[搜索算法的通用优化方法]

[DFS]

[搜索剪枝]

在很多情况下，我们已经找到了一组比较好的解。但是计算机仍然会义无返顾地去搜索比它更“劣”的其他解，搜索到后也只能回溯。为了避免出现这种情况，我们需要灵活地去定制回溯搜索的边界。

***例题** 计算机网络连接[1]

要将 n(n<=30)台计算机连成网络，连接方法：去除首尾两台计算机与一台计算机相连以外，其他计算机只与两台计算机相连。连接的长度则为计算机连接的电缆的长度。

求：一种连接方式，使需要电缆的长度最短。

分析 这个题目用回溯搜索来解决。但是，由于回溯搜索的搜索量比较大，达到了 n!，是不可能搜索完 n=30 的情况的，所以，我们考虑对它进行优化：

假如目前搜索到了一组解，电缆总长度为 kx，那么，如果说以后搜索到的连接方法（不一定是最终连接方法）的连接长度>=kx，那么这个方案的总长度一定不小于 kx，那么，就不必要搜索下去了，直接换下一个结点继续搜索。

路径 A1-A2-…An 与路径 An-An-1-…A1 这两条路径是一个“正反”的关系，本质上是相同的，于是我们可以规定起点始的下标总是小于终点的下标

假如路径的 A-B-C-D 的长度<A-C-B-D 的长度，那么包含 A-C-B-D 路径的路径的长度一定不是最短。

有了上述的优化，题目就可以得到很快的解决了。

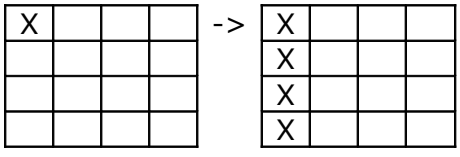
在深度优先搜索的过程当中，往往有很多走不通的“死路”。假如我们把这些“死路”排除在外，不是可以节省很多的时间吗？

打一个比方，前面有一个路径，别人已经提示：“这是死路，肯定不通”，而你的程序仍然很“执着”地要继续朝这个方向走，走到头来才发现，别人的提示是正确的。这样，浪费了很多的时间。针对这种情况，我们可以把“死路”给标记一下不走，就可以得到更高的搜索效率。

***例题** 皇后问题[2]

分析 取 n=4 为例

采用一般的回溯，就是每一行的每个格子放与不放都搜索一下：



然后回溯一次，换下一个点继续搜索。

这个算法的效率，是 $p(n,n)=n!$

实际上，在放置了(1,1)这个皇后，再把皇后放置在(2,1)就是毫无意义的：前面一个皇后一定能攻击到它。

为了避免这种情况，我们这样做：

---

[1] 题目来源：GDOI

[2] 题目来源：经典问题

X	.	.	.
.	.		
.		.	
.			.

->

X	.	.	.
.	.	X	
.		.	
.			.

走了一个棋子以后，把它的“势力范围”给圈出来，并且告诉以后的皇后：这里不能放置。举简单的例子：放置皇后(1,1)，由于打“.”的格子在放了(1,1)这颗子之后，被标注为了“不能走”，所以这些点我们就不去理会了。这样就节省了很多时间，大大提高了搜索的效率。

而对于很多回溯的题目，我们都可以采用分枝定界法，把搜索树中不必要的枝剪去，大大提高了搜索的效率。

[记忆化]

对于一些有最优子结构的问题，我们往往采用动态规划算法来实现。采用动态规划算法，需要弄清状态以及状态是如何转移的，接着列出状态转移方程。首先举一个非常简单的例子

***例题** 数字三角形[3]

分析 无论对与新手还是老手，这都是再熟悉不过的题了，很容易地，我们写出状态转移方程：

f(i, j)=a[i, j] + min{f(i + 1, j) + f(i, j + 1)}

对于动态规划算法解决这个问题，我们根据状态转移方程和状态转移方向，比较容易地写出动态规划的循环表示方法。但是，当状态和转移非常复杂的时候，也许写出循环式的动态规划就不是那么简单了。

我们尝试从正面的思路去分析问题，如上例，不难得出一个非常简单的递归过程：

```
if (i==0) return (a[i, j]);
f1 = f(i + 1, j); f2 = f(i, j + 1);
if f1>f2 return a[i, j] + f1; else return a[i, j] + f2;
```

