《人工智能》课程系列

启发式搜索技术*

武汉纺织大学数学与计算机学院

杜小勤

2018/10/08

Contents

*本系列文档属于讲义性质，仅用于学习目的。Last updated on: October 23, 2019。

1 概述

启发式搜索 (Heuristic Search)，又称为有信息搜索 (Informed Search)，与无信息的搜索 (或基本的搜索) 方式相比，它能够更加有效地求解问题。原因在于，这种搜索方式优先扩展代价最低的节点，而这些节点被认为最有希望导向目标。

此类算法的最基本形式是最佳优先搜索 (Best-First Search)。就算法的形式而言，最佳优先搜索与一致代价搜索 (UCS 算法) 是类似的——它们都需要依据某种代价值对节点进行排序，并优先扩展那些代价低的节点。

因此，算法需要对节点的代价进行评估，我们把执行此任务的函数称为评价函数或代价函数 $f(n)$。

一般情况下，代价函数 $f(n)$ 具有如下形式：

$$f(n) = g(n) + h(n) \tag{1}$$

其中，n 是被评价的节点，$g(n)$ 表示从初始节点到节点 n 的路径代价 (实际代价)，而 $h(n)$ 表示从节点 n 到目标节点的代价估计值。

实际上，从代价函数的观点看，可以将前述的各种搜索算法统一起来：不同的 $f(n)$ 决定了不同的搜索策略。例如，如果让 $f(n) = g(n) = -depth(n)$，其中 $h(n) \equiv 0$，那么该搜索策略执行深度优先搜索；如果让 $f(n) = g(n)$，其中 $h(n) \equiv 0$，那么该搜索策略执行一致代价搜索，即一致代价搜索也是最佳优先搜索的特例；如果在一致代价搜索中，进一步让 $f(n) = g(n) = depth(n)$，则该策略执行宽度优先搜索，即宽度优先搜索也是一致代价搜索的特例。另外，一致代价搜索也是 A* 搜索算法的特例，此时 $h(n) \equiv 0$[1]。可见，代价函数 $f(n)$ 对算法性能的影响很大[2]。

$h(n)$ 被称为启发函数 (Heuristic Function)，它是一个估算函数——估算节点 n 到目标节点的代价。如果一个搜索算法利用了启发函数 $h(n)$，我们就称该算法是启发式搜索算法，否则，称它为无信息搜索算法。

[1]实际上，A* 算法也是最佳优先搜索的特例，下文将给出 A* 搜索算法。$h(n)$ 总是满足 A* 算法的 “可接受”(Admissible，定义见后文) 条件，即 $h(n) \equiv 0 \leq h^*(n)$，其中，$h^*$ 表示节点 n 到目标的最短代价。

[2]同样，下文也将表明，启发函数 $h(n)$ 对启发式搜索算法的影响也是很大的。

以后，我们将假设启发式函数始终是非负的，并且约定，当 n 是目标节点时，$h(n) = 0$。

下面，将首先给出最佳优先搜索算法的一般形式，然后讨论有效利用启发函数的搜索算法。

2 最佳优先搜索算法

从上面的分析可知，最佳优先搜索算法是各类搜索算法的最一般形式——无论是无信息搜索算法，还是启发式搜索算法。

下面给出最佳优先搜索算法的一般形式：

```
Input:
    A graph G and a vertex v of G
    came_from is a DICT, initialized with {}
Output:
    return: GOAL or None
    came_from is changed
def BestFS(G, v, came_from):
    frontier = PriorityQueue()
    cost_so_far = {}
    frontier.enqueue(v, 0)
    cost_so_far[v] = 0
    came_from[v] = None
    while not frontier.is_empty():
        v = frontier.dequeue()
        if v is a goal:
            return v
        else:
            for all edges from v to w in G.adjacentEdges(v):
                new_cost = cost_so_far[v] + G.cost(v, w)
                if w not in cost_so_far or new_cost < cost_so_far[w]:
                    cost_so_far[w] = new_cost
```

```
                priority = new_cost + heuristics(goal, w)
                frontier.put(w, priority)
                came_from[w] = v
    return None
```

注：可以把 frontier.put 理解为“如果该项已经存在，则用新的优先级替换；否则，按优先级插入该项”[3]。

3 贪心最佳优先搜索

贪心最佳优先搜索 (Greedy Best-First Search, GBFS) 只使用启发函数 $h(n)$，即 $f(n) = h(n)$，而 $g(n) \equiv 0$。它的基本思想是，总是试图扩展那些离目标“更近”的节点[4]。

