S-G(9+0) |
| S-A-D | S A D | S-B(1+6);S-A-C(4+4); S-G(9+0); S-A-D-G(9+0) |
| S-B | S A D B | S-A-C(4+4); S-G(9+0); S-A-D-G(9+0); S-B-E(5+10) |
| S-A-C | S A D B C | S-A-C-G(8+0); S-G(9+0);S-A-D-G(9+0); S-B-E(5+10) |
| S-A-C-G | S A D B C G | S-G(9+0); S-A-D-G(9+0); S-B-E(5+10) |

可见，虽然树搜索重复对节点进行访问，但是所找到的路径确实更优的。

## 无信息搜索算法

### 宽度优先搜索BFS



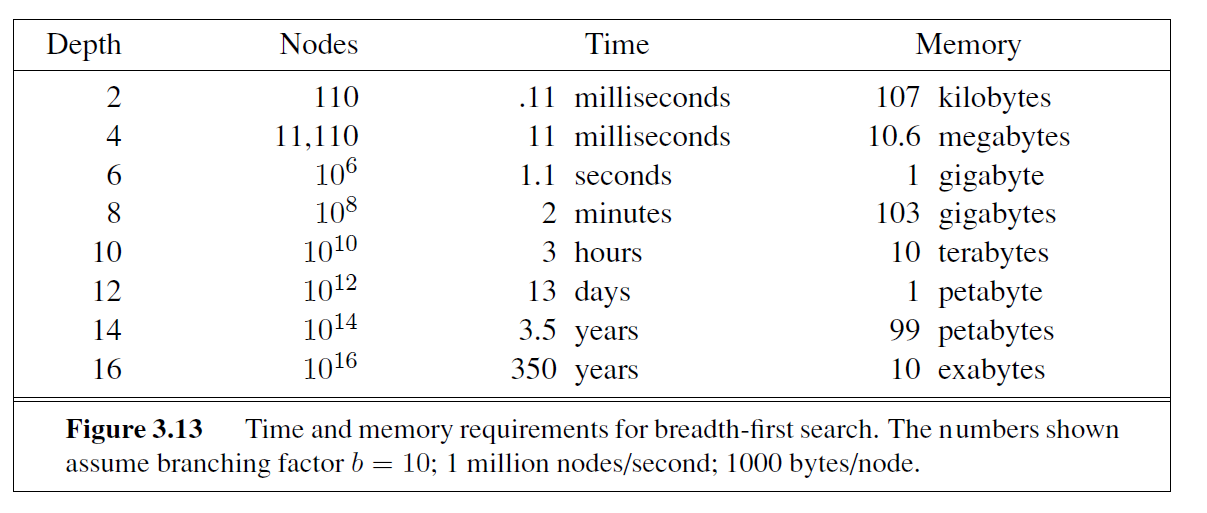


说白了BFS就是用一个FIFO的队列维护frontier中的节点。

我们用BFS的目标是找到目标节点，但是这个从初始节点通向目标节点的路径可能有很多条。我们用BFS的目标是找到目标节点，但是这个从初始节点通向目标节点的路径可能有很多条。很多时候，我们不仅希望找到节点，还希望在搜索的过程中最小化某种指标，这个指标可以用 C(n,a,n’)描述，其中n为节点，n’为下一个节点，a为从节点n到节点n’的路径 。如果满足对任意n，a，n’三元组满足C(n,a,n’)=const>0，可以证明BFS是可以找到最优解的。这个证明是很直观的，只需要一层层的向下找就可以了。往往这样假设的const会是时间，人数等最小单位。注意，如果 C(n,a,n’)const，BFS不保证找到最优解。

