1. Intro. Find placements of at most r queens on an $m \times n$ board, so that (i) no row, column, or diagonal contains three queens; and (ii) adding any new queen would violate condition (i).

(This problem was mentioned in Gardner's column on October 1976, and discussed by Cooper, Pikhurko, Schmitt, and Warrington in *AMM* **121** (2014), 213–221. The latter authors said they did a "brute force search" to show unsatisfiability for $m = n = r = 11$, and it took 900 hours an 3-gigahertz computers. Naturally I wondered if SAT methods would be better.)

Variable $i.j$ means there's a queen on cell (i,j). Variable `R`i means there are two queens in row i. Variable `C`j means there are two queens in column j. Variable `A`d means there are two queens in the diagonal $i + j - 1 = d$, for $1 \le d < m + n$. Variable `D`d means there are two queens in the diagonal $i - j + n = d$, for $1 \le d < m + n$.

The constraints for condition (i) are obvious. For example, there are $\binom{n}{3}$ constraints for each row, saying that no three queens are present in that row.

The constraints for condition (ii) are that, if no queen is in (i,j), then either its `R` or `C` or `A` or `D` variable is true.

(Actually I decided to use a more complex naming scheme for the `R`, `C`, `A`, `D` variables; see below.)

```
#define max_m_plus_n 100    /* upper bound on m + n */
#define max_mn 10000    /* upper bound on mn */
#include <stdio.h>
#include <stdlib.h>
  int m, n, r;    /* command-line parameters */
  char name[max_m_plus_n][8];
  int ap[max_m_plus_n], dp[max_m_plus_n];
  int count[2 * max_mn];    /* used for the cardinality constraints */
  ⟨Subroutine 4⟩;
  main(int argc, char *argv[])
  {
    register int i, j, k, mn, p, t, tl, tr, jl, jr;
    ⟨Process the command line 2⟩;
    for (i = 1; i ≤ m; i++) ⟨Forbid three in row i 3⟩;
    for (j = 1; j ≤ n; j++) ⟨Forbid three in column j 5⟩;
    for (k = 1; k < m + n; k++) ⟨Forbid three in diagonal A_k 6⟩;
    for (k = 1; k < m + n; k++) ⟨Forbid three in diagonal D_k 7⟩;
    ⟨Generate the clauses for condition (ii) 8⟩;
    ⟨Generate clauses to ensure at most r queens 9⟩;
  }
```

2. ⟨ Process the command line 2 ⟩ ≡

```
if (argc ≠ 4 ∨ sscanf(argv[1], "%d", &m) ≠ 1 ∨ sscanf(argv[2], "%d", &n) ≠ 1 ∨ sscanf(argv[3], "%d", &r) ≠ 1)
  {
  fprintf(stderr, "Usage:␣%s␣m␣n␣r\n", argv[0]);
  exit(−1);
}
if (m ≡ 1 ∨ n ≡ 1) {
  fprintf(stderr, "m␣and␣n␣must␣be␣2␣or␣more!\n");
  exit(−2);
}
if (m + n ≥ max_m_plus_n) {
  fprintf(stderr, "m+n␣must␣be␣less␣than␣%d!\n", max_m_plus_n);
  exit(−3);
}
mn = m * n;
if (mn ≥ max_mn) {
  fprintf(stderr, "m*n␣must␣be␣less␣than␣%d!\n", max_mn);
  exit(−3);
}
printf("~␣sat-nothree␣%d␣%d␣%d\n", m, n, r);
```

This code is used in section 1.

3. Sinz's clauses for the cardinality constraints work well for this application. In row i, there are variables iRj and iRj' for $1 \le j < n$, meaning that $x_{i1} + \cdots + x_{ij} \ge 1$ and $x_{i1} + \cdots + x_{i(j+1)} \ge 2$, respectively, The variable iR$n-1$' corresponds to what was called Ri above.

⟨ Forbid three in row i 3 ⟩ ≡

```
{
  for (j = 1; j ≤ n; j++) sprintf(name[j − 1], "%d.%d", i, j);
  forbid(i, 'R', n);
}
```

This code is used in section 1.

