Dancing Links 在搜索中的应用

momodi

2008 年 7 月 8 日

目录

1 Abstract

1.1 Dancing Links是什么

Dancing Links 是knuth 在近几年写的一篇论文，在我看来是一类搜索问题的通用优化, 因此我把它写下来，希望能在竞赛中得到一定的应用。

1.2 Dancing Links的主要原理是什么

Dancing Links 主要是用双向十字链表来存储稀疏矩阵，来达到在搜索中的优化。在搜索问题中，所需要存储的矩阵往往随着递归的加深会变得越来越稀疏，这种情况用Dancing Links 来存储矩阵，往往可以取得非常好的效果。

1.3 这篇论文与knuth论文的不同

本篇论文是对Dancing Links 在竞赛的应用的一个介绍，并列举了几个竞赛中的例题。原文可以在此下载到：

http://www.ocf.berkeley.edu/ jchu/publicportal/sudoku/0011047.pdf

相对于本文，我更建议去读者去看knuth的原文，本文可以当作一个参考。本文还介绍了Sukudo问题和一个Dancing Links的一个变种问题。

2 双向链表

2.1 双向链表的存储结构

双向链表的存储结构往往是有一个空的结点来表示头指针。然后每一个结点有一个pre值域和一个next值域。如果是十字链表的话，结点中指针的值域会变成四个。

2.2 双向链表的应用

众所周知，在单向链表中删除一个结点是非常麻烦，而且低效的。双向链表有着优美的结构，可以方便的删除任意一个结点。有如下操作：

$$L[R[x]] = L[x],\ R[L[x]] = R[x]; \tag{1}$$

相信有写过双向链表经验的人对这两句话一定不会陌生.

2.3 双向链表的恢复

在双向链表中删除一个结点只要用(1) 就可以了，然后来看下面这一段代码：

$$L[R[x]] \ = \ x, \ R[L[x]] \ = \ x; \tag{2}$$

这段代码会将已经删除掉的结点，重新添加回双向链表中。第一次看到这段代码的你是不是会感觉非常奇妙？你有可能会说在链表中进行删除操作但是不释放内存的话是不好的编程习惯，会造成内存溢出错误。但是在这里，我们要的就是这样的效果。

我们在链表中删除，只是进行一个标记，并不进行真正的清空内存操作。这样我们后来便可以用(2) 进行恢复操作了。

你也有可能会说这段代码会有什么用？下面就让我们来真正看看(2) 强大的威力。(2)才是Dancing Links 的核心.

3 Exact Cover Problem

3.1 Description

上节所说的双向链表的恢复操作会在本节中有重要的作用。下面先让我们来看一个问题(Exact Cover Problem)。

给定一个01矩阵, 现在要选择一些行，使得每一列<u>有且仅有一个1</u>. 例子如下：

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \tag{3}$$

对于如上所示的矩阵(3)，我们取行集合{1, 4, 5} 便可使每一列有且仅有一个1。

3.2 Solving an exact cover problem

很明显，exact cover problem 是一个无法用多项式算法解决的问题，解决方法只有搜索! 且看如下搜索代码：

```
if (A is empty) { \\A is the 0-1 matrix.
    the problem is solved;
    return ;
}
choose a column, c,
    that is unremoved and has fewest elements.;    ...(1)
remove c and each row i that A[i][c] == 1 from matrix A;    ...(2)
for (all row, r, that A[r][c] == 1) {
    include r in the partial solution.;
    for each j that A[r][j] == 1 {
        remove column j and each row i
            that A[i][j] == 1 from matrix A. ...(3)
    }
    repeat this algorithm recursively on the reduced matrix A.
    resume all column j and row i that was just removed; ...(4)
}
resume c and i we removed at step (2)   ...(5);
```

上述代码是一个普通的搜索过程，如果你觉得比较陌生，可能是因为跟你平时所用的搜索方法有些不同，但本质上必定是相同的。

我们要做的，是去优化这段代码，但这段代码的算法框架已经固定[1]，几乎没有可优化的余地，就算用非递归来优化也不会得到多少的效率提升，反而会使编码复杂度和出错率大大增加。

