An Extension of Knuth's Dancing Links Algorithm

by Saul Spatz

In [1] , Professor Donald E. Knuth explains the solution of certain "generalized exact cover" problems by algorithm DLX, or "dancing links." In this note, I describe an extension to this algorithm that allows solution of somewhat more general problems of this sort.

An exact cover problem can be described in terms of zero-one matrices. Given such a matrix, the problem is to choose a subset of the rows such that each column of the resulting submatrix contains exactly one 1. In [1], Knuth gives many examples of tiling problems of this type. Here we are to tile a given region, say an *n*-by-*m* rectangle, with a prescribed set of tiles, for example, the twelve pentominoes. The matrix representing this problem will have *mn*+12 columns, one for each cell of the rectangle and one for each tile. It will have one row for each possible way of placing a tile inside the rectangle. The row will have a 1 in the column of the tile that is placed, and a 1 in the column of each cell that is covered by the tile. A moment's thought shows that the exact cover problem is the same as the tiling problem: each tile must be used exactly once, and each cell must be covered by exactly one tile.

In the generalization that Knuth discusses, columns that must have exactly one 1 in them are called "primary" columns, but "secondary" columns, having at most one 1 in them are also allowed. For example, in the famous *n* queens problem of placing *n* nonattacking queens on an *n*-by-*n* chessboard, there is a primary column corresponding to each rank and each file, and a secondary column corresponding to each diagonal. If we are to place *n* nonattacking queens on the board, there will certainly be one

queen in each rank and each file, but the condition on diagonals is simply that there cannot be more than one queen in an diagonal. Since there are 2*n*-1 diagonals in each direction, it is not required that each diagonal have a queen.

In the extension I discuss here, we are also allowed to have columns where the requirement is that there be *at least* one 1. For example, suppose the problem is to find the smallest number of nonattacking queens that can be placed on an *n*-by-*n* chessboard so that each unoccupied square is under attack. Now there is a column for each rank, file, diagonal, and square. A row represents the placement of a queen in a square, and has a 1 in the columns of the rank, file, and diagonals of that square. It also has a 1 in the column of the square itself, and in the column of each square attacked by the queen. The columns for ranks, files, and diagonals must have at most one 1, or two queens would attack one another, and the columns for the squares must have at least one 1, so that the square is occupied or under attack.

The cornerstone of the DLX algorithm in [1] is this observation:

> Suppose x points to an element of a doubly-linked list; let $L[x]$ and $R[x]$ point to the predecessor and successor of that element. Then the operations
>
> $$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x]$$
>
> remove x from the list; every programmer knows this. But comparatively few programmers have realized that the subsequent operations
>
> $$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x$$

will put x back into the list again.

A sparse matrix structure is used to represent the problem, with both rows and columns doubly linked circular lists. The algorithm is described in complete detail in [1]; I will only touch on the high points here. Later on, I will give the details for the extended algorithm. The basic operations are "covering" a column, and "blocking" a row. When a column is covered, it is first deleted from the list of columns remaining to be covered, and then all its rows, that is, all rows having a 1 in this column, are blocked. When a row is blocked, each node in the row is removed from its column, except for the nodes in the column being covered. This allows us to find the elements in the column again, so we can backtrack.

Once the column has been covered, we try adding each of its rows to the solution in turn. When the row is added, each of is columns is covered. If the list of columns to be covered is now empty, we have found a solution. If not, we choose another column to cover. Each time, we choose the column that appears to be hardest to cover, that is, the one with the fewest 1's. If it has zero 1's, then we must backtrack, by uncovering columns and unblocking rows. This, of course, is where the observation about resetting the pointers is used. The treatment of primary and secondary columns is identical, except that the list of columns needing to be covered initially consists only of the primary columns.

To extend this algorithm to the case where at least one 1 is required in some of the columns, we need another kind of column. Rather than introduce tertiary columns, I will speak of "exact", "at most", and "at least" columns. The exact and

at most columns are dealt with precisely as in algorithm DLX, but at least columns need special treatment.

The list of columns to be covered initially comprises the exact columns and the at least columns. When an at least column is covered, we cannot block its rows, since it is permissible to have more than one 1 in the column. However, we must prevent this column from being covered a second time. If, when some other column is covered later in the algorithm, one of the rows in this column is selected, the column will be deleted a second time, with disastrous results. To prevent this, when an at least column is covered, each of its nodes is deleted from its row. This is in sharp contrast to the DLX algorithm, where nodes are continually deleted from and reinserted into their columns, but are never deleted from their rows.

A problem with not blocking the rows of the chosen column is that the same set of rows may be selected multiple times in different orders. To prevent this, we must block a row of an at least column when the row is added to the solution, but we must not unblock it when we backtrack past this selection. The rows are unblocked only when we uncover the column. When we block the row, we also replace the node we removed back in the row, so that once the row has been added to the solution, it looks the same, no matter what kind of column it came from.