因为目标节点在d层，所以BFS的时间复杂度和空间复杂度都是

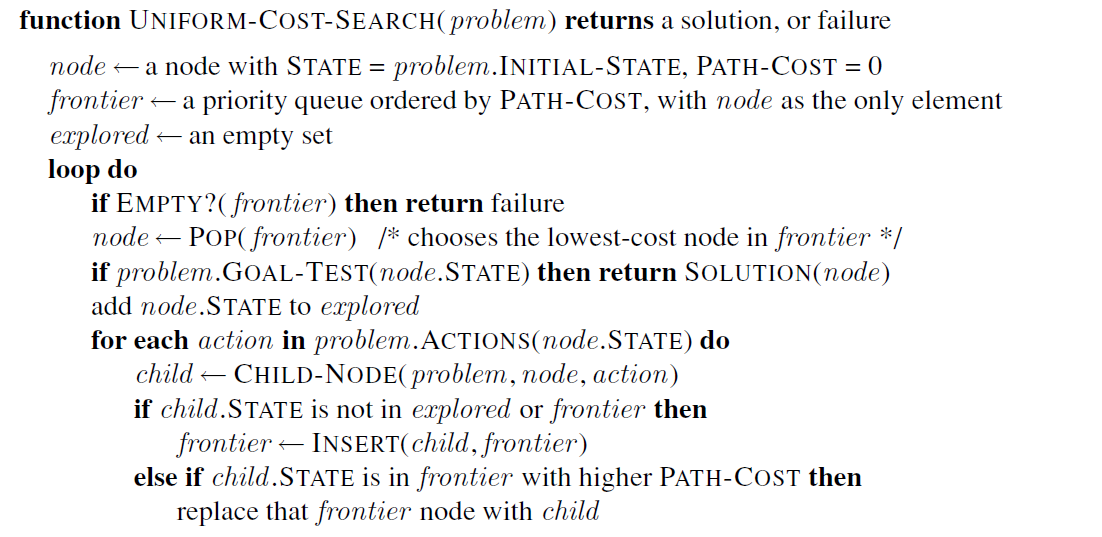




由上图可以看出，对于BFS来说，内存需求和执行时间都是一个大麻烦。一般来说，指数复杂度搜索问题不能应用无信息搜索算法，除了最小路径问题。

### 统一代价搜索UCS

接下来，UCS就是来解决C(n,a,n’)const的问题的。

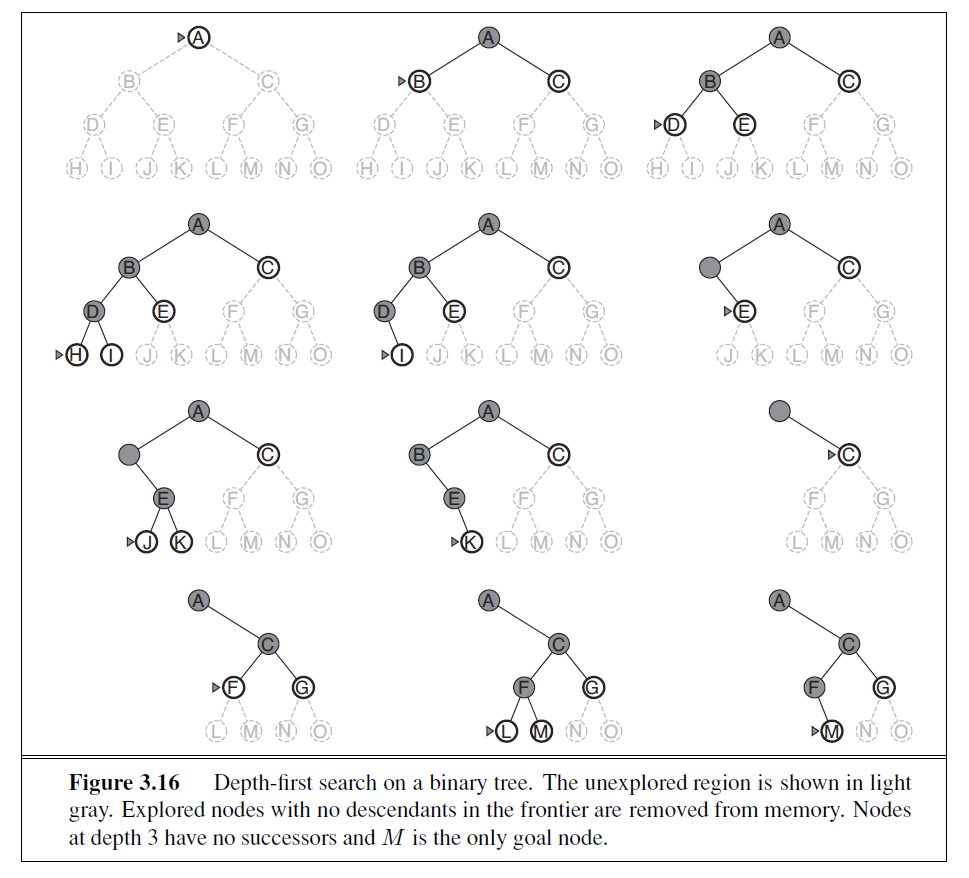


UCS和BFS的主要区别是UCS维护了一个优先队列，通过这个优先队列保证达到目标节点的时候路径一定是最短的。UCS其实就是Dijkstra算法，是从一个顶点到其余各顶点的最短路径算法，解决的是有权图中最短路径问题。

这个也好理解，因为从frontier取出的节点满足chooses the lowest-cost node in frontier，所以如果这个节点是目标节点，那么即便是存在另一个路径也可以通向目标节点，因为当前走过的路径已经比这条路长了，又有C(n,a,n’) >0，所以另一条路径不可能更短。实际上，任何被放入explored的节点都满足最短路径。

### 深度优先搜索DFS

DFS和BFS的区别就是DFS用一个LIFO的队列（其实就是栈）来维护frontier。注意，因为DFS一旦找到了目标节点就会停止，所以当 C(n,a,n’) >0时无法找到最优解。但是，如果 C(n,a,n’) =0 ，任意到达目标节点的解都是最优解，DFS就可以找到最优解。C(n,a,n’) =0 这种情况往往出现在对代价几乎没有要求的情况下，例如不计时间的拼魔方。



值得一提的是，在树搜索算法中，DFS非常容易陷入无穷的循环，所以DFS往往用于图搜索。

因为DFS理论上要搜完整个空间才知道是否有解，所以最大的时间复杂度是 ，但是因为需要记录的节点很少（毕竟是用栈维护），空间复杂度比较低，为  。

### 深度受限搜索DLS

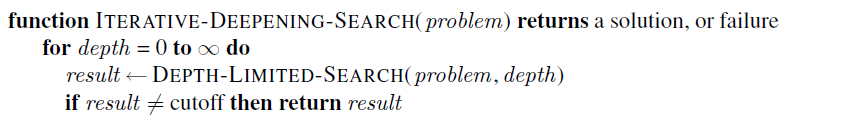
DLS规定了DFS的最大搜索深度，对于一个给定的常数L，DLS在到达这个深度之后就会放弃搜索，转而去探索frontier中的其他节点。

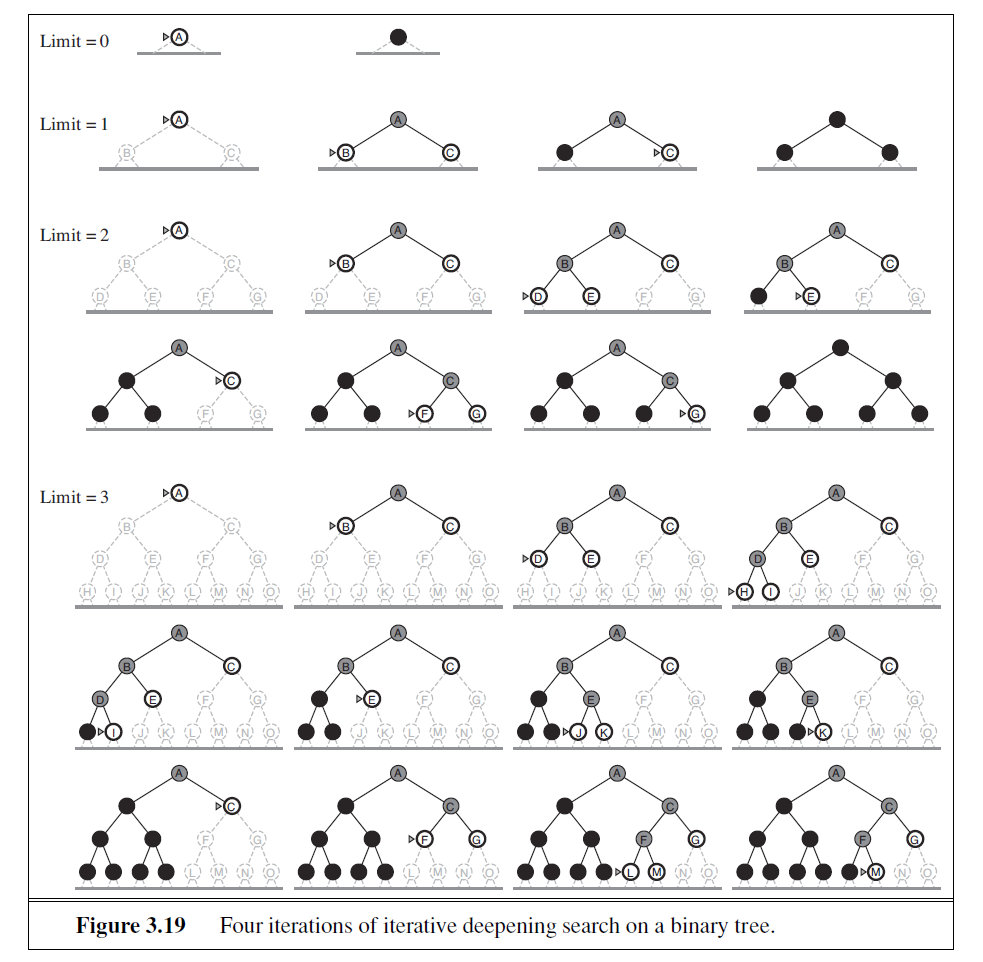


注意DLS是一个递归算法，而之前的算法都是循环的。函数RECURSIVE-DLS可以返回三种值，除了之前的solution和failure，还有递归算法的内部状态cutoff。如果其中一个节点返回cutoff，算法认为到达了受限深度，向上传递cutoff。如果这个节点的所有子节点都不返回solution或者cutoff，也就是说所有节点都返回failure，则向上传递failure。如果最后的跟节点的输出是failure，说明DLS的受限不起作用，也就是说这个图最深的深度也没有受限的深度深。DLS和DFS的复杂度很像，时间和空间复杂度分别是和。

