1. Intro. Kazimierz Zarankiewicz asked [*Colloquium Mathematicum* **2** (1951), 301] for the smallest N such that every $n \times n$ matrix of zeros and ones contains a 2×2 submatrix of ones. R. K. Guy [in *Theory of Graphs* (Academic Press, 1968), 119–150] considered generalizations of the problem to nonsqaure matrices and submatrices, and tabulated results for small cases. Here I simply generate clauses that are satisfiable if and only if there's an $m \times n$ matrix containing at least r 1s but no such 2×2 submatrix.

This problem is interesting because of its many symmetries: $m!$ ways to permute the rows, times $n!$ ways to permute the columns. (If $m = n$, we can also transpose the matrix.)

I remove many of the symmetries, by requiring that the rows are in lexicographic order (when restricted to the first p columns) and the columns are in lexicographic order (when read top-down and restricted to the first q rows).

Setting $p = n$ and $q = m$ gives the maximum constraints, but smaller values may provide satisfactory symmetry breaking with less total cost.

```
#define nmax 1000   /* upper bound on m × n */
#include <stdio.h>
#include <stdlib.h>
  int m, n, r, p, q;   /* command-line parameters */
  int count[2 * nmax];   /* used for the cardinality constraints */
  main(int argc, char *argv[])
  {
    register int i, j, ii, jj, k, mn, t, tl, tr, jl, jr;
    ⟨ Process the command line 2 ⟩;
    ⟨ Generate the clauses for the lexicographic row constraints 4 ⟩;
    ⟨ Generate the clauses for the lexicographic column constraints 5 ⟩;
    ⟨ Generate the clauses for the rectangle constraints 3 ⟩;
    ⟨ Generate the clauses for the cardinality constraints 6 ⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
  if (argc ≠ 6 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1 ∨ sscanf(argv[3], "%d",
        &r) ≠ 1 ∨ sscanf(argv[4], "%d", &p) ≠ 1 ∨ sscanf(argv[5], "%d", &q) ≠ 1) {
    fprintf(stderr, "Usage:␣%s␣m␣n␣r␣p␣q\n", argv[0]);
    exit(−1);
  }
  mn = m * n;
  if (mn > nmax) {
    fprintf(stderr, "Sorry:␣mn␣is␣%d,␣and␣I'm␣set␣up␣for␣at␣most␣%d!\n", mn, nmax);
    exit(−2);
  }
  if (p > n) {
    fprintf(stderr, "Parameter␣p␣should␣be␣at␣most␣n␣(%d),␣not␣%d!\n", n, p);
    exit(−3);
  }
  if (q > m) {
    fprintf(stderr, "Parameter␣q␣should␣be␣at␣most␣m␣(%d),␣not␣%d!\n", m, q);
    exit(−4);
  }
  printf("~␣sat-zarank␣%d␣%d␣%d␣%d␣%d\n", m, n, r, p, q);
```

This code is used in section 1.

3. ⟨ Generate the clauses for the rectangle constraints 3 ⟩ ≡

```
for (i = 0; i < m; i++)
  for (ii = i + 1; ii < m; ii++)
    for (j = 0; j < n; j++)
      for (jj = j + 1; jj < n; jj++) {
        printf("~%d.%d ~%d.%d ~%d.%d ~%d.%d\n", i, j, ii, j, i, jj, ii, jj);
      }
```

This code is used in section 1.

4. (See SAT-LEXORDER.) I choose *decreasing* order, because (a) fewer binary matrices with a given number of 1s (assumed less than mn/2) are doubly ordered when we do it this way; and (b) the connected components of the underlying bipartite graph are nicely revealed, as proved by Mader and Mutzbauer in 2001.

⟨ Generate the clauses for the lexicographic row constraints 4 ⟩ ≡

```
for (i = 1; i < m; i++) {
  for (k = 1; k ≤ p; k++) {
    if (k ≠ p) {
      if (k ≠ 1) printf("~R%d.%d", i, k − 1);
      printf(" R%d.%d %d.%d\n", i, k, i − 1, k − 1);
      if (k ≠ 1) printf("~R%d.%d", i, k − 1);
      printf(" R%d.%d ~%d.%d\n", i, k, i, k − 1);
    }
    if (k ≠ 1) printf("~R%d.%d", i, k − 1);
    printf(" %d.%d ~%d.%d\n", i − 1, k − 1, i, k − 1);
  }
}
```

This code is used in section 1.

5. ⟨ Generate the clauses for the lexicographic column constraints 5 ⟩ ≡

```
for (i = 1; i < n; i++) {
  for (k = 1; k ≤ q; k++) {
    if (k ≠ q) {
      if (k ≠ 1) printf("~C%d.%d", k − 1, i);
      printf(" C%d.%d %d.%d\n", k, i, k − 1, i − 1);
      if (k ≠ 1) printf("~C%d.%d", k − 1, i);
      printf(" C%d.%d ~%d.%d\n", k, i, k − 1, i);
    }
    if (k ≠ 1) printf("~C%d.%d", k − 1, i);
    printf(" %d.%d ~%d.%d\n", k − 1, i − 1, k − 1, i);
  }
}
```

This code is used in section 1.