如图3-1所示，展示了罗马尼亚地图的一个简化版本。现在，考虑将此算法应用于罗马尼亚地图的路径搜索问题中。

设启发函数 $h(n)$ 为节点之间的直线距离，并假设目的地为 Bucharest，图3-2列出了每个节点到目标节点 Bucharest 的直线距离。

现在，假设需要搜索节点 Arad 到 Bucharest 的一条路径，那么本算法得到的结果将是：Arad→Sibiu→Fagaras→Bucharest。对于此例，贪心最佳搜索算法在没有扩展任何多余节点的情况下就找到了目标节点。这表明，此时的搜索代价是最小的。但是，搜索到的路径却不是最优的。因为路径 Arad→Sibiu→Fagaras→Bucharest 的总距离为 450，而路径 Arad→Sibiu→Rimnicu Vilcea→Pitesti→Bucharest 的总距离为 418。

贪心算法的名称由此而来，它的每一步都试图扩展离目标“最近”的节点。

[3]在实际实现时，如果明确知道问题有解，可以简单地使用 frontier.enqueue 方法替换 put 方法，即无论存在与否，均按优先级插入该项。这种变形的处理方法，不会影响算法的功能实现。原因在于，即使该项已经存在，新插入项的优先级高，必将优先扩展。另外，算法在找到一个目标后会停止运行，即使 frontier 优先级队列里还包含节点。

[4]离目标的远近与启发函数 $h(n)$ 的定义有关。在一些情况下，离目标“更近”的节点未必真的离目标更近。例如，考虑迷宫问题，假设在某个节点与目标之间存在障碍物——如果将 $h(n)$ 定义成节点到目标的直线距离，那么此时，即便该节点到目标的直线距离最短，但并不是当前的最佳扩展节点。

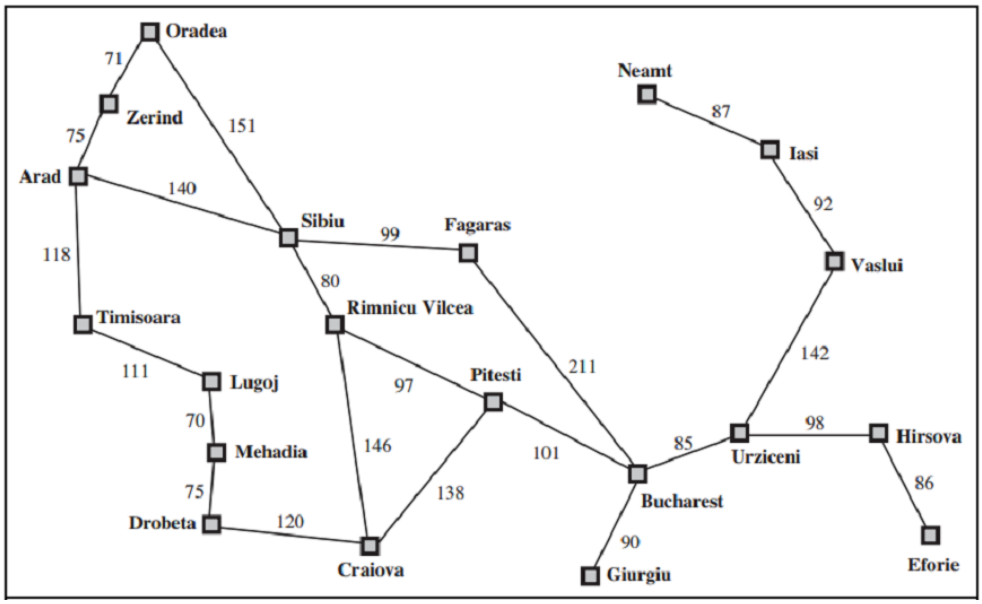

图 3-1: 罗马尼亚地图的简化版

Arad	366	**Mehadia**	241
Bucharest	0	**Neamt**	234
Craiova	160	**Oradea**	380
Drobeta	242	**Pitesti**	100
Eforie	161	**Rimnicu Vilcea**	193
Fagaras	176	**Sibiu**	253
Giurgiu	77	**Timisoara**	329
Hirsova	151	**Urziceni**	80
Iasi	226	**Vaslui**	199
Lugoj	244	**Zerind**	374