如何去实现代码行(1) (2) (3) (4) 是优化的关键，也是Dancing Links发挥其作用的地方。我们可以直观的去想到一个简单的方法：

Step (1): 把矩阵进行扫描，选出未做删除标记的列中元素个数最少的。如果你不理解为什么选最少的，可以当它是随便选取的，不影响正确性。

Step (2): 对当前列和行做标记，标记其已经删除。

Step (3): 同Step (2)

Step (4): 把相应的标记删除。

Step (5): 同Step (4)

[1]Step (1)中有一个巨大的优化，就是选取最少元素的列，但这是搜索优化问题，与我们讨论的重点Dancing Links没多大关系，固不做过多论述

然而，这个方法很明显是非常低效的，做标记是快速的。但是相应的带来的问题，是查找的时候会变得非常慢。复杂度始终是O(n)(n为矩阵大小)。在step (1) (2) (3) (4) (5) 中我们都用到了查找操作，所以我们不能忽视查找的复杂度。在这个实现方式中，把查找的效率提升多少，整个程序便可提升多少。

我们再做观察，还可发现，矩阵随着递归的深入便会变得越来越稀疏。而我们所对应的查找操作，却没有利用这一特性，一直在盲目的进行查找。

或许有读者会自然而然的想到链表。链表可以高效的支持删除操作，链表的长度也是随着删除操作的执行而变短，不会存的上述做法的问题。

从前面的论述，我们知道了程序的瓶颈在于查找操作。因为矩阵会越来越稀疏，所以我们可以直观的感觉到把链表用在此处，会及大的提高效率。

有了链表，我们的查找操作只用到了遍历链表，因为已经被标记删除的结点是不存在在链表中的。很显然用链表的话查找操作的效率已经提升到了极致，不可能再提升了。

有人可能会发现问题，我们在链表中删除结点是容易，可是恢复容易吗？Step (4) (5) 该如何实现呢？哈哈，请看本论文开头对双向链表的介绍，看似无用的双向链表的恢复操作，在这里不是正好用上吗？

好了，到这里，已经把Dancing Links基本介绍完了。如果你真正理解了，可能会惊叹，这么简单呀！这就是Dancing Links吗？但是Dancing Links的强大之处还得等到自己实践之后才能体会到。下面我们来细细展开Dancing Links。

3.3 The Dance Steps

一个比较好的实现方式是用数组来模拟链表，这样也可方便的建立矩阵，也可以加快运行速度。对每一个对象，记录如下几个信息：

- L[x], R[x], U[x], D[x], C[x];
- 双向十字链表用LRUD来记录，LR来记录左右方向的双向链表，UD来记录上下方向的双向链表。
- head 指向总的头指针，head通过LR来贯穿的列指针头。
- 每一列都有列指针头。C[x]是指向其列指针头的地址。行指针头可有可无，在我的实现中没有显示的表示出来。在某些题目中，加个行指针头还是很有必要的。
- 另外，开两个数组S[x], O[x]; S[x]记录列链表中结点的总数。O[x]用来记录搜索结果。