显而易见，这个算法就是最简单的搜索算法。时间复杂度为 2^n，明显是会超时的。分析一下搜索的过程，实际上，很多调用都是不必要的，也就是把产生过的最优状态，又产生了一次。为了避免浪费，很显然，我们存放一个 opt 数组：

Opt[i, j] - 每产生一个 f(i, j)，将 f(i, j)的值放入 opt 中，以后再次调用到 f(i, j)的时候，直接从 opt[i, j]来取就可以了。

于是动态规划的状态转移方程被直观地表示出来了，这样节省了思维的难度，减少了编程的技巧，而运行时间只是相差常数的复杂度，而且在相当多的情况下，递归算法能更好地避免浪费[4]，在比赛中是非常实用的。

总结 记忆化搜索是对动态规划状态转移方程的直观表示。本质上来说，它仍然是用搜索算法的核心，只不过使用“记录求过的状态”的办法，来避免重复搜索，这样，记忆化搜索的每一步，也可以对应到动态规划算法中去。记忆化搜索有优化方便、调试容易、思维直观的优点，但是效率上比循环的动态规划差一个常数，但是时间和空间复杂度是同一数量级的（尽管空间上也差一个常数，那就是堆栈空间）。当 n 比较小的时候，我们可以忽略这个常数，从而记忆化搜索可以和动态规

[3] 题目来源：经典问题

[4] 参见江苏省队选拔 Day1 巧切蛋糕

划达到完全相同的效果。

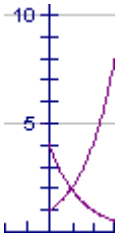
[BFS]

[双向搜索]

在 bfs 算法所能解决的问题当中，有相当一部分，是给你初状态和末状态，让你求一条从初状态到末状态的最短路，实际上，bfs 的结点产生系统也最适合解决这一类的问题。对于无休止以指数级膨胀的队列长度，选手们往往束手无策。

其实这一类问题，有一个比较难实现，但是却能很好提高算法效率的办法，那就是，我们从初始结点开始扩展，每扩展一层（暂时称它为 f1），再从目标结点按照产生系统相反的办法来扩展结点（称它为 f2），直到 f1 和 f2 产生出了相同的结点，把中间路线输出就可以了。

这一类问题，很明显，需要状态产生是可逆并且容易实现的。太复杂的逆向状态产生也许会带来更大的副面效果，为了更好地对比这两种算法的优劣，我绘制了一张函数图像：



很容易看出，双向搜索比单向搜索在数量级不断增大的时候，双向搜索所体现出的优势就非常明显了。扩展次数越多，这个差距越明显。假设一个结点产生系统呈二叉树状增长，那么扩展 n 次的代价，单向是 2^n，而双向则是 $2*2^{n/2}=2^{n/2+1}$，两者相差甚远。

***例题** 魔板问题[5]

分析 这个题描述的非常明确：给定初始状态和目标状态和状态产生系统，要求从初状态到末状态的最短路径，符合上述双向 bfs 的特点。很明显，我们可以用双向广度搜索来解决。

对于每一个状态，有两种存储方式：

（1） 直接用 8 个 Byte 类型存储

（2） 用一个 Longint 类型压缩存储

虽然（2）略浪费了一些时间，但是却节省了一半的空间，对于状态比较多的情况，这是很合算的，因为占用空间变大，势必造成时间的增加，在状态较多的情况下，两者是比较平衡的。

双向 bfs 比较难于掌握，比较重要的原因，是因为它易错。本来检查左右两边是否重合的过程就比较复杂，而且一旦两边扩展的结点没有查找到（两端的搜索“失之交臂”），往往该程序会陷入死循环。所以实际处理的时候，一定要相当细心。

本题还可以和下面即将介绍的散列表联用，会收到非常良好的效果。

小结 双向 bfs 应用范围还是相当广泛的，所有已知初状态和末状态，让你求一

[5] 题目来源：IOI1998。尽管这个题使用 Hash Table 来优化能收到相当好的效果，我们仍然把它作为双向宽度搜索的典型例题来讲。

条从初状态到末状态的最短路的一类问题，几乎都可以用本算法来解决。本算法思想非常简单，但是实现却比较困难，需要比较多的编程经验和比较丰富的编程技巧，但是换来的效果也是非常明显的，一般竞赛时不推荐使用，但是真正做题是，双向 bfs 无疑是一种高效的优化途径。