There is a sight asymmetry in uncovering columns. A column can be covered under two different circumstances; when the column is chosen, or because the column has a 1 in a row that has just been added to the solution. The covering process is the same both times, although it distinguishes at least columns from the other two types. Uncovering is identical for exact and at most columns, no

matter why the column was covered. For an least column that was covered because it was chosen, when the time comes to uncover it, all it rows have been blocked, and it can be uncovered just like an exact column. If, however, the at least column was covered because a row was added to the solution, its rows have not been blocked, and we have only to reinsert its nodes back into their rows. We must either have two versions of the uncover function, or the uncover function must, in the case of an at least column, test the nodes to determine if they are in their rows, and take the appropriate action, reinserting the node in its row if it is absent, or unblocking the row if it is present.

The first alternative, two versions of the function, is a tiny bit more efficient, but the second is also worth noting. It is easy to modify the algorithm to provide a Monte Carlo estimate of the running time, as described in [2]. Backtracking is not needed if only one experiment is done, but typically, one performs the experiment several times, and takes the average of the outcomes as an estimate. It is more time-consuming to set up the sparse matrix than to do the experiment, so for multiple experiments, we want to backtrack to the initial matrix after each trial. However, in the Monte Carlo method, we only select one row from a column, not each row in turn. Therefore, to "unchoose" an at least column, we must use the second alternative for the uncover function, because only one of its rows will be blocked.

It is easy to tell if a node r has been deleted form its row. If $R[L[r]] \neq r$, the node has been deleted from its row. Of course, if it is the only node in its row, deletion will still leave $R[L[r]] = r$, but then replacing it in the row would have no effect anyway.

Here is the description of the algorithm, given as the description of DLX from [1], with modifications noted.

> One good way to implement algorithm [DLX] is to represent each 1 in the matrix *A* as a *data object x* with five fields *L*[*x*], *R*[*x*], *U*[*x*], *D*[*x*], *C*[*x*]. Rows of the matrix are doubly linked as circular lists via the *L* and *R* fields ("left" and "right"); columns are doubly linked as circular lists via the *U* and *D* fields ("up" and "down"). Each column also contains a special data object called its *list header*.
>
> The list headers are part of a larger object called a *column object*. Each column object *y* contains the fields *L*[*y*], *R*[*y*], *U*[*y*], *D*[*y*] and *C*[*y*] of a data object and two additional fields, *S*[*y*] ("size") and *N*[*y*] ("name"); the size is the number of 1s in the column, and the name is a symbolic identifier for printing the answers. The *C* field of each object points to the column object at the head of the relevant column.

For the extension, the column object also contains a field *T*[*y*] ("type") with a symbolic value of EXACT, ATLEAST, or ATMOST. The name field is not needed unless one wants to follow Knuth's convention of identifying a row by the names of its columns. Personally, I find it more natural to give names to the rows.

To continue the description from [1],

> The L and R fields of the list headers link together all columns that still need to be covered. This circular list also includes a special column object called the *root*, *h*,

which serves as a master header for all the active headers. …

Our … algorithm to find all exact covers can now be cast … as a recursive procedure *search*(*k*), which is invoked initially with $k = 0$:

> If $R[h] = h$, print the current solution (see below) and return.
> Otherwise, choose a column object c (see below).
> Cover column c (see below).
> For each $r \leftarrow D[c], D[D[c]], \ldots$, while $r \neq c$,
> set $O_k \leftarrow r$;
> for each $j \leftarrow R[r], R[R[r]], \ldots$, while $j \neq r$,
> cover column $C[j]$ (see below);
> *search*($k + 1$);
> set $r \leftarrow O_k$ and $c \leftarrow C[r]$;
> for each $j \leftarrow L[r], L[L[r]], \ldots$, while $j \neq r$,
> uncover column $C[j]$ (see below).
> [Unchoose] column c (see below) and return.

The operation of printing the solution, in Knuth's formulation, simply consists of printing the names of all the columns in the rows containing $O_0, O_1, \ldots, O_{k-1}$, and the operation of choosing a column is just finding the uncovered column with the smallest size field

The first for each loop in the above code must be modified for at least columns:

> For each $r \leftarrow D[c], D[D[c]], \ldots$, while $r \neq c$,
> if $T[y]$ = ATLEAST then
> set $R[L[r]] \leftarrow L[R[r] \leftarrow r$;
> block row r (see below);
> set $O_k \leftarrow r$;

etc.

The operation of covering column *c* must take account of the column type. It removes *c* from the header list, and if *c* is not an at least column, it blocks all of *c*'s rows. If *c* is an at least column, it removes all data objects in *c*'s list from their rows.

Set $L[R[c]] \leftarrow L[c]$ and $R[L[c]] \leftarrow R[c]$
If $T[c]$ = ATLEAST
    then for each $i \leftarrow D[c], D[D[c]], \ldots$, while $i \neq c$,
        set $L[R[i]] \leftarrow L[i]$ and $R[L[i]] \leftarrow R[i]$;
    else for each $i \leftarrow D[c], D[D[c]], \ldots$, while $i \neq c$,
        block row *i* (see below).