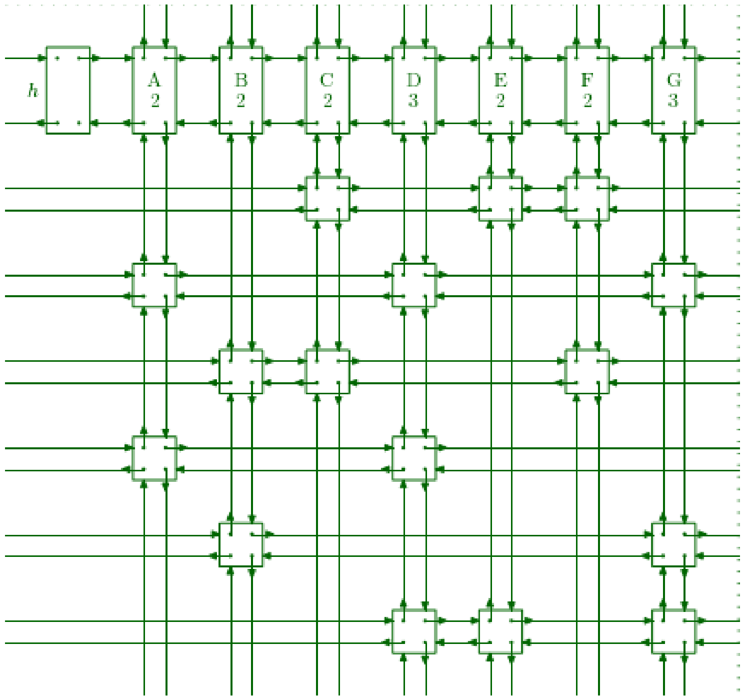

图 1: 矩阵A的表示

0-1矩阵(3)的存储图形如图(1)所示。

- h代表总的头链表head.
- ABCDEFG为列的指针头。

Exact Cover Problem的完整c代码如下所示，调用过程为dfs(0)。

```
void remove(const int &c) {
//remove column c and all row i that A[i][c] == 1
    L[R[c]] = L[c];
    R[L[c]] = R[c];
    //remove column c;
    for (int i = D[c]; i != c; i = D[i]) {
    //remove i that A[i][c] == 1
        for (int j = R[i]; j != i; j = R[j]) {
            U[D[j]] = U[j];
            D[U[j]] = D[j];
            --S[C[j]];
            //decrease the count of column C[j];
```

```
        }
    }
}

void resume(const int &c) {
    for (int i = U[c]; i != c; i = U[i]) {
        for (int j = L[i]; j != i; j = L[j]) {
            ++S[C[j]];
            U[D[j]] = j;
            D[U[j]] = j;
        }
    }
    L[R[c]] = c;
    R[L[c]] = c;
}

bool dfs(const int &k) {
    if (R[head] == head) {
        One of the answers has been found.
        return true;
    }
    int s(maxint), c;
    for (int t = R[head]; t != head; t = R[t]) {
    //select the column c which has the fewest number of element.
        if (S[t] < s) {
            s = S[t];
            c = t;
        }
    }
    remove(c);
    for (int i = D[c]; i != c; i = D[i]) {
        O[k] = i;//record the answer.
        for (int j = R[i]; j != i; j = R[j]) {
            remove(C[j]);
        }
        if (dfs(k + 1)) {
```

```
            return true;
        }
        for (int j = L[i]; j != i; j = L[j]) {
            resume(C[j]);
        }
    }
    resume(c);
    return false;
}
```

函数remove删除了矩阵中的列c及其对应的行。而函数resume则是还原它们，可能有些读者会注意到这两个函数里面顺序的问题。在这里我遵循的原则是先删除的后还原，后删除的先还原。当然，改变其中的某些顺序是不影响正确性的。这样写是为了遵循一定的原则。

另外还有一个小问题是我觉得应该提及的。在dfs过程中我们如何选择column c，是一个非常重要的问题。这个程序中我们选择的是元素最少的c，这个优化是显然的。

但如果是平局，我们该选择哪一个呢？在这个程序中，这一点并没有体现出来，如果是平局的话，那这段程序相当于随便选取了一列。这个问题在平时应该用可能会极大的影响效率。只能根据具体题目具体来判定如何解决了。做题的时候如果你发现你写的Dancing Links比别人的慢一些，那很有可能就是因为这里的问题了。

3.4 Hust Online Judge Problem 1017

http://acm.hust.edu.cn/thanks/problem.php?id=1017

这是一道exact cover problem的一个题目。想测试代码正确性的话可以在此提交测试。

4 Sudoku to exact cover problem

4.1 What is Sudoku?

Sudoku是一个最近几年非常流行的益智游戏。它的中文名是数独。

Sudoku问题是在一个9 * 9的格子上，内部有3 * 3的9个小块。你的目标是把每一个小格填上1-9中的某一个数，使得每一行、每一列、每一个小块都

包含1-9每个数各一次。在初始情况下，有一些小格是已经填好了的，你所要做的是把剩下的填好。

Sudoku的规则非常简单，但是却很有挑战性。

当今众多手机中已经预装了Sudoku游戏。很多linux桌面系统，比如Gnome，也预装了Sudoku。在网上也有很多Sudoku的网站，可以在网上搜索"Sudoku"来找到一些。今年的ICPC final总决赛的时候，每天的新闻纸背面也是一道数独题。答案在第二天的新闻纸中给出。

