Solving Klotski

Karl Wiberg <kha@treskal.com>

Wednesday, August 12, 2009

1 Introduction

Klotski is a sliding block puzzle: You have ten wooden blocks of different sizes to slide around on a game board, so that the largest block gets to the exit. I had one of these puzzles as a kid, but never managed to solve it; more recently, I bought one as a birthday present for one of my brothers. The passing years had not noticeably improved my ability to solve the puzzle by hand, but I do have a dregree in computer science now, and it struck me that Klotski looked like it might have a state space small enough to make a simple brute-force search feasible.

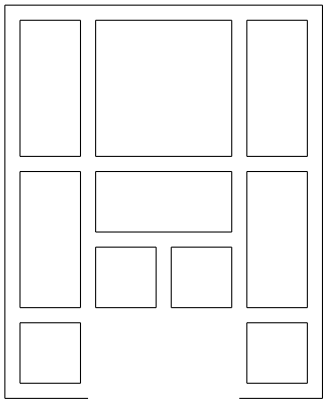


Figure 1: The starting position.

2 The rules

The game consists of ten wooden blocks: four 1×1 squares, five 1×2 rectangles, and one 2×2 rectangle. They are arranged on a 4×5 board with an exit at the bottom; the starting position is shown in figure 1.

Note that at any time, two of the twenty spaces on the board are empty.

The object of the game is to slide the blocks around until the big square is positioned bottom center, so that it may exit the playing area.

One move consists of sliding a block either up, down, left, or right into an adjacent empty space on the board.[1] The blocks cannot rotate, and they may not exit the board (the game ends right *before* the big square can exit).

3 Solving it

I wrote a small program in Haskell that solves the game by trying every possible move in a breadth-first search, stopping at the first winning position. Because it examines all positions reachable in k or fewer moves before going on to examine positions reachable in $k + 1$ moves, it is not only guaranteed to find a solution if one exists; the solution is also guaranteed to be (one of) the shortest. (This is basically Dijkstra's algorithm.)

Including the code that draws the diagrams in this paper, and comments and blank lines,

[1] If the two empty spaces are adjacent, it is sometimes possible to slide a block two steps in a single movement; for the purposes of this paper, such a maneuver counts as two moves.

it is less than 160 lines (see section 5). This is largely because I did not have to make any clever optimizations whatsoever—as soon as I got it to work at all, it found the solution in about three seconds.[2]

This is because the number of possible states accessible from the initial state is indeed very small; as the program tries every possible move that does not take it to an already visited position, the number of new positions accessible for each additional move is never more than about 750. After 167 moves we have visited all 25,955 reachable positions (but we find the first winning positions after 116 moves, and can stop searching there if all we want is to solve the game).

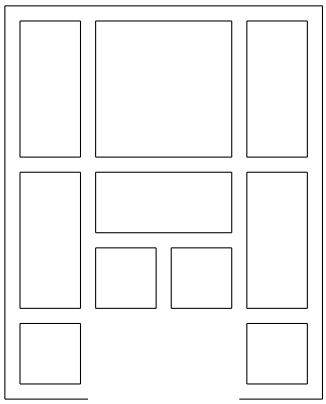
In contrast, the same search strategy applied to a game such as chess, which has a much higher branching factor, would see the number of accessible positions keep multiplying by about 30 every time we increased the number of moves by one. The total number of reachable positions has been estimated to be about 10^{50}, an intractably large number. Real chess programs search only a fraction of this vast state space; the trick to making a strong program is to make it search the right fraction.

Section 4 lists the complete solution found by the program. Of course, many other solutions exist, including those that have the same number of moves as this one; but as discussed above, there can be no solutions with fewer moves.

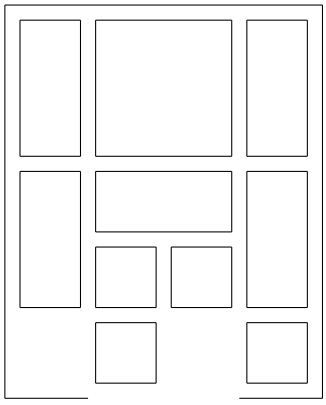
[2] It has been said that a programmer is a person who will happily spend a day writing a program that solves a one-hour problem in one second ...