图 3-2: 节点到目标 Bucharest 的直线距离

在不记录已访问节点的情况下，贪心最佳优先搜索算法也不是完备的。例如，假设搜索从 Iasi 到 Fagaras 的一条路径，算法将会在 Iasi↔Neamt 之间来回循环。

在最坏情况下，算法的时间复杂度与空间复杂度都是 $O(b^m)$，其中 b 是分支因子，m 是搜索的最大深度。

4 A* 搜索

4.1 启发函数的约束

A* 搜索算法是最佳优先搜索算法中的代表算法，它既考虑了实际代价函数 $g(n)$，也考虑了启发函数 $h(n)$(并对启发函数做了某种限制，以确保算法的完备性与最优性，下文将会讨论这种限制)，即 $f(n) = g(n) + h(n)$。

A* 算法的基本思想是，由于 $g(n)$ 表示开始节点到节点 n 的实际路径代价，而 $h(n)$ 表示节点 n 到目标节点的最小路径代价的估计值，那么 $f(n)$ 就表示从开始节点到目标节点且经过节点 n 的最小代价解的估计代价；因此，如果需要找到从开始节点到目标节点的最小代价解，那么首先扩展 $f(n) = g(n) + h(n)$ 值最小的节点就是合理有效的。

为确保 A* 算法能够搜索到最优解，必须对启发函数 $h(n)$ 施加下面 2 个限制：

1. $h(n)$ 必须是可接受的 (Admissible) 或具有可接受性 (Admissibility)

   可接受性指的是 $h(n)$ 从来不会过高估计节点 n 到达目标节点的代价。设 $h^*(n)$ 表示节点 n 到目标节点的实际最小代价，则可接受性意味着 $h(n) \leq h^*(n)$。这就意味着 $f(n) = g(n) + h(n)$ 也从来不会超过经过节点 n 的解的实际代价。

   实际上，可接受的启发函数属于一种乐观的估值函数，因为它预估的问题求解代价不超过问题求解的实际代价。

   在罗马尼亚路径搜索的例子中，$h(n)$ 计算两点之间的直线距离。显然，该 $h(n)$ 是可接受的启发函数，因为两点之间的距离最短，$h(n)$ 肯定不会超过实际代价 $h^*(n)$。

2. $h(n)$ 必须是一致的 (Consistent) 或具有一致性 (Consistency) 或具有单调性 (Monotonicity)

该约束条件略强于可接受条件，只对基于图搜索的 A* 搜索算法进行限制(原因详见4.1脚注)。下文将证明，单调一致的启发函数一定是可接受的。

一致性指的是，对于每个节点 n 及其任一后继节点 n'，满足下面的不等式关系：

$$h(n) \leq cost(n, n') + h(n') \tag{2}$$

其中，$cost(n, n')$ 表示节点 n 到后继节点 n' 的单步代价。此外，对于任一目标节点 g，有 $h(g) = 0$。

直觉上看，一致性条件必然包含着可接受条件：对于任一目标节点，$h(g) = 0$，从节点 n_i 到该目标节点的最小代价为 $h^*(n_i)$，它是由一段段的单步代价相加而成的，而一致性条件就确保了 $h(n_i) \leq cost(n_i, n_{i+1}) + cost(n_{i+1}, n_{i+2}) + ... + cost(n_k, g) + h(g) = h^*(n)$，即可接受条件亦成立。

正式地，可以采用数学归纳法证明“单调一致的启发函数一定是可接受的”。设 n_k 是目标节点的父节点，依据单调一致条件，$h(n_k) \leq cost(n_k, g) + h(g) = h^*(n_k)$ 成立。现假设该最短路径上的任一节点 n_i，$h(n_i) \leq h^*(n_i)$ 成立，可以推出：

$$h(n_{i-1}) \leq cost(n_{i-1}, n_i) + h(n_i) \leq cost(n_{i-1}, n_i) + h^*(n_i) = h^*(n_{i-1}) \tag{3}$$