[散列表]

很明显，宽度优先搜索产生的状态是非常多的，因为扩展结点是不必要回溯，所以宽度搜索是一种以空间换时间的搜索策略，对空间占用量比较大，所以空间上的优化成为宽度搜索的主要优化之一，而避免重复状态又是减少空间浪费的最主要途径。拿迷宫问题为例，如果根本不避免重复状态直接搜索，其空间复杂度约为 2^{m+n}，而如果避免了重复状态，时间复杂度为 m*n，这两个数据的大小有着天壤之别。

避免重复状态，也就是在生成一个状态以后，把它记录在一种形式的表里，接着在以后产生状态以后，判断这个表里是否含有这个状态，不难看出，这实际上就是一个插入和查找的过程。

为了使插入和查找更加地迅速，我们可以采取散列表这种数据结构，因为只有它，才能在非常短的时间内实现插入和查找，插入的时间复杂度和链接表相当，而查找的速度远远快于二分法。当建立一个完整状态的队列（状态表）不是很困难的时候，散列表往往也能够随之起到很重要的作用，大大提高时间复杂度。

***例题** 解密牛语[6]

分析 基于密码编译规则，我们很容易地可以想出一个非常简单的 dfs 方法，当然，那是明显要超时的，于是我们不得不采用宽度优先搜索算法。这个题有几个比较常规的剪枝，在这里就不一一列举了。我们看一看搜索的主体部分。

Bfs 不能避免的是重复状态，而用循环判断重复是得不偿失的，在状态多的情况下，循环法甚至比 dfs 效率更低，而且低得多。本题难就难在字符串的散列压缩上。首先，我们需要明确的，是散列表对于字符串，是不可能做到一对一的映射的，那么我们不可避免地，要把字符串以一定的函数转换为编号，并且进行取 mod。事实证明，我们在 hash 表里再存储一次状态，空间上是吃不消的（尽管那样是保证正确的），于是我们根据字符序数和位置取得 hash 地址，并且直接对 hash[p]赋值，选取适当的 hash 函数，还是可以有效地解决问题的。

根据 hash 表的原则，mod 的数值最好取 1.1a~1.6a 之间的一个质数，在具体实现中，我们取 502973[7]，收到比较良好的效果。

总结 散列表同记忆化搜索一样，是非常重要的搜索优化方式，同记忆化搜索一样，它能够把搜索算法的效率从大指数级提高到小指数级、多项式级甚至常数级。同时，作为一种高效查找方法，散列表以及散列表的思想渗透在了计算机技术当中，成为很多算法的核心。

在这里，使用散列表还有很多技巧：例如也许从现有状态出发，推出不重复的散列地址来很困难，选手就可以尝试着用一个比较简单的方法推出一个可能重复的、或者数量非常大的特征状态，然后用 mod 大数或者拉链解决冲突的方法来处理，同样能够收到良好的效果。

---

[6] 题目来源：USACO training system

[7] 具体问题以及竞赛中，我们可以临时写一个质数产生程序。

散列表的优化技术并不困难，但是散列函数的构造、处理冲突的技巧和散列表运用的角度，是很值得思考的问题。具体方法可以参见《数据结构》和相关例题。

[估价函数]

启发式搜索的主要目标是使用一个函数去判断所有状态的“好坏”，以提高搜索找到解的效率。

通常，估价函数表示成一个函数或是一个状态，这个函数叫做“估价函数”。对于相同的题目，有时有不同的估价函数。直观地看，越优秀的估价函数，搜索的速度就越快。当然，估价函数不一定是十全十美的（否则就是贪心法了），总归会对某些状态予以不太准确的评价，于是，评价值（函数返回值）和实际好坏的差异越小，估价函数就越优秀。

注意！一个人脑看起来似乎非常非常弱智甚至笑它太傻的估价函数可能收到非常大的效果，也许搜索算法的运行时间（或空间）会缩小 100000 倍甚至更多。

对于启发式搜索的应用，有以下几点：

（1）启发式剪枝

最简单也是最常用的启发式搜索是利用估价函数来剪枝。假设我们的问题是要求找最小总花费。对于一个可接受的估价函数，当前花费是 A，启发函数返回了 B，当前子问的最优解是 A+B。如果找到的一个解一个花费是 C，C<A+B，这个状态就不必要搜索了。