### 迭代加深搜索IDS

IDS 就是在DLS的基础上，不断的将常数L扩大，从而增加搜索深度。



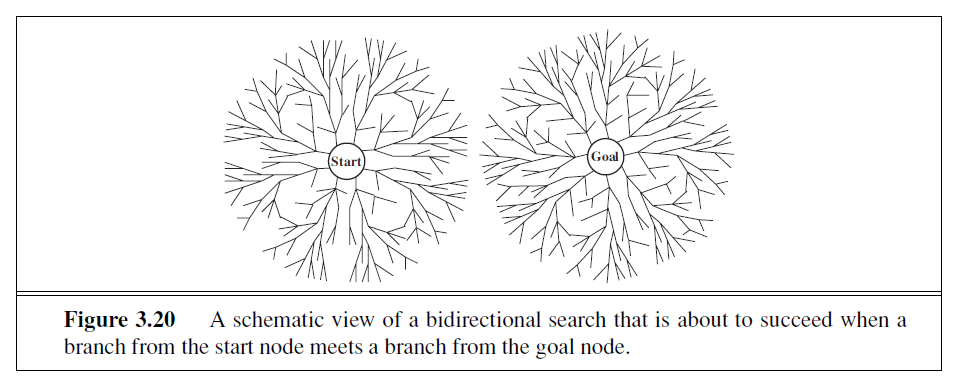


IDS弥补了深度受限搜索的缺点，它允许算法不断的增加深度来寻找目标节点。它和BFS看起来很像，而且IDS似乎在实现上产生了很大的重复，实际上IDS比BFS的空间复杂度小的多，但是时间复杂度相对大一些。

IDS的时间和空间复杂度分别是  和  ，注意 的时间复杂度是相当高的。

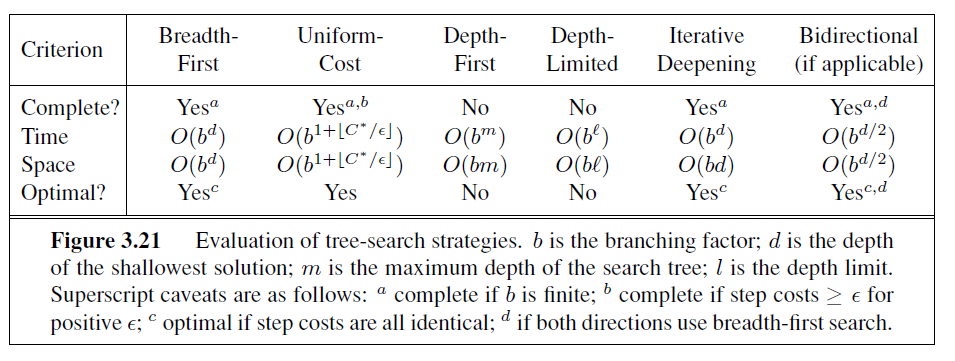
### 双向搜索BS

BS的假设是我们不仅可以从初始节点出发，同时还可以从目标节点出发，从而实现双向的搜索。BS的内部往往是由BFS实现的，如果从初始节点和从目标节点的搜索都是BFS，可以保证算法的最优。实际上，BS就是将BFS画的半径为d的一个大圆，换成了半径为d/2的两个小圆。



因为BS是基于BFS实现的，所以时间复杂度和空间复杂度都很像，都是。

最后比较一些这些算法:



注意，这里DFS不完备是因为可能陷入无穷循环，不满足最优性是因为默认C(n,a,n’) >0。

## 有信息搜索算法

所谓的有信息搜索算法又称启发式搜索，往往假定了我们对搜索问题还有额外的信息。

启发式搜索依赖于evaluation function f（n），f（n）会评估每个节点的代价，代价最小的节点优先被扩展，这与UCS算法很相似。区别在于UCS是根据已经探索过的路径信息给出接下来扩展的节点。而启发式搜索则根据额外的信息。

我们可以定义g（n）为之前探索过的节点的代价总和，那么UCS就相当于f（n）=g（n）。

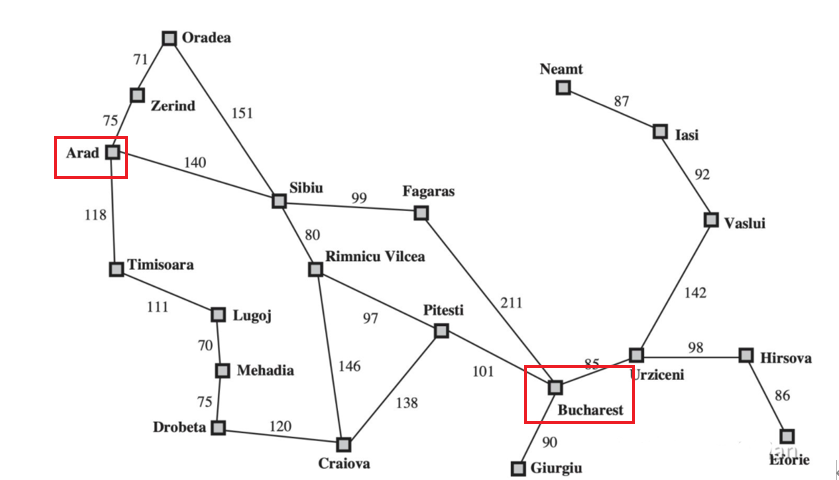
而大多数最优先算法中，我们可以定义一个函数heuristic function h（n）表示：



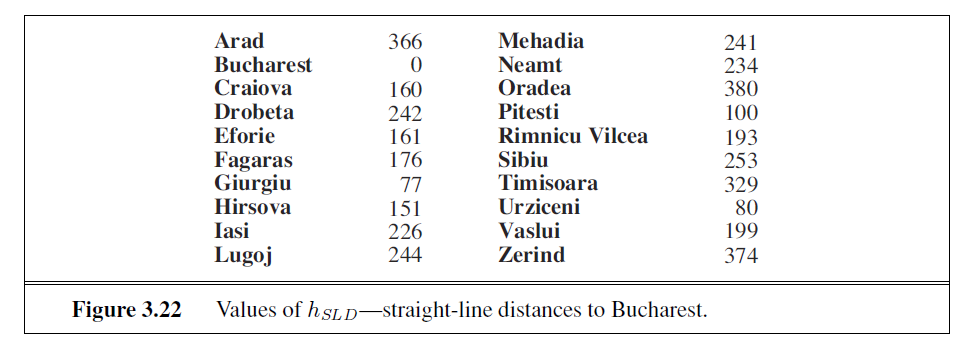
h（n）根据不同策略评估每个节点到目标节点的代价，从中选取代价最小的节点。对于所有h（n）满足h（n）0，且对于目标节点满足h（n）=0.