其中，n_{i-1} 表示节点 n_i 的父节点。

但是，反过来，可接受条件不一定意味着单调一致条件的成立。不过，可以利用 pathmax 公式将可接受的启发函数转换成单调一致的启发函数：

$$h'(p) \leftarrow max(h(p), h(n) - cost(n, p)) \tag{4}$$

实际上，可以证明 (见下文)，如果 $h(n)$ 是可接受的且问题有解时，那么 A* 算法的树搜索版本是最优的；而如果 $h(n)$ 是单调一致的且问题有解时，那么 A* 算法的图搜索版本是最优的[5]。

[5]在文献 [1] 中，搜索算法区分为树搜索版本与图搜索版本。前者没有添加 CLOSED 表，允许节点重复访问；后者添加了 CLOSED 表，不允许节点的重复访问——如果节点既没有在 OPEN 表中，也没有在 CLOSED 表中，才被允许添加到 OPEN 表中。通俗地讲，树搜索版本是遗忘搜索历史的算法，而图搜索版本是记忆搜索历史的算法。参见文献 [1]P69-70。下文将表明，如果使用单调一致的启发函数，一旦一个节点被扩展，那么它的代价已经是最低的，意味着该节点不会再被扩展，这正适用于图搜索版本的 A* 算法。

4.2 A* 算法的完备性

为确保 A* 算法的完备性，需要对状态空间做出如下 2 个弱假设[6]：

- 每个节点的后继子节点个数是有限的；
- 状态空间的单步代价是非零代价 (大于某个非常小的阈值 $\epsilon > 0$)；

下面将证明，当问题有解时，A* 算法具有完备性，即一定能够找到解。

设 f^* 表示最优解的最小代价，如果 A* 算法不能找到目标状态，表明在通往最优解的路径上，有无限多个节点 n，它们满足条件 $f(n) = g(n) + h(n) \leq f^*$。这种情况只有在如下情形下才会出现：

- 节点存在无限多个后继子节点

  上述约束条件的第 1 条，可以排除此种情形；

- 搜索陷入了一条有无限多个节点而每个节点的代价满足条件 $f(n) \leq f^*$ 的路径

  考虑约束条件的第 2 条，即单步代价大于 $\epsilon(> 0)$。因此，在该约束条件下，即使 ϵ 的值很小，但是在有无限多个这类节点的情况下，每搜索一步，它的累加代价 $g(n)$ 还是会慢慢变大，从而导致 $f(n) > f^*$，这与假设矛盾。因此，此种情形也不存在。

综上所述，当问题有解时，A* 算法一定能够找到解，即该算法具有完备性。

4.3 A* 算法的最优性

当问题有解时，A* 算法一定能够找到一条到达目标节点的最佳路径，即 A* 算法具有完备性与最优性。

实际上，前述的宽度优先搜索与一致代价搜索算法都是 A* 算法的特例，都具有最优性与完备性[7]。这 2 个特例也验证了 A* 算法的上述结论。需要注意的是，

[6]对于绝大多数的搜索任务而言，这 2 个假设非常弱。例如，在一致代价搜索中，也要求状态空间的单步代价是非零代价 (大于某个非常小的阈值 $\epsilon > 0$)。

[7]对于一致代价搜索算法而言，在满足每步非零代价的条件时，算法才是完备的。但是，该条件非常弱，一般情况下，容易得到满足。

即使算法具有最优性，其搜索效率不一定高，它取决于被访问的节点个数[8]。此外，一些算法还存在反复扩展与访问相同节点的情况。

下面首先证明，如果 $h(n)$ 满足可接受条件且问题有解时，那么 A* 算法的树搜索版本一定是最优的。

设 f^* 表示最优解的最小代价，$goal$ 表示次优解的目标节点。由于 $goal$ 是目标节点，$h(goal) = 0$ 成立，且 $f(goal) = g(goal) + h(goal) = g(goal) > f^*$。

另一方面，如果问题有解，那么在最优解的路径上，必定存在某个节点 n，由于 $h(n)$ 是可接受的，则有 $f(n) = g(n) + h(n) \leq f^*$。

综合上述 2 个结论，可知：$f(n) \leq f^* < f(goal)$，这表明节点 $goal$ 没有机会得到扩展，A* 一定返回最优解。

下面再证明，如果 $h(n)$ 满足单调一致条件且问题有解时，那么 A* 算法的图搜索版本是最优的。

我们先表明，单调一致的启发函数具有如下性质：如果 $h(n)$ 具有单调一致性，那么在任何路径上，它的 $f(n)$ 是非递减的。该性质的证明直接来自于单调一致性的定义。设 n' 是 n 的后继节点，$g(n') = g(n) + cost(n, n')$ 成立，则 $f(n') = g(n') + h(n') = g(n) + cost(n, n') + h(n') \geq g(n) + h(n) = f(n)$。