这样编写和调试也比较简单（假设一个状态需要长时间而被剪掉……），且可以极大地提高程序效率。它对 DFS 尤其有效。

（2）最佳优先搜索

这种方法好比就是贪心算法。

每次不扩展所有子结点，而是按“好坏程度”来扩展。与贪心不同的是，贪心只尝试“最优”路径，但是 BFS 首先扩展“希望大”的，再扩展“希望小”的，如果结合上述描述，搜索会得到很好的结果。

（3）A*法

A*法是类似贪心的 BFS。

BFS 一般扩展最小耗费的点。A*算法在另一方面，扩展最有希望的点（估价函数返回值最优）。

状态被保存在一个优先队列中，按照 Cost*价值排列。每一次，程序处理最低优先的点，且把它的孩子按照适当顺序处理。

对于一个可容许的估价函数，第一个找到的状态保证是最优的。

***例题** 八数码问题[8]

明显的，我们可以用 hash 表的办法，映射每一个八数码状态，总搜索量为 9!=362880。这个数值很小，是完全可以忍受的。但是，如果采用了估价函数进行优化，采用最佳优先搜索方法，会收到更好的效果。

八数码问题有一个非常经典的估价函数：

对于数码每一个方格和目标的差距相加，例如：

```
3  1  2        1  2  3
4  6  5  ->    4  5  6
0  8  7        7  8  0
```

估价函数返回的值为：

[8] 尽管用 hash 表可以完美解决这个问题，用启发函数法能够收到更好的效果

2 + 1 + 1 + 0 + 1 + 1 + 2 + 0 + 2 = 10

返回值越小，说明该状态离目标状态最近。虽然这个启发函数不完美，误判的情况很多，但是它能够非常大地提高搜索效率，在 n 比较大的时候，有助于在非常短的时间内找到可行解，并且可以用收缩节点的办法，找出更优的可行解。

小结 启发式搜索并不是完美的搜索。无论怎样的启发函数，都会存在一定的缺陷但是缺陷并不影响搜索效率的提高。竞赛和实际问题中，越来越多的问题，不要求最优解，只要求可行解，但是往往找这些可行解相当的困难，此时往往就需要启发函数来帮忙。启发函数会在极其短的时间内找到一个较优解，接着，我们可以根据这个解进行限界甚至收缩，得到满意的结果。

[搜索算法的特例优化方法]

[扩展结点上的优化]

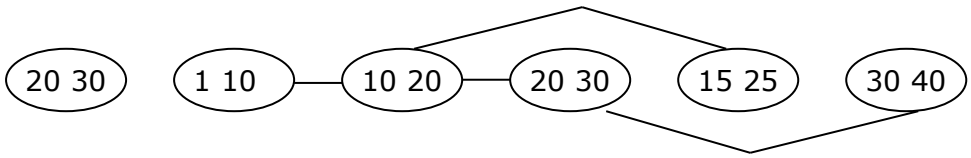
众所周知，对于某些动态规划题，对状态转移的设计要求是非常高的。往往用 $o(\log_2 n)$的时间复杂度实现状态的转移，与用 o(n)的时间复杂度实现状态转移效率有天壤之别。

这种优化技巧很明显可以用在记忆化的 dfs 上，而且往往收到很好的效果，因为记忆化 dfs 与循环实现的动态规划只有常数上的差别，而对于某些状态不完全的动态规划，记忆化搜索的效率甚至好过循环。

其实在搜索算法中，状态转移上的优化（也就是扩展结点上的优化），在优化中仍然能起很大的作用，只是大家总是忙于探索如何减少搜索算法的浪费，忽略了这一点。

***例题** 最大机[9]

分析 对于两个排序机 a[i]和 a[j]，如果 a[i].endvex 在 a[j].startvex 和 a[j].endvex 之间，我们就把它们连一条边，样例如下图所示：



很明显，我们所要求的排序机数量，就是 a[p].startvex=1, a[q].startvex=n 的最短路径。因为这个最短路径非常特殊，所有的边权都是 1，所以我们采取用宽度优先搜索的办法来实现。

粗略地看一下，bfs 的时间复杂度是 o(m)，是完全可以承受的，但是不能忽略的是，如果我们用 Dijkstra 式的扩展方式，那么，扩展结点的复杂度是 o(m),对于每一个结点都扩展一次，就是 m^2。而 m 最大到 500000，m^2 的算法是必然行不通的。

