July 6, 2007 at 10:31

1. Intro. I want to investigate the number of self-avoiding walks (saws) on a finite rectangular grid. The input for this program will be the size (m, n) of the grid. The output will be the number of saws from (x, y) to (x', y'), for each possible starting and stoping positions.

I do not know any efficient algorithm to solve this problem. I will pose the condition $mn \le 20$. I hope that even the results for such small grids might provide some insight.

There seems to be some research in mathematics on this and on related subjects. For the case when the grid is a square and the saws start from a corner and stop on the opposite corner, the number of walks is $1, 2, 12, 184, \ldots$ This is the sequence A007764. Doron Zeilberger did some work in 2001. Alas, he's using the comercial MAPLE program to do the computations, which means I can't check them. (I prefer reading the program to reading the article but I much prefer being able to run the program too.) I did not read any of the articles and I doubt I shall ever. I want to see what I can do on my own.

2. The state space is the set of previously visited positions plus the current position. The moves are up, right, down, and left; a move is legal if it does not leave the grid and does not go to a previously visited position. The position (x, y) is represented by the integer $nx + y$. The set of visited positions is encoded in the bits of an integer: The position p was previously visited iff $s \mathbin{\&} (1 \ll p)$. The size of the state space is $2^{mn}mn \le 20{,}971{,}520$.

Without loss of generality we can restrict the starting point to the lower left quadrant. This will make the program faster.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
  typedef int restype;    /∗ the type of the result; I might change it to long long later ∗/
  int m, n;    /∗ the size of the grid is m × n ∗/
  int stop;    /∗ the stop position; (x, y) is represented as n ∗ x + y ∗/
  restype cache[20][1 ≪ 20];
    /∗ cache[p][s] is the number of paths from p to stop that do not use positions in the set s, or −1 ∗/
  restype solve(int pos, int seen)
  {
    if (pos ≡ stop) return 1;
    if (seen & (1 ≪ pos)) return 0;
    if (cache[pos][seen] ≠ −1) return cache[pos][seen];
    seen |= 1 ≪ pos;
    ⟨Sum the number of paths from the neighbors of pos to stop and return 3⟩;
  }
  int main( )
  {
    scanf("%d␣%d\n", &m, &n);
    if (m ∗ n ≤ 0 ∨ m ∗ n > 20) {
      fprintf(stderr, "The␣size␣of␣the␣grid␣(%dx%d)␣should␣be␣>0␣and␣<=20.\n", m, n);
      return 1;
    }
    for (int x = 0; x < (m + 1)/2; ++x)
      for (int y = 0; y < (n + 1)/2; ++y)
        for (stop = 0; stop < m ∗ n; ++stop) {
          memset(cache, −1, sizeof (cache));
          printf("(%d,%d)->(%d,%d)=%d\n", x, y, stop/n, stop % n, solve(n ∗ x + y, 0));
        }
  }
```

3. The problem is now only a matter of looking at the ≤ 4 neighbors and summing the results for them.

⟨ Sum the number of paths from the neighbors of *pos* to *stop* and return 3 ⟩ ≡

```
restype &r = cache[pos][seen] = 0;     /∗ the result ∗/
if (pos + n < m ∗ n) r += solve(pos + n, seen);
if (pos − n ≥ 0) r += solve(pos − n, seen);
if (pos % n) r += solve(pos − 1, seen);
if ((pos + 1) % n) r += solve(pos + 1, seen);
return r;
```

This code is used in section 2.

4. Index.

⟨ Sum the number of paths from the neighbors of *pos* to *stop* and return 3 ⟩ Used in section 2.

SELFAVOID