### 贪心算法Greedy best-first search

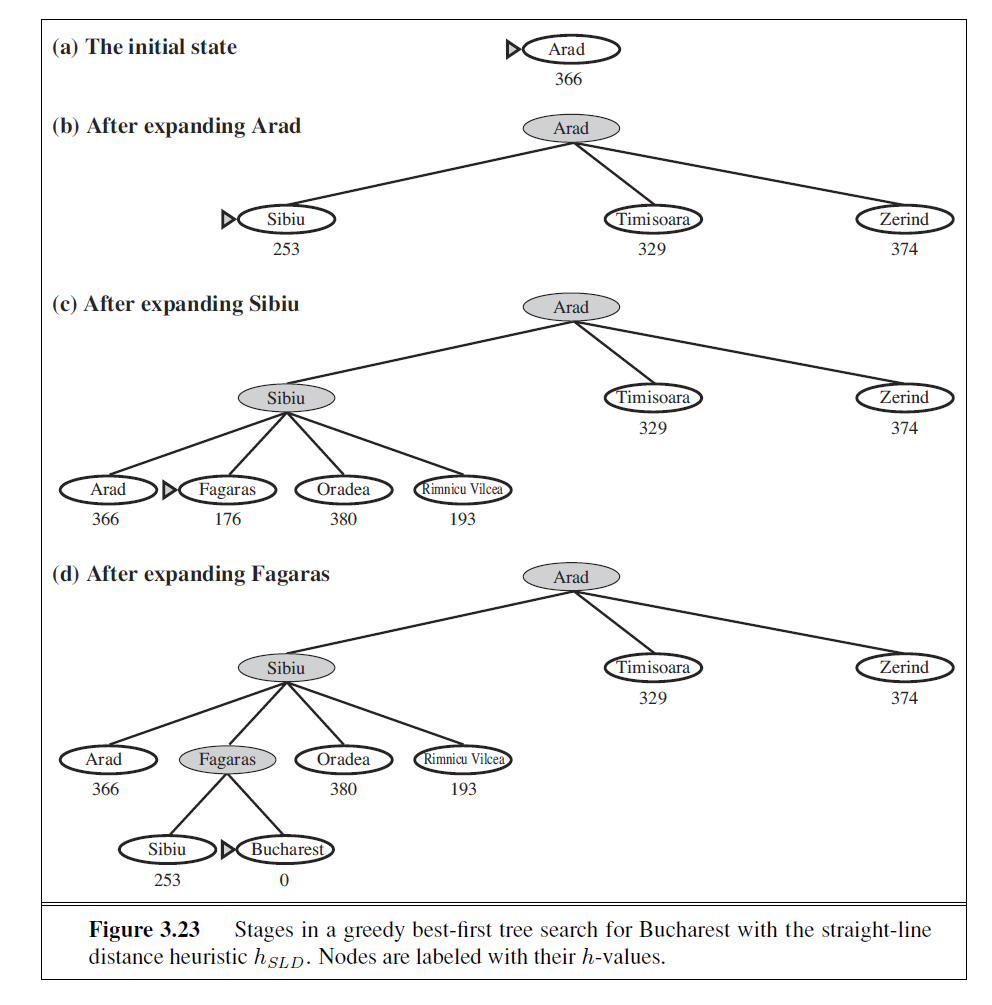
我们假定在下面的地图上搜索，初始节点是Arad，目标节点是Bucharest。



贪心搜索即令f（n）=h（n），即只根据启发进行搜索。假设h（n）计算从当前城市到Bucharest的直线距离。（注意，这个直线距离是原问题所不具备的，因此是“额外的知识”）。各个城市的h（n）为：



根据代价最小，即直线距离最小优先原则进行搜索，过程如下：



但是，贪心搜索不能保证最优，甚至都不能保证完备，因为它可能陷入“死胡同”，例如从Iasi到Fagaras，它会选择去Neamt，然后就卡死了。

### A\* 搜索

为了避免贪心算法不计成本地选择代价最小的节点，更为广泛运用的最优先搜索是A\*搜索，他的evaluate function f（n）=g（n）+h（n），由于g（n）给出了从初始节点到节点n的代价，而h（n）给出了节点n到目标节点的最小代价，故可以知道：



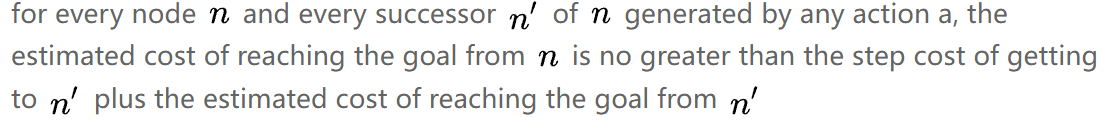
当 h（n）满足Admissibility和consistency这两种性质，A\*算法可以保证找到最优解。