4.2 How to Solve Sudoku Puzzle?

Sudoku的解法很多，大体上是分为:

- 搜索
- 构造
- 搜索+构造

对于普通的9 * 9的Sudoku，最简单的搜索过程是肯定会超时的。我们需要加优化，搜索中比较常用的，优化搜索顺序。在这里可以起到非常关键的作用。

下面介绍一种搜索策略：

每次判断81个小格的可能性，从中选出可能性最少的来搜索。通过位运算等种种优化常数的方法。还是可以达到一个比较理想的效果的。pku3074可以在600ms左右用这种方法ac。

我们在这个策略的基础上增加一种搜索方式：

判断某个数在某个区域内的可能性。

加上这两个搜索方式后，Sudoku就可以比较快的出解了。对于9 * 9的Sudoku几乎是瞬出。pku3074可以做到200ms以内。（常数优化的比较好）。

对于Sudoku也的构造过程，这里不作过多介绍。Sudoku的构造方法非常多，但大多是人工构造的方法。用计算机去构造，是一个非常繁重的过程，根本不适合于OI和ICPC。

对于在搜索过程中加上部分构造过程增加效率。这里也不作过多介绍。方法也是很多，有兴趣的可以去网上查看相关论文。

据说今年MCM（美国数学建模竞赛）的B类题目就是Sudoku问题。听我们集训队的yyt说，他们便是写了一个构造Sudoku的程序得了一等奖，不过程序实在是太长了，1k行+？

4.3 Dancing Links在解决Sudoku这类问题上的优势

很多这种覆盖性的搜索问题都可以转化为exact cover problem。然后我们便可以用Dancing Links来解决之。这也是Dancing Links强大的地方。如果你认为Dancing Links只是解决一些特殊问题的特殊方法，那你就大错特错了。非常多问题可以转化成exact cover problem或类似于exact cover 的问题。

Dancing Links有什么优势呢？

- Dancing Links是一种通用的搜索方法，不用你自己去实现搜索过程。不用你对每一道题目都设计数据结构，来设计搜索方法。你所要做的只是去建一个模型。
- Dancing Links不用你自己去想优化搜索的方法。你在建模的和转化的过程中，Dancing Links往往已经为你想好了你都想不到的优化。
- Dancing Links可以减小常数，甚至可以在复杂度上去除一个系数。极大的提高了运行效率。
- Dancing Links代码非常短，方便在比赛现场敲，极大的提高了代码速度。也减少了debug的时间。

搜索问题千变万化，剪枝非常难想，就算想到了，也很难知道这是不是有用的剪枝。在用Dancing Links来解决这类问题的时候，你会发现这一类的很多问题都可以转化成你以前熟悉或做过的题目。那剪枝就不再是问题了。因为你已经把剪枝提交想好了。

实现也不再是问题了，只要转化正确，再加上Dancing Links的模版。代码正确性和实现速度都有了质的提升。做搜索题就仿佛做网络流题目一样了。

建模+ 模版== AC

哈哈，是不是很让人心动。那就来看一下如何转化sudoku吧。

4.4 转化模型

对于一个9 * 9的数独，建立如下的矩阵。:

行：

一共9 * 9 * 9 == 729行。一共9 * 9小格，每一格有9种可能性(1 - 9)，每一种可能都对应着一行。

列：

一共(9 + 9 + 9) * 9 + 81 == 324 种前面三个9分别代表着9行9列和9小块。乘以9的意思是9种可能，因为每种可能只可以选择一个。81代表着81个小格，限制着每一个小格只可以放一个地方。