但是不是这样就可以否定 bfs 算法呢？答案是否定的。

紧扣着产生结点的规律，又因为 n<=50000，于是我们开一个 hash 表，hash[i]表示可以以第 i 个排序位置为起点的排序机，一开始不是把 m 个排序机的信息单纯放在数组中，而是放在这个 hash 表中，以待查，每次扩展结点的时候进行扫描的时候，不必要把 m 个都扫描一次，只要对映射在 a[p].startvex 到 a[p].endvex 的结点扫描一次就可以了，而且，因为边权为 1，所以扫描过的结点就可以不扫描了，于是可以直接把散列表中

[9] 题目来源：ACM CEERC 2003 Problem I
问题用搜索算法解决以后，侯启明同学提出了一个扫描式的贪心方法，时间复杂度为 o(m)。但是由于思维复杂度很高，这里不予介绍。

的结点删除，以增加下一次扫描的速度。

散列表解决冲突可以使用一个单链表（方便插入与删除）。而因为只有 i>j 的时候，a[j]才可以和 a[i]相连，所以 hash 单元里也可以是一棵以结点号为关键字的二叉排序树，以提高检索速度。对于扩展结点时的扫描，也可以再开一个 hash 表来映射结点，这样两个不同的 hash 表进行映射，大大增加了效率，算法时间复杂度接近 o(m*f(x))，f(x)为 hash 查找的平均时间。

我们再看看动态规划算法。一般的动态规划也是 m^2，如果用二叉排序树或二叉平衡树（尽管那很烦琐）来实现，可以优化到 $m\log_2 m$，但是 $\log_2 m$ 比 f(x)要大的多：500000 个元素映射到 50000 个 hash 单元中，平均每个单元只有 10 个元素，加上二叉排序树的复杂度是 $\log_2(k)$，这个数值大约是 3.3，而 $\log_2 m$ 的数值大约是 18.9。两者效率相差约 5.7 倍。

小结 搜索的时间的确主要耗费在检查状态树上，但是当结点非常多的时候，我们仍然有必要好好考虑应该如何优化结点的扩展。毕竟每转移一次状态耗费 n 的时间，在 n 很大的时候是得不偿失的。

状态扩展的本质是查找。当状态转移数非常数时，可以考虑排序+二分查找，也可以构建索引、散列或者是二叉排序树，也可以将这几个数据结构进行有机结合，例如索引元素是散列，散列解决冲突用排序树。每每遇到这个种类的问题，我们需要充分利用数据结构上的技巧，把状态转移时间降到最低。

[图论模型的转化]

在众多搜索问题中，有相当一部分需要通过对问题进行构图，并且转化成图论问题，如最短路径、最小生成树、最大流等。而伴随着图的不同转化方式，所适用的办法不一定相同，就拿排序机为例，如果按照上例的构图方法，就是最短路径问题，如果换一种表示方法，我们还可以把它转化成最小生成树问题。

为什么要讨论这个问题呢？因为不同图论模型的转换，可能导致使用算法的不同，也可能导致算法复杂度不同。上例中，最小生成树法和最短路径法，本质上基本是相同的，所以经过优化，时间复杂度也基本相同。下面，我们讨论一个例题，就可以体会到图论模型转换的重要性。

***例题** 多米诺骨牌[10]

分析 方法 1：顺着题目的思路，非常明显，对于 n 个骨牌，如果两个骨牌可以连接，则把它们之间连一条边，很明显，这个问题转化成了建立以后图的哈密顿回路[11]。众所周知，哈密顿回路是经典的 NP 问题[12]。实现这个算法，有比较多的办法。稳定而且保守的办法是对这张图进行 dfs 以及回溯。每次探索结点。当然，这个算法的复杂度数量级为 o(n!)，这个数值是非常大的，尽管实际上远达不到这个数值，但是当 n>=20 的时候，在时限内出解也是非常困难的。

方法 2：利用这个题目的特殊性，我们可以把 0-6 这 7 种牌面作为结点，而多米诺骨牌作为这些结点之间连结的边，从而重新构图，最后所求的问题，转换为了欧拉路问题[13]。对于欧拉路问题，有一个经典算法：