admissible heuristic指的是



显然，在我们的例子中，因为两点之间直线最短，所以admissible是满足的。

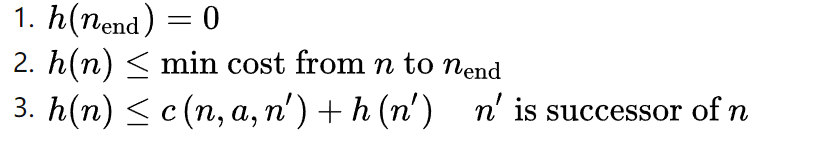
consistent heuristic是指



也就是满足：



其实，算上之前要求的对于目标节点满足 h（n）=0，我们对 h（n） 有三个条件，分别是

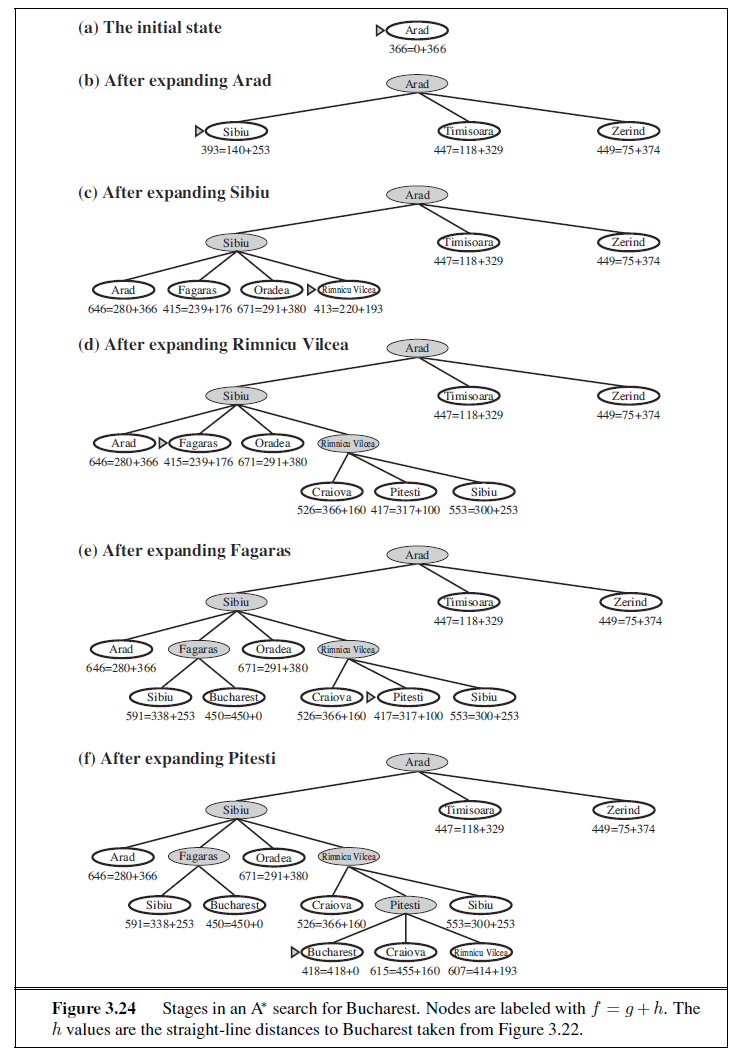


而在CS221中只给了第1个和第3个条件，实际上，我们可以从1，3条件推出2，也可以从2，3条件推出1。1

如果有  和 ，对于的前继必满足，数学归纳得到所有的节点都满足。

如果有2，3条件，显然有.

接下来，只要把UCS的 g（n） 换成 f（n） 就可以使用A\*算法了。



接下来，我们证明A\*算法的最优性。

首先不难看出 f（n） 是不降的：



根据和UCS类似的推导，我们可以说明一旦任意节点n被放入explored，它的路径都满足 f（n） 最小。

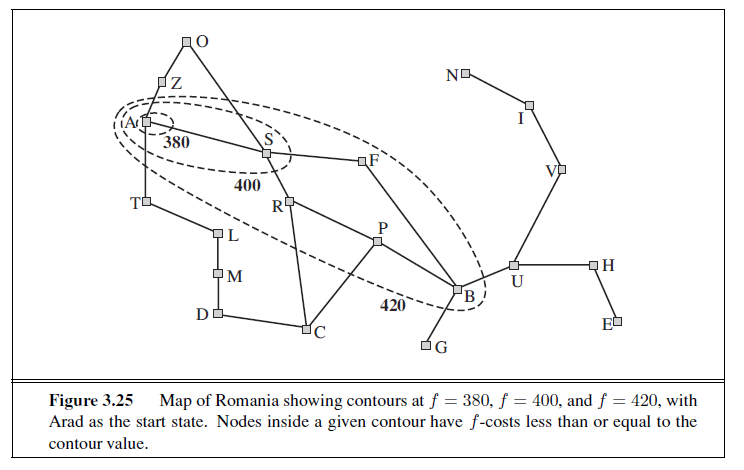
这看似没什么用，因为 f（n） 是我们事先定义的，对于除了目标节点以外的节点，满足 f（n） 最小并不能保证找到了最短路径。

但是目标节点不同，因为,所以,所以目标节点满足g（n）最小，因此我们可以找到通往目标节点的最短路径。

那既然A\*算法满足了最优性，这个性质UCS也可以满足，为什么我们不用UCS而是用A\*呢？

直观的理解，UCS是像圆一样从一个点蔓延，直到找到了目标节点；而A\*则是朝着目标节点的方向蔓延，尽管做不到只走一条路径，但也高效的多。

我们可以用等高线的方式更好的理解这个性质：



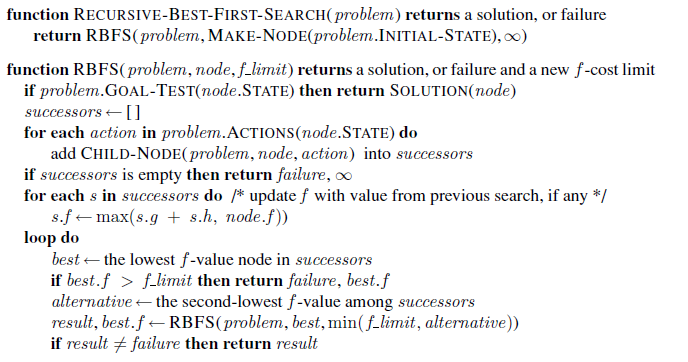
图中的等高线是 f（n）的值，我们设最优路径的代价是C\*的话，因为A\*和UCS的内核是一样的，它会优先探索所有f（n）最小的节点，所以图中所有小于C\*的节点都已经被探索了。

进一步讲，我们知道



从上面的树状图可以看出，这个问题的C\*是418，而 f（n）小于的节点（图中最大的虚线圈内的节点）都已经被探索了。所以A\*的一个问题就是要探索完所有f（n）小于C\*的节点，这些节点的数目可能是指数级的。但无论如何，A\*都要比UCS好，因为UCS要探索完所有 f（n）小于C\*的节点， [公式] 的等高线将是圆，而不是椭圆，这意味这UCS要探索更多。