4 Solution

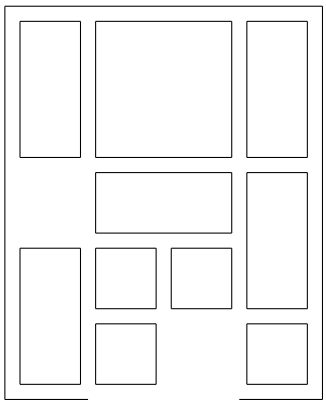
After 0 steps



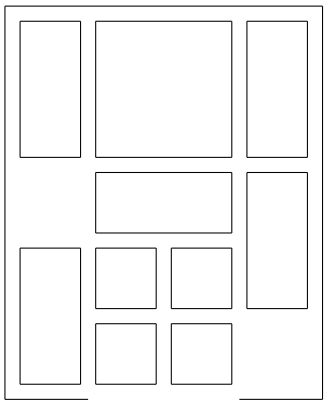
After 1 steps



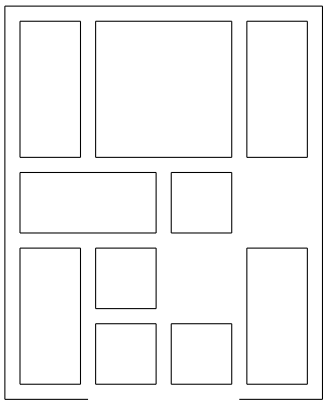
After 2 steps



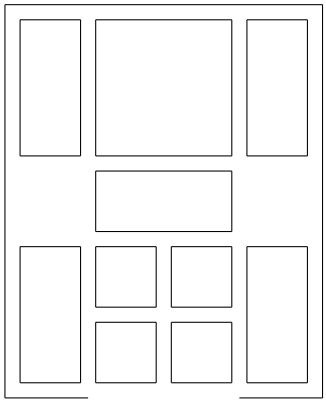
After 3 steps



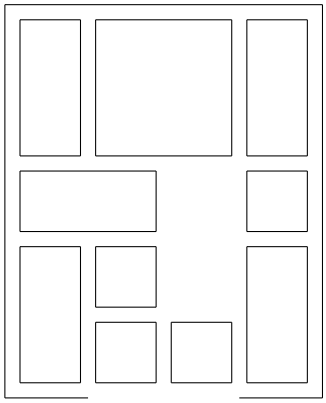
After 6 steps



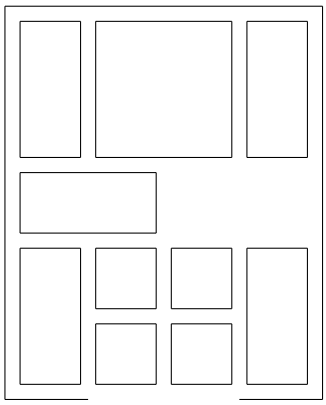
After 4 steps



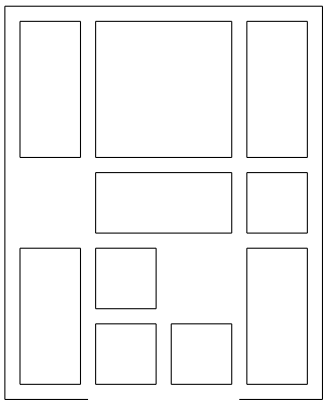
After 7 steps



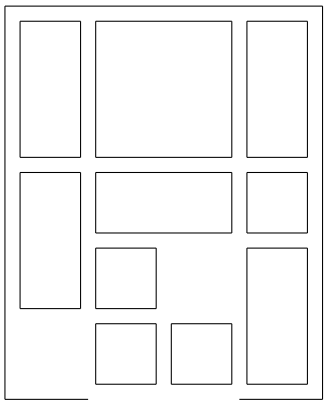
After 5 steps



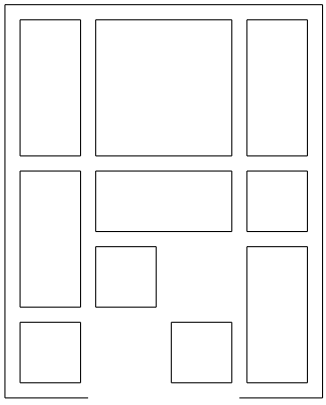