使用一般图的深度优先搜索方法，注意访问过的边不要访问第二次，接着在回溯的时

[10] 题目来源：Sgu Problemset

[11] 哈密顿回路：对于一个无向图，存在一条路径，使得每个点都被访问且仅访问一次。

[12] NP 问题：时间复杂度不能用多项式表示的问题。

[13] 欧拉路问题：对于一个无向图，存在一条路径，访问每一条边都被访问且仅访问一次。

候，记录回溯的边。最后把这些边整理一下，就得到了这个图的欧拉回路。

而如果一个图不连通或不满足存在 0 或 2 个奇度点，则这个图不存在欧拉回路。这个程序的时间复杂度是 $o(n^2)$。对于 n=100 来说，是非常小的一个数字，相比 n!的复杂度，已经有了本质的飞跃。经过进一步的优化，可以将程序复杂度优化为 o(e)，也就是 o(n)。

小结 事实上，图论模型的转换，就是问题类型上的转换，也就是思考问题角度的转换。进行转化的目的，也就是简化问题或者使得问题能够更好地解决。

从而，我们看到，解决一个搜索问题，不但需要微观上各种数据结构、算法性、程序设计的技巧，也需要宏观上去解决问题分析的问题。因为数据结构、算法上的优化，是不能带来本质上优化的，但是图论模型的转换做得到，一个巧妙的图论模型往往会带来意想不到的效果。

[化整为零式的拆点方法]

看一个很简单的例子：迷宫问题。有 n*m 的方格，可以向相邻方向移动，求从(1,1)点到(n,m)点的最短路径。

这是一个最短路径问题，熟悉 bfs 的同学一定会用 o(n*m)的算法来实现[14]，当然，这个题也可以用 Dijkstra 算法来实现。分析一下复杂度：有 n^2 个点，有 4*n^2 条边，Dijkstra 算法的标准复杂度是 o(16*n^4)，优化后可以为 $o(n^2\log_2(n))$[15]。

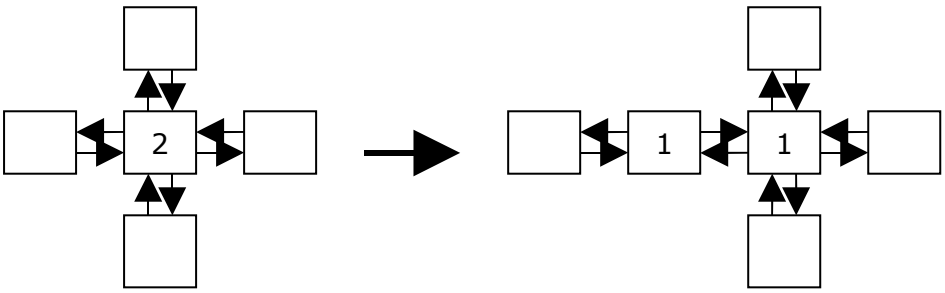
同样的，如果我们这个迷宫改变一下，有的通过代价是 2，有的通过代价是 1，那么，似乎第 1 种简单 bfs 的办法就行不通了。第二种 Dijkstra 仍然行的通，但是时间效率并不理想。

于是，从把复杂问题简单化的角度考虑，我们可以尝试着把复杂结点拆成简单结点来处理。

***例题 救援行动[16]**

分析 拿到题目，乍一看，似乎可以用最简单的 bfs 解决，其实不然。最基本简单的 bfs 算法在处理点的时候，因为深度度不同，所以先扩展到的不一定最优，从而违背了最优子结构，每个点不能只被扩展一次，从而造成了空间浪费。那么，我们尝试把复杂问题向简单问题靠拢。

对于一个“看守”（也就是移动到该单元格需要 2 单位时间），我们为什么不能把它看成两个一般结点组成的必经路径呢？想到这里，问题的思路就已经非常明晰了。进行“拆点”的过程如下图：



分析起来比较简单，实现起来，如果还是用最短路径的方法，扫描次数为 4*n*m，这已经比 Dijkstra 算法进了一大步（从 $o(p\log_2 p)$的算法到 o(p)的算法），为此，有一个细节上的处理技巧：在 bfs 扩展结点的时候，当扩展到一个为“X”的结点以后，占用一个