根据该性质，可以证明，当 A* 算法选择节点 n 进行扩展时，表明已经找到了到达节点 n 的最优路径，即此时节点 n 的代价已经是最小代价，且节点 n 以后不会再被扩展[9]。如若不然，假设在后面的搜索过程中，找到了另一条到达节点 n 的路径，它的代价比现有路径的代价低。设新路径上节点 n 的父节点为 n'(父节点 n' 扩展了节点 n)，根据单调一致的性质，路径上节点的代价是非递减的，这意味着 n' 的代价比 n 的代价低，应该先被扩展，而这与 n 先于 n' 被扩展的事实相矛盾。因此，假设不成立。

由此可见，使用单调一致启发函数的 A* 算法以 $f(n)$ 非递减的顺序扩展节点，且不会重复扩展同一节点多次。这意味着，第 1 个被选择扩展的目标节点，它的代价一定是最小的，后续扩展的目标节点的代价都不会低于它。因此，如果 $h(n)$ 满足单调一致条件且问题有解时，那么 A* 算法的图搜索版本一定是最优的。

$f(n)$ 代价沿着任何路径都是非递减的事实，也意味着可以在状态空间上绘制

[8]如果一个搜索算法具有最优性，同时也扩展最少的节点，则称该算法也满足搜索最优性。

[9]需要注意的是，如果 A* 算法使用了一个可接受但非单调一致的启发函数，那么同一节点可能被扩展多次：只要找到了到达该节点的一个更好路径。

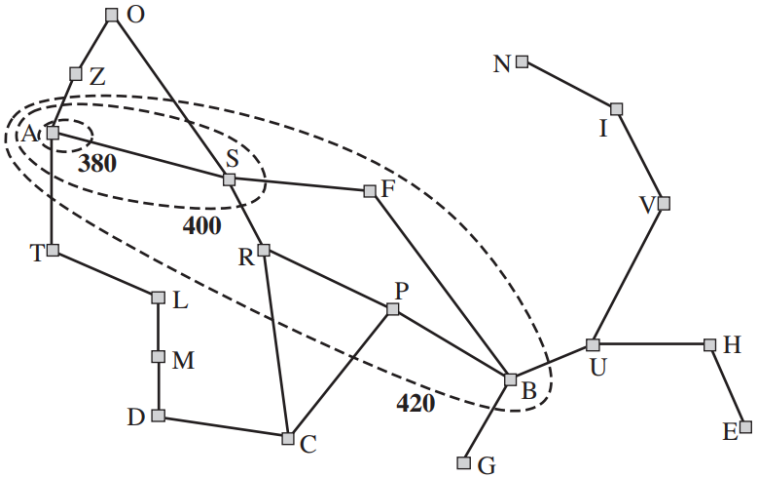

图 4-3: 罗马尼亚地图的等值线 (Arad 为初始节点)

等值线 (或等高线)，如图4-3所示。从该图可以看出，每条等值线近似为椭圆。如果使用更精确的启发函数，等值线就会向目标节点方向进一步拉伸，具有明显的目标导向性。而如果让 $h(n) = 0$(例如，一致代价搜索和宽度优先搜索，它们是 A* 搜索的特例)，那么搜索就会失去目标导向，此时的等值线呈现出圆形。

4.4 A* 算法存在的问题

A* 算法具有完备性与最优性，在满足单调一致性条件时，还具有效率 (搜索) 最优性。

然而，这并不意味着，A* 算法就一定能够满足我们的问题需求。A* 算法的问题在于，对于相当多的问题，在搜索空间中处于目标等值线以内的节点数量，仍然与解路径的长度呈指数级增长关系。

对于那些步骤代价为常量的问题，时间复杂度是深度 d 和相对误差 ϵ 或绝对误差 Δ 的函数。相关文献，请参考文献 [1] 的“第 3 章-参考文献与历史注释”。定义启发函数的绝对误差与相对误差为：$\Delta \equiv h^* - h$、$\epsilon \equiv (h^* - h)/h^*$。