这样我们把矩阵建立起来，把行和列对应起来之后，行i可以放在列j上就把A[i][j]设为1否则设为0。然后套用Exact Cover Problem的定义：选择一些行，使得每一列有且仅有一个1。哈哈，是不是对应着sudoku的一个解？

前面我已经说过Sudoku的搜索模型，现在再结合转化后的模型，你会不会觉得本质上是一样的呢？其实是一样的。

请注意每一列只能有一个1，而且必需有一个1。

我们把列分成两类的话，一类是代表着每一个小格的可能性，另一类是代表着每个区域的某个数的可能性。第一类是式子中的81，第二类是(9 + 9 + 9) * 9这一部分。

这样我们所选择的行就对应着答案，而且因为列的限制，这个答案也是符合Sudoku的要求的。

那你也有可能会说，Dancing Links的优化体现在哪里呢？试想，这个矩阵是非常大的(324 * 729)，如果不用Dancing Links来解，可能出解吗？

4.5 Pku Online Judge Problem 3076 3074

http://acm.pku.edu.cn/JudgeOnline/problem?id=3074

http://acm.pku.edu.cn/JudgeOnline/problem?id=3076

3074是一个9 * 9的数独。

3076是一个16 * 16的数独。

这两道题目用前文论述的方法来解，效率非常高。Problem Status里面前几名的AC Code基本都是用Dancing Links来解的。（不信你发mail问问他们咯）.

3076里面有些人是0ms AC的。而且代码长度非常短，我十分怀疑他们是直接交的数据。也希望POJ的管理员能够把数据变一下吧。

5 Exact cover problem 变种

5.1 Abstract

了解本节需要知道A*的一些知识。如果你没有做过有关A*的题目，可能在看本节的时候有些困难。因为篇幅，时间和精力都不允许我再去介绍A*。所以我下面所介绍的知识就默认你已经很精通A*啦，不会的集训队

队员自己去找资料吧。但是我推荐在你对A*有所了解之后，再回过头来看一下lrj的书对A*在竞赛中应用的介绍。

推荐学习A*所要做的题目：

- 第k短路
- 15数码
- 第k短简单路。

5.2 Description

我们把exact cover problem的定义改一下：在0-1矩阵中选取最少的行，使得每一列最少有一个1。

我们可以想成一种二分图的支配集的模型:
左边的点可以支配右边的点，然后我们要选最少的左边的点，使得右边的点都被支配了。

我们用A* + Dancing Links来解决此题。

A*是算法的框架，Dancing Links来实现数据结构，和优化常数。

A*中比较重要的是h函数的设计，h函数最好设计成离线的，因为计算h函数的复杂度也在很大程度上决定了程序的效率。那h函数如果想不出离线的呢？那我们就应该在保证h函数尽量单调的情况下，来减少计算h函数的常数。

因为h函数是与矩阵有关系的，所以对于稀疏的矩阵来说，选择Dancing Links是非常有必要的。随着递归的深入，Dancing Links的优势会体现的越来越明显。

5.3 H function

对于上节所描述的模型，我设计的h函数是这样的：

ans来记录h函数的数值。

对当前矩阵来说，选择一个未被控制的列，很明显该列最少需要1个行来控制，所以我把ans++。该列被控制后，我把它所对应的行，全部设为已经选择，并把这些行对应的列也设为被控制。继续选择未被控制的列，直到没有这样的列。

通过这样的操作，就可以求出一个粗略的h函数。

5.4 Method

有了h函数，剩下的就好说了。不管用A*也好，用IDA*也好，实现是比较简单的。

我的实现方法是IDA*，因为对于一个矩阵进行判重和hash还是比较麻烦的。再加上本题答案都比较小。所以IDA*是一个比较好的选择。

5.5 Pku Online Judge Problem 1084

http://acm.pku.edu.cn/JudgeOnline/problem?id=3076f

这道题目也是lrj书上一道习题。题目模型就是我上面所说的。

我写的程序实验可以过6 0这样的数据。推荐写个程序来实践一下。