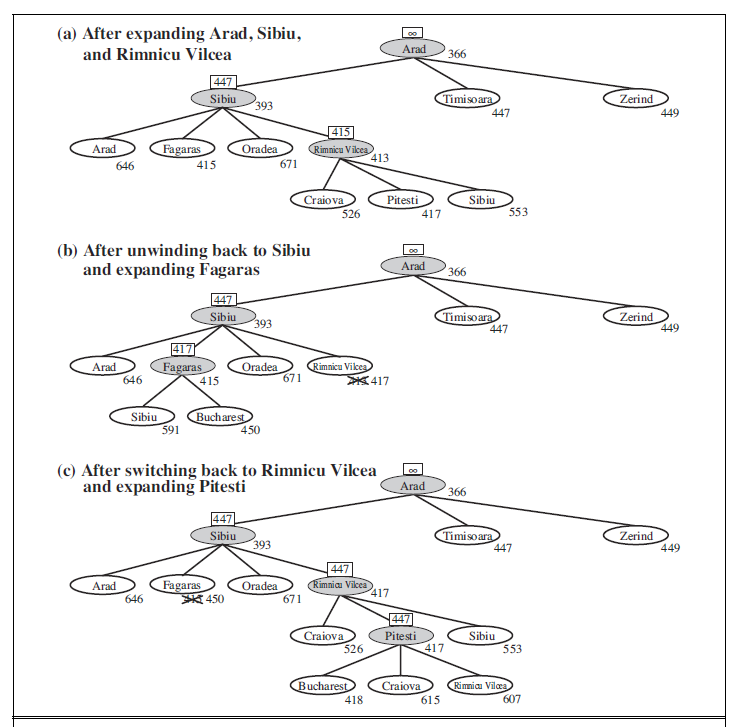
### 迭代最佳优先搜索RBFS



RBFS是一个递归算法，在递归的内部进行了循环。

它维护了一个successor队列，这个队列计算出每个子节点的f（n），选择f（n）最小的作为best节点，同时记录f（n）第二小的节点的代价为alternative，对best节点进行向下搜索，一旦发现代价大于min（f\_limit,alternative）则向上回归(这个过程中不断更新节点的节点代价值),对第二小节点进行递归，进行第二轮扩展，best节点和alternative值不断更新，如此递归搜索下去。

在之前的例子中，RBFS的探索过程如下：

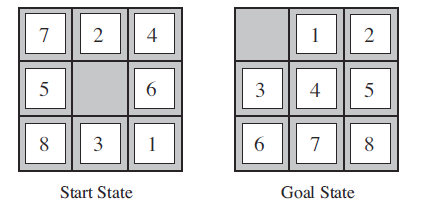


RBFS相比A\*节省了大量的空间，实际上，它的空间复杂度和节点的数目是线性的。但是因为它会在递归的过程中“遗忘”因此时间复杂度更高。

## 启发式设计

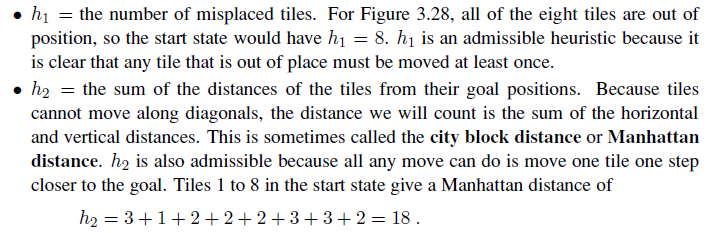
通过刚才A\*和RBFS的例子可以看出来，一个好的 h（n） 对启发式搜索起到了很大的帮助。总的来说，它可以让 f（n）小于C\*的节点的数目变少，也就是让A\*的等高线更窄，让这个椭圆更扁一些，一个极端的情况是这个椭圆只包括从初始状态到目标状态最优路径上的节点。那么，如何设计更好的 h（n） 呢？

以8-puzzle为的例子：



目标是从Start State挪到Goal State。在8-puzzle中，我们可以看到搜索树的宽度最小为2（空格在四角），最大为4（空格在中间），总的来说平均为3。8-puzzle问题的平均移动次数是22次，需要探索  个状态。然而，通过合理的剪枝（停止探索重复状态），这个数目可以减少至181440。但是，即便剪枝，15-puzzle对应的数目依旧达到了 ，我们不得不寻找有效的启发式。

有两种启发式值得参考：

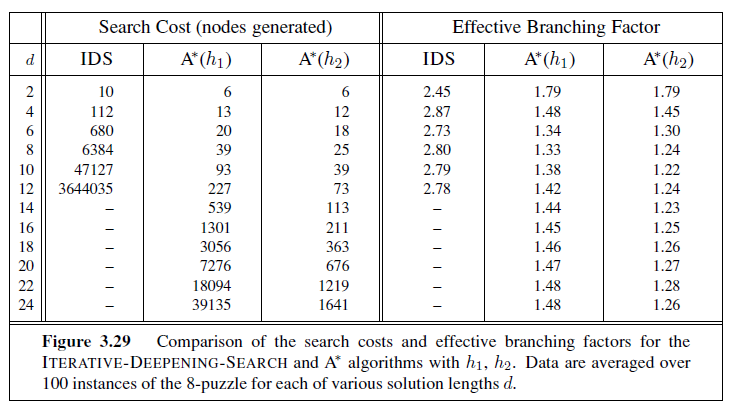


显然，对于所有的 n ，都应该满足h2（n） h1（n）.

我们定义一个用来评价heuristic好坏的指标effective branching factor b\*，它的定义为

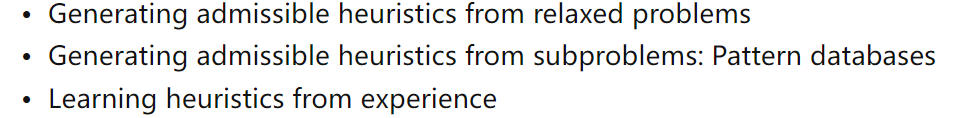


其中 d 是A\*算法搜索的深度， N是算法探索的节点数. 其实，直觉上，我们可以用N表示heuristic的好坏，因为N就是（近似）小于C\*的节点的个数，也就是椭圆里节点的个数。但是因为初始节点和目标节点的不同，椭圆的大小是不好确定的，这时候只统计N就没有意义了，所以我们用深度 d 来计算相对的branching factor。



可以看到h1是优于h2的。这里有一个更一般的结论，因为h2（n） h1（n） 恒成立，所以h2作为heuristic比 h1更好。

生成heuristic主要有三个方法，分别是:



第三个会用到一些机器学习的方法，但是并不能保证heuristic是admissible和consistent的。此外，如果我们有多个heuristic都满足admissible和consistent，我们可以得到新的heuristic。



性能好：因为

admissible： 每个都满足，自然满足.