4. ⟨ Subroutine 4 ⟩ ≡

```
void forbid(int i, char t, int n)
{
  register j;
  for (j = 1; j < n; j++) {
    if (j < n - 1) {
      printf("~%d%c%d %d%c%d\n", i, t, j, i, t, j + 1);
      printf("~%d%c%d' %d%c%d'\n", i, t, j, i, t, j + 1);
      printf("~%s ~%d%c%d'\n", name[j + 1], i, t, j);
    }
    printf("~%s %d%c%d\n", name[j - 1], i, t, j);
    printf("~%d%c%d %s", i, t, j, name[j - 1]);
    if (j > 1) printf(" %d%c%d\n", i, t, j - 1);
    else printf("\n");
    printf("~%s ~%d%c%d %d%c%d'\n", name[j], i, t, j, i, t, j);
    printf("~%d%c%d' %d%c%d\n", i, t, j, i, t, j);
    printf("~%d%c%d' %s", i, t, j, name[j]);
    if (j > 1) printf(" %d%c%d'\n", i, t, j - 1);
    else printf("\n");
  }
}
```

This code is used in section 1.

5. ⟨ Forbid three in column j 5 ⟩ ≡

```
{
  for (i = 1; i ≤ m; i++) sprintf(name[i - 1], "%d.%d", i, j);
  forbid(j, 'C', m);
}
```

This code is used in section 1.

6. ⟨ Forbid three in diagonal A_k 6 ⟩ ≡

```
{
  p = 0;
  for (i = 1; i ≤ m; i++) {
    j = k + 1 - i;
    if (j ≥ 1 ∧ j ≤ n) {
      sprintf(name[p], "%d.%d", i, j);
      p++;
    }
  }
  forbid(k, 'A', p);
  ap[k] = p;
}
```

This code is used in section 1.

7. ⟨ Forbid three in diagonal D_k 7 ⟩ ≡

```
{
  p = 0;
  for (i = 1; i ≤ m; i++) {
    j = i + n − k;
    if (j ≥ 1 ∧ j ≤ n) {
      sprintf(name[p], "%d.%d", i, j);
      p++;
    }
  }
  forbid(k, 'D', p);
  dp[k] = p;
}
```

This code is used in section 1.

8. ⟨ Generate the clauses for condition (ii) 8 ⟩ ≡

```
for (i = 1; i ≤ m; i++)
  for (j = 1; j ≤ n; j++) {
    printf("%d.%d␣%dR%d'␣%dC%d'", i, j, i, n − 1, j, m − 1);
    if (ap[i + j − 1] > 1) printf("␣%dA%d'", i + j − 1, ap[i + j − 1] − 1);
    if (dp[i − j + n] > 1) printf("␣%dD%d'", i − j + n, ap[i − j + n] − 1);
    printf("\n");
  }
```

This code is used in section 1.

9. The clauses of SAT-THRESHOLD-BB are used for the $\le r$ constraint.

⟨ Generate clauses to ensure at most r queens 9 ⟩ ≡

```
⟨ Build the complete binary tree with mn leaves 10 ⟩;
for (i = mn − 2; i; i−−) ⟨ Generate the clauses for node i 11 ⟩;
⟨ Generate the clauses at the root 12 ⟩;
```

This code is used in section 1.

10. 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 10 ⟩ ≡

```
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 9.

11. 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 xbar(k) printf("~%d.%d", (((k) − mn + 1)/n) + 1, (((k) − mn + 1) % n) + 1)

⟨Generate the clauses for node i 11⟩ ≡
  {
    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) xbar(i + i + 1);
            else printf("~B%d.%d", i + i + 2, jl);
          }
          if (jr) {
            printf("␣");
            if (i + i + 2 ≥ mn − 1) xbar(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 9.

12. 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 12⟩ ≡
  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) xbar(1);
      else printf("~B2.%d", jl);
      printf("␣");
      if (2 ≥ mn − 1) xbar(2);
      else printf("~B3.%d", jr);
      printf("\n");
    }
  }
```

This code is used in section 9.

13. Index.

SAT-NOTHREE