A* 算法的复杂度非常依赖于对状态空间所作的假设。最简单的状态空间只有一个目标状态。在这种情况下，A* 的时间复杂度是最大绝对误差的指数级关系：$O(b^{\Delta}) = O(b^{\epsilon d}) = O((b^{\epsilon})^d)$。这表明，有效的分支因子为 b^{ϵ}。如果状态空间中包含了多个目标状态，情况就会变得更加糟糕。

因此，在这些问题当中，坚持使用 A* 算法找到问题的最优解就变得不那么

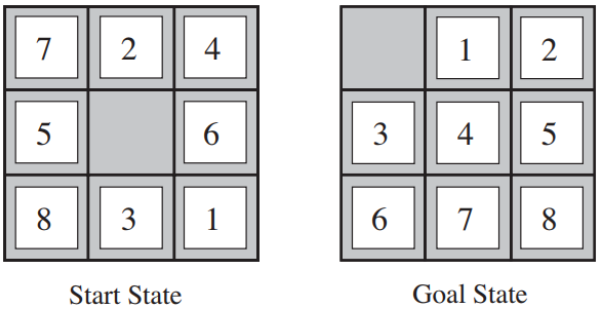

图 4-4: 一个典型的 8 数码实例

实用了。此时，可以使用 A* 的各种变形算法，快速地找到局部最优解，或者设计更精确却不严格满足可接受条件的启发函数。

然而，时间复杂度还不是 A* 算法的主要缺点。因为 A* 算法在内存中保存了所有已生成的节点 (这与图版本的搜索算法一样，需要记录访问历史)，A* 算法常常在问题求解之前就已经耗尽了内存。因此，对于很多大规模的搜索问题，A* 算法并不实用。

为此，一些算法进行了改进——通过多花费一些执行时间来克服内存问题，但同时又保证算法的最优性与完备性。例如，将迭代加深与 A* 算法进行结合的迭代加深 A* 算法 (IDA*)、递归最佳优先搜索 (RBFS)、内存受限 A*(MA*) 与简化的内存受限 A*(SMA*) 等算法，就属于 A* 的变形算法。还有一类被称为元学习的搜索算法，可以从经验中学习怎样避免探索没有希望的子树。

4.5 启发函数

启发函数对 A* 算法的性能影响非常大。下面以 8 数码问题为例，探讨启发函数的性质。

在 8 数码问题中，有一个初始状态和目标状态，每次移动一个方格到空格位置，直至达到目标状态为止。如图4-4所示，这是一个典型的 8 数码实例，解路径的长度为 26 步。

一个随机产生的 8 数码问题，解路径的平均长度为 22 步[10]，平均分支因子为

[10]注：8 数码问题有解的条件是，初始状态与目标状态的逆序数必须同为奇数或偶数。例如，在图4-4中，初始状态可表示为序列 $\{7,2,4,5,0,6,8,3,1\}$，目标状态可表示为序列 $\{0,1,2,3,4,5,6,7,8\}$。在计算逆序数时，不考虑空格 (用 0 表示)，它们的逆序数分别为 $0+1+1+1+1+0+5+7=16$ 和 0。因此，该 8 数码问题有解。

3。这意味着，它的状态个数为 $3^{22} \approx 3.1 \times 10^{10}$ 个。但是，在初始状态确定的情况下，并不是所有的状态都是可达的，可达的状态个数为 $9!/2 = 181440$ 个，此数目被削减了大约 170000 倍，状态空间似乎并不大。然而，对于 15 数码问题，该数目大概是 10^{13}。因此，需要找到好的启发函数，以便 A* 算法能够有效地工作。

常用的启发函数如下：

- h_1，不在位的方格个数

  本例中，8 个方格都不在相应的目标位置，$h_1 = 8$。

- h_2，所有方格到相应目标位置的距离和

  “距离” 指的是水平与垂直的距离和，也被称为曼哈顿距离或街区距离。

  本例中，$h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$。

这 2 个启发函数都满足可接受条件，它们都不会超过实际移动距离。

4.5.1 精确度对性能的影响

A* 算法要求启发函数具备可接受条件，即 $h \leq h^*$ 成立。

现假设有 2 个可接受的启发函数 h_1 和 h_2，如何衡量它们的性能，或者如何比较它们的优劣呢？

直观上来说，如果一个启发函数 h_1 比另一个启发函数 h_2 扩展更少的节点而找到最优解，那么我们称算法 1 要优于算法 2。

在 8 数码问题中，为了测试启发函数 h_1 和 h_2，可以随机地生成 1200 个 8 数码问题，解路径长度从 2 到 24 不等 (每个偶数值有 100 个例子)，再分别使用迭代加深搜索、A* 算法对它们求解，结果如图4-5所示。