Blocking row *i* means removing it from all column lists, except the column list we started in.

For each $j \leftarrow R[i], R[R[i]], \ldots$, while $j \neq i$,
    set $U[D[j]] \leftarrow U[j]$, $D[U[j]] \leftarrow D[j]$,
    and set $S[C[j]] \leftarrow S[C[j]] - 1$.

As Knuth says, "Finally, we get to the point of this whole algorithm, the operation of uncovering a given column *c*. Here is where the links do their dance," and here is where a modification for the at least columns is needed.

If $T[c]$ = ATLEAST
    then for each $i \leftarrow U[c], U[U[c]], \ldots$, while $i \neq c$,
        set $L[R[i]] \leftarrow R[L[i]] \leftarrow i$;
    else for each $i \leftarrow U[c], U[U[c]], \ldots$, while $i \neq c$,
        for each $j \leftarrow L[i], L[L[i]], \ldots$, while $j \neq i$,
            set $S[C[j]] \leftarrow S[C[j]] + 1$,
            and set $U[D[j]] \leftarrow D[U[j]] \leftarrow j$.
Set $L[R[c]] \leftarrow R[L[c]] \leftarrow c$.

In the modified algorithm, we also need the unchoose operation. (In [1], the last line of the algorithm reads, "Uncover column *c* … and return.") This is the same as uncover, without the ATLEAST case,

For each $i \leftarrow U[c], U[U[c]], \ldots$, while $i \neq c$,
  for each $j \leftarrow L[i], L[L[i]], \ldots$, while $j \neq i$,
    set $S[C[j]] \leftarrow S[C[j]] + 1$,
    and set $U[D[j]] \leftarrow D[U[j]] \leftarrow j$.
Set $L[R[c]] \leftarrow R[L[c]] \leftarrow c$.

After all this, you may object that we haven't solved the problem of find the minimum number of nonattacking queens on an *n*-by-*n* chessboard that attack all the unoccupied squares. The algorithm finds all ways of placing nonattacking queens on the board so that each unoccupied square is under attack, without regard to minimization. However, minimizing the number of queens is a straightforward application of branch and bound, and requires only minimal; changes to the algorithm.

An interesting application of the at least columns comes from the problem of placing the eight major chess pieces – the king, queen, two rooks, two knights, and two bishops, with the bishops on squares of opposite colors, so that every square, including those occupied by the pieces, is under attack. According to [3], this problem was proposed in 1849, and remained unsolved until 1988.

A backtrack program for this problem is a bit awkward to implement, since placing a piece will block the attacks of previously-placed pieces, and squares that once were attacked no longer will be. An idea that will quickly occur to one is to ignore the blocking, and to find all placements of

the pieces so that each square is "weakly attacked" as it is called in [3]. This is a generalized exact cover problem, and any solution to the original problem is a solution to the generalized one, so we have only to inspect the solutions to see if any solves the original problem. It turns out that, up to rotations and reflections, there are only 813 solutions to the generalized problem, so it would be feasible to filter them by hand.

In the generalized problem, there is an exact column for each of the eight pieces, an at most column for each square, indicating that a piece is placed on that square, and at least column for each square, indicating that the square is attacked. A row represents placing a particular piece in a particular square, an therefore has a 1 in one exact column, one at most column, and a number of at least columns.

To account for rotations and reflections, as in [3] we can restrict the queen to one of the squares in the lower left quadrant, and the bishop traveling on white squares to squares below the minor (northeast to southwest) diagonal. However, because the rooks and the knights are indistinguishable, each solution will be encountered four times. We can avoid this by considering each possible placement of the "lexicographically least" rook and knight as a different problem, and accumulating the solutions to all these problems.
That is, for *r*1 from 1 to 63, and *k*1 from 1 to 63, solve the problem with rook1 on square *r*1, knight1 on square *k*1, rook2 allowed to occupy any of squares *r*1+1 to 64, and knight2 allowed to occupy any of squares *k*1+1 to 64. Accumulate all the solutions to these 3,969 problems, and filter out the solutions to the original problem, of which there turn out to be none.

In [3], a clever and elegant method is used to merge potential solutions, and to prune the search tree very efficiently. This results in a much faster program than the approach described above, but of course, it takes considerably longer to write the program than it does to code the problem and run it through a generalized exact cover solver. What's more, I for one, would never have come up with the clever idea at the heart of [3], so that for me at least, a more mechanical approach is not only more efficient, but necessary.

[1] Donald E. Knuth, "Dancing Links," preprint (1999), available from http://www-cs-staff.stanford.edu/~knuth/preprints.html#ref

[2] Donald E. Knuth, "Estimating the Efficiency of Backtrack Programs," *Math. Comp.* v. 29, 1975, pp. 121-136.

[3] A. D. Robinson, B. J. Hafner, and S. S. Skiena, Eight Pieces Cannot Cover a Chessboard," *Comp. J.*, v. 32, 1989, pp.567-570.