consistent：即证，其中，

。若结论是显然的；若有。

可以证明，这个新的heuristic性能更好，且满足admissible和consistent。

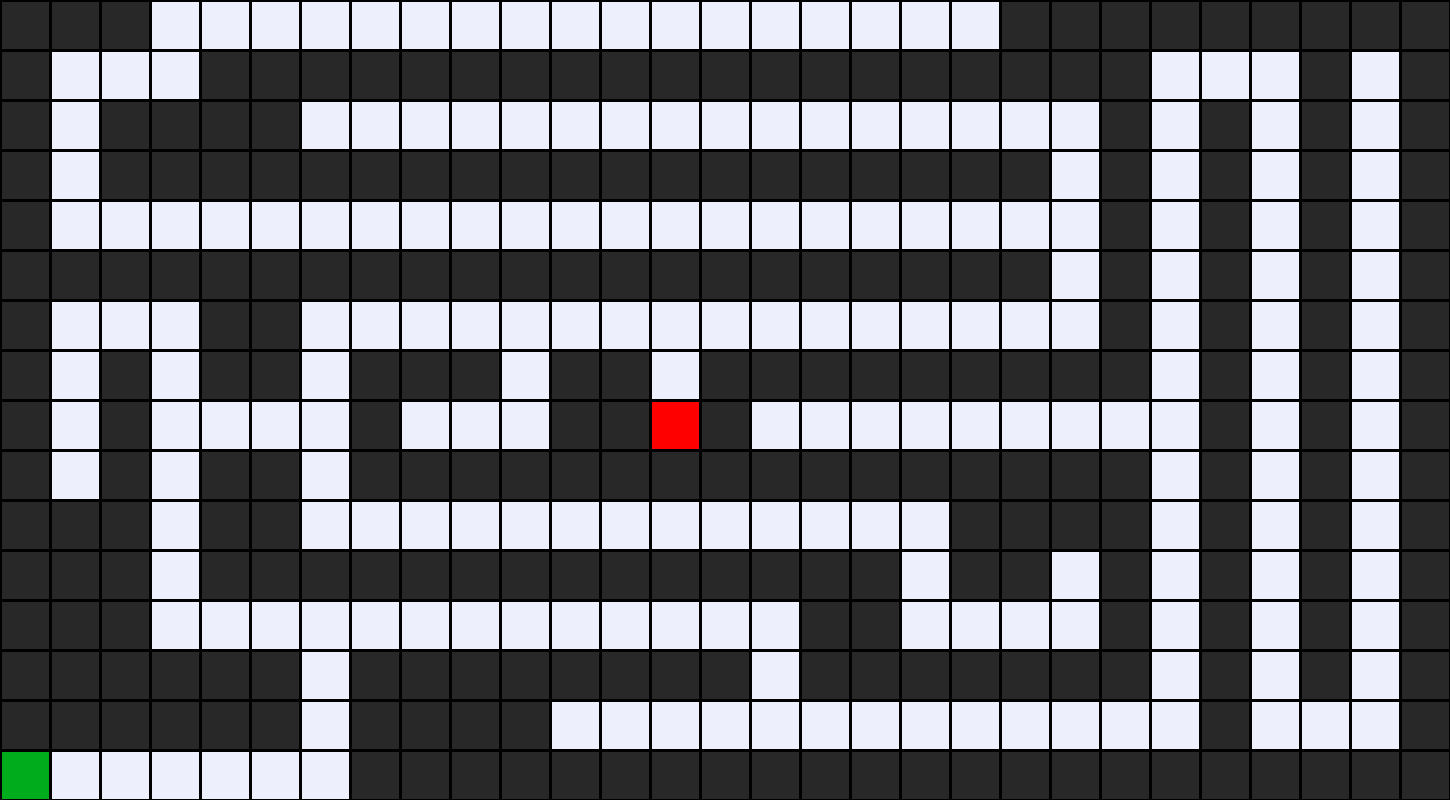
## 习题解答

Exercise 3:

Your goal is to navigate a robot out of a maze. The robot starts in the center of the maze facing north. You can turn the robot to face north, east, south, or west. You can direct the robot to move forward a certain distance, although it will stop before hitting a wall.

1. Formulate this problem. How large is the state space?

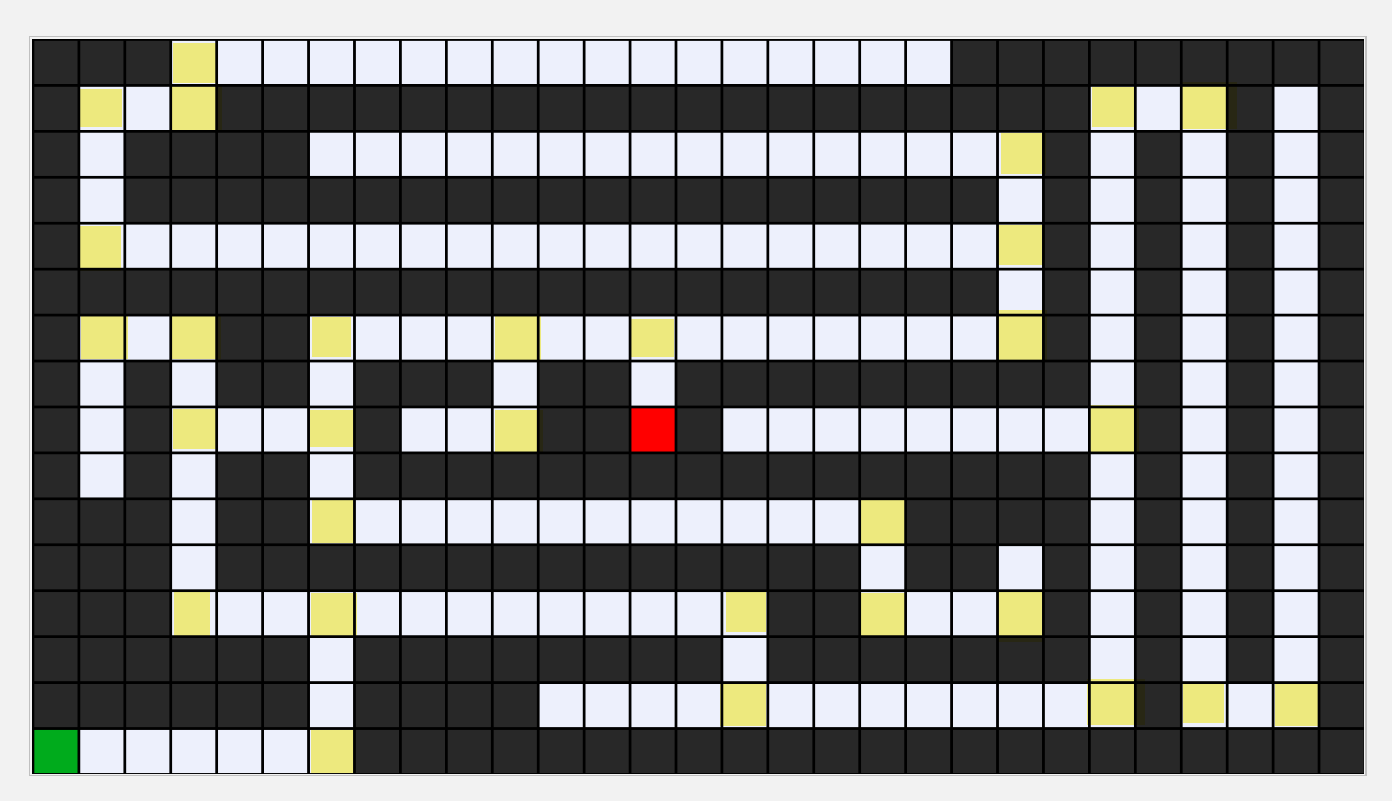
将迷宫整个区域等分为若干区域，每个区域为一个节点的状态，状态标明了位置，初始状态即为起点位置，目标状态就是终点位置，相邻区域可由相对方位标明，这说明节点可以通过动作（如左转弯、右转弯）进行连接，从而划分父节点和子节点（扩展节点）。一下问题分析通过PYTHON实现：