左右图分别表示不同深度下 3 个算法的平均扩展节点数和有效分支因子。易知，h_2 要好于 h_1，并且要远远好于迭代加深算法。有效 (平均) 分支因子 b^* 的计算公式如下：

$$n = b^* + (b^*)^2 + ... + (b^*)^d \tag{5}$$

其中，n 表示扩展的节点总数，d 表示解的深度。

	Search Cost (nodes generated)			Effective Branching Factor		
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	3644035	227	73	2.78	1.42	1.24
14	–	539	113	–	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

图 4-5: 不同启发函数的比较

有效分支因子也是衡量算法优劣的一个标准。它可能随着问题实例发生变化，但是在足够难的问题中，它是相当稳定的[11]。

一般而言，使用启发值更大的可接受启发函数总是最佳选择，前提是计算启发值的时间不能太多。对于上述启发函数 h_1 和 h_2 来说，h_2 总要优于 h_1，即 h_2 总比 h_1 扩展更少的节点。原因在于，每个 $f(n) < f^*$ 的节点都必将被扩展，而对于任一节点，$h_1(n) \leq h_2(n)$，那么 $f_1(n) \leq f_2(n)$ 成立，这意味着在 A_2^* 中被扩展的节点在 A_1^* 中也被扩展，而 A_1^* 中还扩展其它的节点。

对实际算法进行随机统计，有益于探讨启发函数的总体性能与实用性。设计良好的启发函数会使 b^* 的值接近于 1，以合理的计算代价对大规模问题进行求解。

4.5.2 利用松弛问题设计启发函数

所谓松弛问题，指的是去掉原问题中的某些 (动作或转移) 约束条件，这样就更容易地找到原问题的最优解。这样做的意义在于，一个松弛问题的最优解代价可以作为原问题的可接受启发函数，更进一步地，由于得到的启发值是松弛问题的确切代价，那么它一定遵守三角不等式，即满足单调一致条件。

如果对问题定义采用形式语言描述，那么就有可能自动地构造出松弛问题，并

[11] 有效分支因子的存在，也佐证了4.4中提到的 A* 算法问题——A* 算法扩展的节点数随深度呈指数级增长。

进而自动地生成启发函数。一个名为 Absolver 的求解程序为 8 数码问题找到了比以前更好的启发函数，并且也为著名的魔方游戏找到了第 1 个有用的启发函数[12]。

松弛问题的状态空间是原问题状态空间的超图，因为松弛问题相当于减少了原状态空间中节点到节点之间进行转换的动作限制，即在原状态空间中，为某些节点增加了转移边。原问题中的任一最优解一定是松弛问题中的解，但松弛问题可能存在更好的解，理由是增加的边可能导致捷径。因此，松弛问题中的最优解代价一定是原问题的可接受的启发值，并且，由于得到的启发值是松弛问题的实际代价，那么它也一定满足三角不等式，即也满足单调一致条件。

以 8 数码问题为例，原问题的动作约束可以描述如下：如果方格 A 与方格 B 直接 (水平或垂直) 相邻，并且 B 是空的，那么棋子可以从方格 A 移动到方格 B。如果去掉其中的 1 个或 2 个条件，可以生成如下的松弛问题：

1. 如果方格 A 与方格 B 直接相邻，棋子可以从方格 A 移动到方格 B；

2. 棋子可以从方格 A 移动到方格 B；

第 1 个松弛问题，可以计算出启发函数 h_2，即曼哈顿距离的启发函数——依次将每个棋子移入相应的目标位置，h_2 就是相应的步数；第 2 个松弛问题，可以计算出启发函数 h_1——依次将不在位的棋子一步移入相应的目标位置，h_1 就是相应的步数。

如果已经找到一个启发函数集合 $h = \{h_1, h_2, ..., h_m\}$，并且它们都没有绝对的优势，此时可以采用如下公式确定“最佳的”启发函数：

$$h(n) = max\{h_1(n), h_2(n), ..., h_m(n)\} \tag{6}$$

该启发函数比集合 h 中的任一启发函数都要好。

4.5.3 利用状态模式设计启发函数

状态空间是由节点及节点之间的转移 (有时被称为动作) 组成的。在松弛问题中，对原问题的转移条件进行了弱化或放宽了限制，从而可以让算法更方便地在状态空间的超图中进行搜索，并以此来设计启发函数。

[12]参见论文，Prieditis, A. E. (1993). Machine discovery of effective admissible heuristics. Machine Learning, 12(1-3), 117-141.