6. Finally come the clauses that require at least r 1s in the matrix. As usual, I copy stuff from SAT-THRESHOLD-BB.

⟨ Generate the clauses for the cardinality constraints 6 ⟩ ≡

```
⟨ Build the complete binary tree with mn leaves 7 ⟩;
r = mn − r;    /* convert to asking for at most mn − r zeroes */
for (i = mn − 2; i; i−−) ⟨ Generate the clauses for node i 8 ⟩;
⟨ Generate the clauses at the root 9 ⟩;
```

This code is used in section 1.

7. The tree has $2mn-1$ nodes, with 0 as the root; the leaves start at node $mn-1$. Nonleaf node k has left child $2k+1$ and right child $2k+2$. Here we simply fill the *count* array.

⟨ Build the complete binary tree with *mn* leaves 7 ⟩ ≡

```
for (k = mn + mn − 2; k ≥ mn − 1; k−−) count[k] = 1;
for ( ; k ≥ 0; k−−) count[k] = count[k + k + 1] + count[k + k + 2];
if (count[0] ≠ mn) fprintf(stderr, "I'm␣totally␣confused.\n");
```

This code is used in section 6.

8. If there are t leaves below node i, we introduce $k=\min(r,t)$ variables Bi+1.j for $1\le j\le k$. This variable is 1 if (but not only if) at least j of those leaf variables are true. If $t>r$, we also assert that no $r+1$ of those variables are true.

```
#define x(k) printf("%d.%d", ((k) − mn + 1)/n, ((k) − mn + 1) % n)
```

⟨ Generate the clauses for node *i* 8 ⟩ ≡

```
{
  t = count[i], tl = count[i + i + 1], tr = count[i + i + 2];
  if (t > r + 1) t = r + 1;
  if (tl > r) tl = r;
  if (tr > r) tr = r;
  for (jl = 0; jl ≤ tl; jl++)
    for (jr = 0; jr ≤ tr; jr++)
      if ((jl + jr ≤ t) ∧ (jl + jr) > 0) {
        if (jl) {
          if (i + i + 1 ≥ mn − 1) x(i + i + 1);
          else printf("~B%d.%d", i + i + 2, jl);
        }
        if (jr) {
          printf("␣");
          if (i + i + 2 ≥ mn − 1) x(i + i + 2);
          else printf("~B%d.%d", i + i + 3, jr);
        }
        if (jl + jr ≤ r) printf("␣B%d.%d\n", i + 1, jl + jr);
        else printf("\n");
      }
}
```

This code is used in section 6.

9. Finally, we assert that at most r of the x's aren't true, by implicitly asserting that the (nonexistent) variable `B1.`r+1 is false.

⟨ Generate the clauses at the root 9 ⟩ ≡

```
tl = count[1], tr = count[2];
if (tl > r) tl = r;
for (jl = 1; jl ≤ tl; jl++) {
  jr = r + 1 − jl;
  if (jr ≤ tr) {
    if (1 ≥ mn − 1) x(1);
    else printf("~B2.%d", jl);
    printf("␣");
    if (2 ≥ mn − 1) x(2);
    else printf("~B3.%d", jr);
    printf("\n");
  }
}
```

This code is used in section 6.

10. Index.

argc: 1, 2.
argv: 1, 2.
count: 1, 7, 8, 9.
exit: 2.
fprintf: 2, 7.
i: 1.
ii: 1, 3.
j: 1.
jj: 1, 3.
jl: 1, 8, 9.
jr: 1, 8, 9.
k: 1.
m: 1.
main: 1.
mn: 1, 2, 6, 7, 8, 9.
n: 1.
nmax: 1, 2.
p: 1.
printf: 2, 3, 4, 5, 8, 9.
q: 1.
r: 1.
sscanf: 2.
stderr: 2, 7.
t: 1.
tl: 1, 8, 9.
tr: 1, 8, 9.
x: 8.

⟨ Build the complete binary tree with mn leaves 7 ⟩ Used in section 6.
⟨ Generate the clauses at the root 9 ⟩ Used in section 6.
⟨ Generate the clauses for node i 8 ⟩ Used in section 6.
⟨ Generate the clauses for the cardinality constraints 6 ⟩ Used in section 1.
⟨ Generate the clauses for the lexicographic column constraints 5 ⟩ Used in section 1.
⟨ Generate the clauses for the lexicographic row constraints 4 ⟩ Used in section 1.
⟨ Generate the clauses for the rectangle constraints 3 ⟩ Used in section 1.
⟨ Process the command line 2 ⟩ Used in section 1.

SAT-ZARANK

	Section	Page
Intro	1	1
Index	10	5