# 搬运工问题的启示

重庆外语学校 刘汝佳

# 一 状态空间搜索基本知识

## 1.状态空间(state space)

对于一个实际的问题，我们可以把它进行一定的抽象。通俗的说，状态(state)是对问题在某一时刻的进展情况的数学描述，状态转移(state-transition)就是问题从一种状态转移到另一种（或几种）状态的操作。如果只有一个智能体(Agent)可以实施这种状态转移，则我们的目的是单一的，也就是从确定的起始状态(start state)经过一系列状态转移而到达一个（或多个）目标状态(goal state)。

如果不止一个智能体可以操纵状态转移(例如下棋)，那么它们可能会朝不同的，甚至是对立的目标进行状态转移。这样的题目不在本文讨论范围之内。

我们知道，搜索的过程实际是在遍历一个隐式图，它的结点是所有的状态，有向边对应于状态转移。一个可行解就是一条从起始结点出发到目标状态集中任意一个结点的路径。这个图称为状态空间(state space)，这样的搜索就是状态空间搜索(Single-Agent Search)

## 2.盲目搜索(Uninformed Search)

盲目搜索主要包括以下几种：

### 纯随机搜索(Random Generation and Random Walk)

听起来比较“傻”，但是当深度很大，可行解比较多，解的深度又不重要的时候还是有用的，而且改进后的随机搜索可以对付解分布比较有规律（相对密集或平均，或按黄金分割比例分布等）的题目。一个典型的例子是：你在慌乱中找东西的时候，往往都是进行随机搜索。

### 广度优先搜索(BFS)和深度优先搜索(DFS)

大家都很熟悉它们的时间效率，空间效率和特点了吧。广度优先搜索的例子是你的眼镜掉在地上以后，你趴在地板上找：）- 你总是先摸最接近你的地方，如果没有，在摸远一点的地方…深度优先搜索的典型例子是走迷宫。它们还有逆向和双向的搜索方式，但是不再本文讨论范围之内。

### 重复式搜索

这些搜索通过对搜索树扩展式做一些限制，用逐步放宽条件的方式进行重复

搜索。这些方法包括：

### 重复式深度优先(Iterative Deepening)

限制搜索树的最大深度 Dmax,然后进行搜索。如果没有解就加大 Dmax 再搜索。虽然这样进行了很多重复工作，但是因为搜索的工作量与深度成指数关系，因此上一次（重复的）工作量比起当前的搜索量来是比较小的。这种方法适合搜索树总的来说又宽又深，但是可行解却不是很深的题目（一般的深度优先可能陷入很深的又没有解的地方，广度优先的话空间又不够）

### 重复式广度优先(Iterative Broadening)

它限制的是从一个结点扩展出来的子节点的最大值 Bmax,但是因为优点不是很明显，应用并不多，研究得也比较少。

### 柱型搜索(Beam Search)

它限制的是每层搜索树节点总数的最大值 Wmax。显然这样搜索树大小与深度成正比，但是可能错过很接近起点的解，而增加 Wmax 的时候保留哪些节点，Wmax 增加多少是当前正在研究的问题。

## 3.启发式搜索(Informed Search)

我们觉得一些问题很有“想头”，主要是因为启发信息比较多，思考起来容易入手，但是却不容易找到解。我们不愿意手工一个一个盲目的试验，同样也不愿意我们的程序机械的搜索。也就是说，我们希望尽可能的挖掘题目自身的特点，让搜索智能化。下面介绍的启发式搜索就是这样的一种智能化搜索方法。

在刚才的那些算法中，我们没有利用状态本身的信息，只是利用了状态转移来进行搜索。事实上，我们自己在解决问题的时候常常会估计状态离目标到底有多接近，进而对多种方案进行选择。把这种方法用到搜索中来，我们可以用一个状态的估价函数来估计它到目标状态的距离。这个估价函数是和问题息息相关的，体现了一定的智能。为了以后叙述方便，我们先介绍一些记号：

| | |
|---|---|
| S | 问题的任何一种状态 |
| H*(s) | s 到目标的实际（最短）距离 – 可惜事先不知道:） |
| H(s) | s 的启发函数 – s 到目标距离的下界，也就是 h(s)<=h*(s),如果 h 函数对任意状态 s1 和 s2,还满足 h(s1)<=h(s2)+c(s1,s2)（其中 c(s1,s2)代表状态 s1 转移到 s2 的代价）,也就是状态转移时，下界 h 的减少值最多等于状态转移的实际代价，我们说 h 函数是相容(consistent)的。(其实就是要求 h 不能减少得太快) |
| G(s) | 到达 s 状态之前的代价，一般就采用 s 在搜索树中的深度。 |
| F(s) | s 的估价函数，也就是到达目标的总代价的估计。直观上，应该有 f(s)=g(s)+h(s)，即已经付出的和将要付出的代价之和。如果 g 是相容的，对于 s1 和它的后辈节点，有 h(s1)<=h(s2)+c(s1,s2) 两边同时加上 g(s1),有 h(s1)+g(s1)<=h(s2)+g(s1)+c(s1,s2),也就是 f(s1)<=f(s2)。因此 f 函数单调递增。 |

表 1　启发式搜索用到的符号

## 贪心搜索(Best-First Search)

象广度优先搜索一样用一个队列储存待扩展，但是按照 h 函数值从小到大排序(其实就是优先队列)。显然由于 h 估计的不精确性，贪心搜索不能保证得到的第一个解最优，而且可能很久都找不到一个解。

## A*算法

和贪心搜索很类似，不过是按照 f 函数值进行排序。但是这样会多出一个问题：新生成的状态可能已经遇到过了的。为什么会这样呢？由于贪心搜索是按照 h 函数值排序，而 h 只与状态有关，因此不会出现重复，而 f 值不仅状态有关，还与状态转移到 s 的方式有关，因此可能出现同一个状态有不同的 f 值。解决方式也很简单，如果新状态 s1 与已经遇到的状态 s2 相同，保留 f 值比较小的一个就可以了。(如果 s2 是待扩展结点，是有可能出现 f(s2)>f(s1)的情况的，只有已扩展结点才保证 f 值递增)。A*算法保证得到最优解，但是所用的空间是很大的，难以适应我们的搬运工问题。

## IDA*算法

既然 A*算法存在空间问题，那么我们能不能借用深度优先搜索的空间优势，用重复式搜索的方式来缓解危机呢？经过研究，Korf 于 1985 年提出了一个 Iternative Deepening A*(IDA*)算法，比较好的解决了这一问题。一开始，我们把深度最大值 Dmax 设为起始结点的 h 值，开始进行深度优先搜索，忽略所有 f 值大于 Dmax 的结点，减少了很多搜索量。如果没有解，再加大 Dmax 的值，直到找到一个解。容易证明这个解一定是最优的。由于改成了深度优先的方式，与 A*比较起来，IDA*更加实用：

1. 不需要判重，不需要排序，只用栈就可以了。操作简单。
2. 空间需求大大减少，与搜索树大小成对数关系。

## 其他的启发式搜索

这些方法包括深度优先+最优剪枝式的 A*，双向 A*,但是由于很不成熟或者用处并不大，这里就不介绍了。A*算法有一个加权的形式，由于在搬运工问题中效果不明显，这里从略。