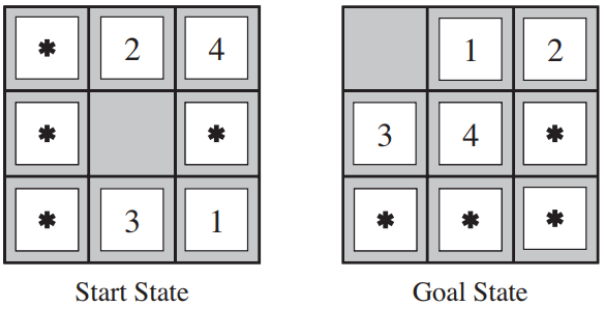

图 4-6: 起始节点与目标节点的状态模式

假设对状态空间中的状态 (节点) 进行模式化处理，即允许状态中的某些“位”为任意值 (用 * 表示)，而其它一些“位”为具体值的情形。显然，与原问题相比，新的子问题更易求解，因为只需要考虑那些有具体值的“位”，将它们移入相应的目标位置中，而忽略那些带有 * 的“位”。图4-6展示的是 8 数码问题中的起始节点与目标节点的状态模式。

在这个实例中，子问题涉及到将棋子 1-4 移动到相应的目标位置上。显然，与松弛问题一样，子问题的最优解代价可以作为原问题的可接受启发函数。在某些情况下，它比曼哈顿启发函数 h_2 要更准确些。

如果针对每个子问题的实例，都计算出相应的解代价，那么就形成了所谓的模式数据库。上述的 1-2-3-4 棋子结构就可以形成一个模式数据库，同样也可以构造其它形式的 1-2-3-4 棋子结构，并构造相应的模式数据库。当然，也可以构造 5-6-7-8 或 2-4-6-8 等模式数据库。每个模式数据库都能产生一个启发函数，这些启发函数可以按照公式6组合使用，所得到的启发函数要比曼哈顿启发函数 h_2 更精确——在求解随机的 15 数码问题时，所生成的节点数要少 1000 倍。

还有一种思路，可以得到更为精确的启发函数。例如，可以将特定的 1-2-3-4 模式数据库的启发函数与特定的 5-6-7-8 模式数据库的启发函数进行相加——当然，相应的启发函数不是求解子问题的总代价值，而是只涉及 1-2-3-4 或 5-6-7-8 棋子的移动代价。因此，2 个子问题的代价之和，仍然满足可接受条件。这就是不相交的模式数据库的设计思路。与使用曼哈顿启发函数相比，所生成的节点数减少了 10,000 倍，而对于 24 数码问题，减少的节点数可以达到百万倍。

4.5.4 从经验中学习启发函数

这是一个有趣的研究问题，迄今为止，从经验中学习仍然是各领域的研究热点。可以相信，在未来很长的一段时间内，它将始终是一个新算法层出不穷、令人激动的研究领域。

在所有的问题中，“经验” 意味着许多的问题实例，智能体 (Agent) 既可以在线的方式，也可以离线的方式进行学习或训练。

对于 8 数码问题，每个最优解都可以成为 “经验” 实例，可供智能体学习或训练——这些实例都包括解路径上的状态及到达解的代价。

在理想情况下，学习算法可以从大量的实例中自动地构造出启发函数，并能预测出搜索过程中出现的其它状态的解代价。可用的学习算法包括监督学习、半监督学习及无监督学习等方法。这些学习方法将在后面陆续介绍。

5 练习

1. 编写程序，实现 8 数码问题的 A* 搜索算法；

6 参考文献

1. Stuart J. Russell, Peter Norvig, 殷建平等译。《人工智能：一种现代的方法》，第 3 版，清华大学出版社。

2. 马少平，朱小燕。《人工智能》，2004 年 8 月第 1 版，清华大学出版社。