[14] 也就是用一个 hash 表记录访问过的结点，保证每个点最多只访问一次。
[15] 使用菲波那契堆实现优先队列。
[16] 题目来源：作者原创

单位空间，把同样的结点再扩展一遍，并且将原来的结点置为“无效（防止以后继续扩展）”。从而，这个算法扫描结点的次书为 2*n*m，比直接进行图论转换效率高一倍。

小结 应用同类方法可以解决的，还有 SGOI 的街道问题等经典问题。这一类问题，往往拿到以后难于处理，没有头续，想出的一般算法又不一定能应对问题的数量级，且问题分析后，发现问题牵涉的状态都是建立在较小整数的基础上。

解决这一类问题，“拆点”其实是一种高级的算法技巧，需要比较高的问题分析能。上例拆开的，仅仅是最简单意义上的拆结点，而且只对固定的结点拆成了两个，但是，它体现了拆点法的最本质思想。拆结点将非常规问题常规化。我们可以作一个比较，对于上例，n<=200，所以 Dijkstra 算法也是能通过的，当然，为了扫描节约时间，还必须把图建成邻接表的形式，且需要用菲波那契堆来实现算法，编程复杂度是可想而知的。而拆点的 bfs 方法，虽然思维难度稍微大一些，但是不但提高了效率，还简化了编程复杂度，甚至可以直接在 bfs 的基本模块上修改。

拆点算法扩大了算法的常数项，但是换来的，往往比想出一种复杂高效算法更快、更巧、更省力。

[不断变化图论模型的搜索]

某些搜索问题乍看之下不好解决，于是就把它们进行适当的转换，通常，我们会把一个实际问题抽象出一个图论模型来，接着在图论模型的基础上进行搜索。

现实中的问题，并不是所有的问题用一个单一的、固定不变的图论模型就能轻易解决的，某些时候，构造好的图论模型也在不断变化。

解决这一类问题，我们必须很好地把握问题的本质，善于发觉变化中的不变，发现变化中的规律。把一些看起来可以无限重复、循环的问题，通过数学手段的证明，转换成有限次数的搜索，最后制定出完美的搜索算法。

***例题** 火山[17]

分析 最基本的想法：根据时间，进行 dfs 搜索，搜索所有路径，最后找出最短的路径，并且把它输出[18]。

很显然，为了避免走到火山口上，这个题每一个方格可以经过一次，两次甚至更多。于是 dfs 搜索即使限定了搜索界限，效率仍然非常低下，但是 n,m<=15 的情况下，解决问题还是没有问题的。那么，有没有更迅速的算法呢？

答案是肯定的。首先，我们要明确的是，所有火山喷发的周期是 lcm(1,2,3,4,5,6)。这个数值是 60，也就是说，无论如何，解的步数也不会超过 60*2=120，这对于 dfs 是很有用的界限。

接着，我们可以构造一个[1..Maxn, 1..Maxn, 1..Maxt]的数组来表示迷宫，对于状态[x, y, t]={true, false}分别表示在 t 时间(x, y)点能否经过（当格是否有火山喷发），这样，我们把原先棘手难于处理的图论模型转换成了比较简单的图论模型。

继续分析，我们可以证明，如果在[x, y, Time0]到达过(x, y)点，那么有另外一条路径，在 Time0 时同样到达了(x, y)点，那么这就是没有意义的。换句话说，这个图论模型无论如何变化，仍然具有最短路径的最优子结构。

这个图论模型具有如下特点：

1) 边权为 1

2) 最优子结构

---

[17] 题目来源：Zoj Monthly Contest March 13, 2004 本题是这次 Monthly 通过数较少的一题

[18] 这个算法的时间复杂度非常高，是非多项式级的，但是加入较强剪枝和启发函数后仍然能得到良好效果

3) 深度有限

有了这 3 个特点，我们很容易想到最简单的 bfs，而且，根据(2)我们可以知道，当访问过(x, y, Time)这个结点以后，这个结点就不能再被访问了，于是 Array[x, y, Time]就可以标为 False。

这个算法的复杂度是 o(N*M*T)。对于这个题目而言，极限数据只需要运算 27000 次，是非常快速的算法。

小结 笔者第一次接触这种类型的题，是在 GDOI-Z4 邀请赛 AMI。那个题的规模更小，最短路径也可以通过。这个优化技巧，适用于很多地方，它把无限化为了有限，把运动化成了静止，不但给分析问题、解题带来了很大的方便，而且迅速地提高了程序效率。