如上图所示，迷宫被划分为16\*29共464个区域，其中黑色方块为围墙，白色为走廊，红色为机器人初始位置，绿色为迷宫出口。黑色区域为不可行位置，即无法到达的状态，故除了黑色区域外，其他区域构成节点状态空间，相邻区域通过动作联系，即东西南北四个方位。故针对此问题模型可得状态空间可通过除黑色外的方块的数量来表示，即state space=205。

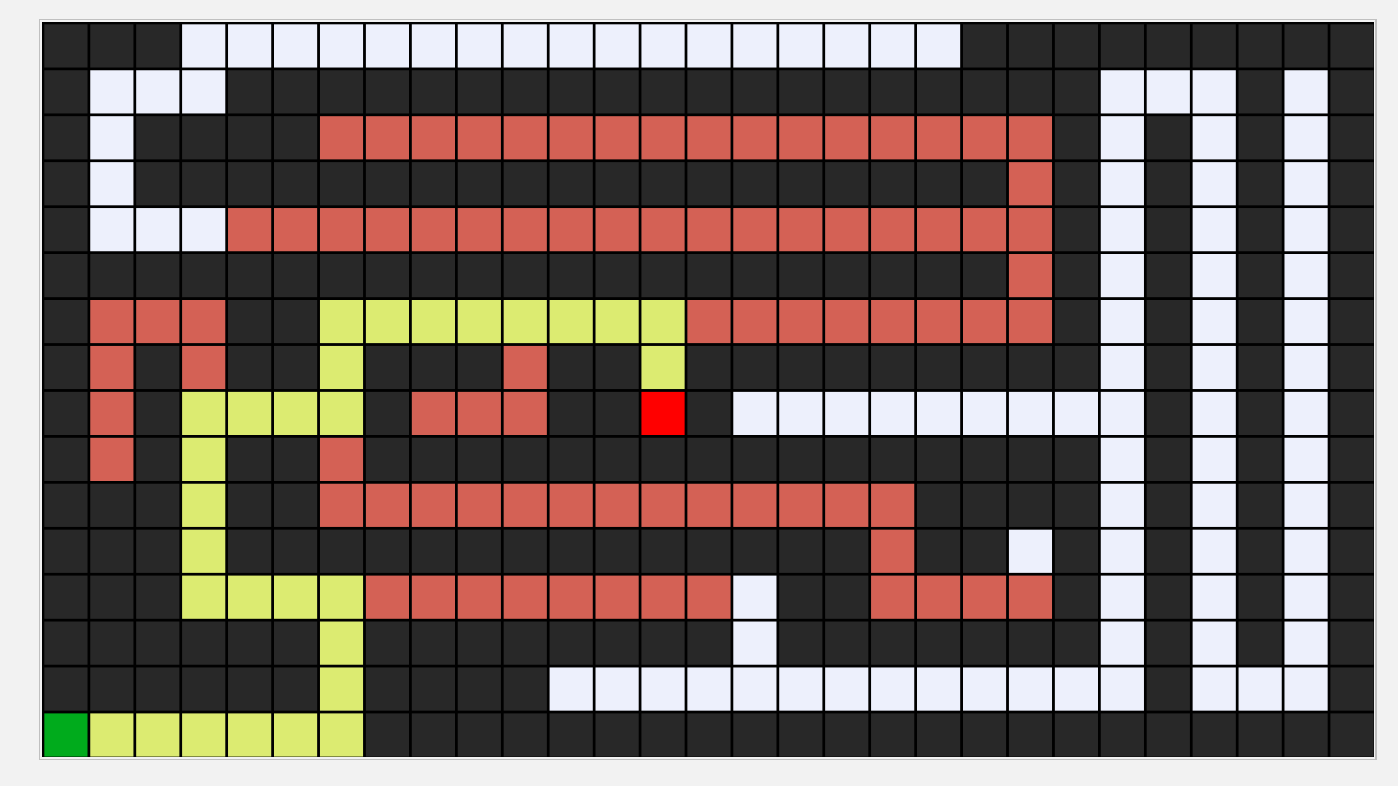
1. In navigating a maze, the only place we need to turn is at the intersection of two or more corridors. Reformulate this problem using this observation. How large is the state space now?

重新对问题进行分析，我们可以发现在迷宫中，只有在两条或更多走廊的交汇处才需要转弯，而在一条走廊里机器人行走的方向只有一个（不考虑后退），故这些只有一个行动方向的区域可以不用考虑，仅仅通过记录转弯的区域就可以得到机器人行走的路径，故状态空间可以缩小为转弯区域的数量，即下图中黄色方块数目，state space=30，空间大大缩小。



1. From each point in the maze, we can move in any of the four directions until we reach a turning point, and this is the only action we need to do. Reformulate the problem using these actions. Do we need to keep track of the robot’s orientation now?

不需要记录机器的行进方向，应为通过父节点和子节点间的状态即可知道机器人行进的方向。比如下图，我们假设左上角为坐标系原点，x轴和y轴方向分别为向右和向下，迷宫被划分为若干行和列，每一个方块由所在列数和行数确定，故每一个方块的位置可以通过坐标（x，y）确定，即节点状态的确定。相邻节点的坐标信息就包含了方位信息，比如父节点坐标（1，1），子节点坐标（1，2），就可以推断出行进方向为向右。



1. In our initial description of the problem we already abstracted from the real world, restricting actions and removing details. List three such simplifications we made.
2. 我们将机器人所在区域的坐标定义为机器人所处节点状态
3. 同时，我们对状态空间进行缩小，只记录机器人在转弯处的信息，即节点状态为处于转弯处的坐标
4. 最后，我们对动作进行简化，因为相邻节点即蕴含了动作的信息，故节点无需记录机器人行进方向。

伪代码如下:

function MAZE-EXIT-SEARCH（exit）returns a path，or failure

node🡨a node with STATE=beginning,PARENT=none

frontier🡨a FIFO queue with node as the only element

explored🡨an empty set

loop do

if EMPTY?(frontier) then return failure

node🡨POP(frontier) /\* chooses the shallowest node in frontier \*/

add node.STATE to explored

neighbor🡨NEIGHBOR(node)

while not EMPTY?( neighbor) do

if neighbor contain exit then return PATH(neighbor,node)

if isINTERSECTION?(neighbor) then

if neighbor.STATE is not in explored or frontier then

frontier🡨INSERT(neighbor,frontier)

neighbor🡨NEIGHBOR(neighbor)

end