After 8 steps



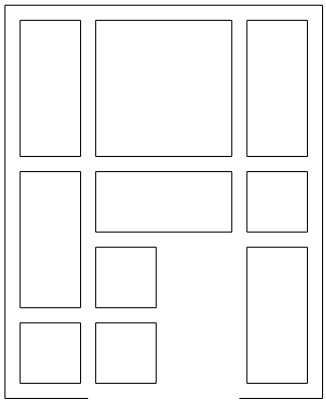
After 9 steps



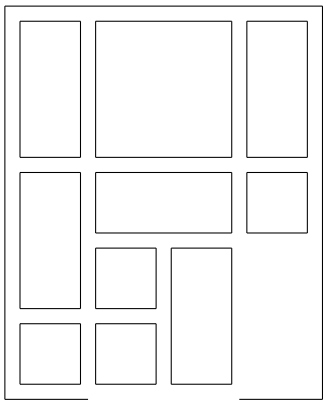
After 10 steps



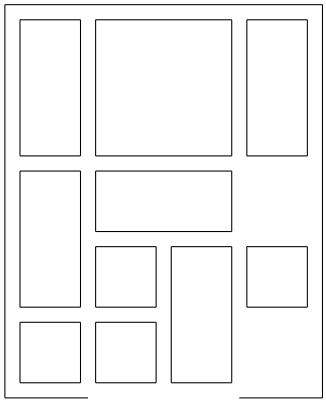
After 11 steps



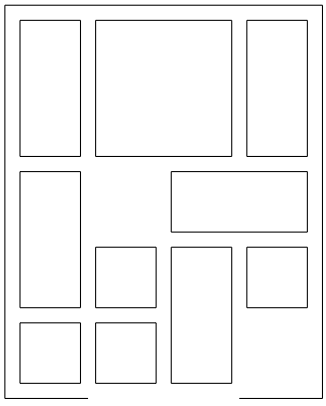
After 12 steps



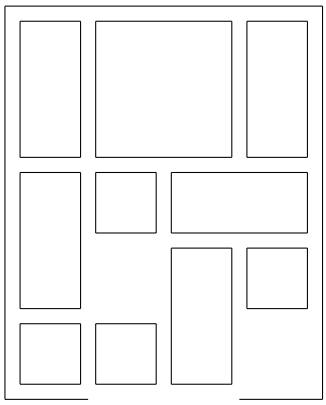
After 13 steps



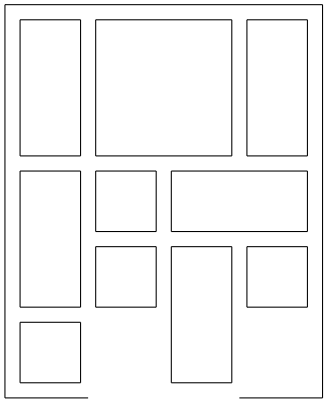
After 14 steps



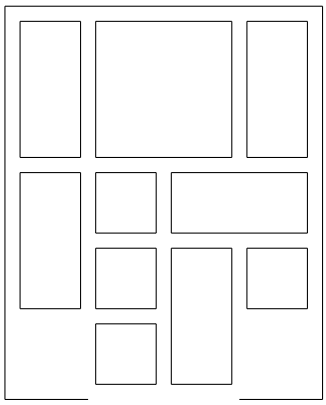
After 15 steps



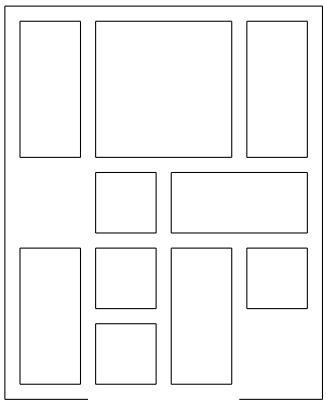
After 16 steps



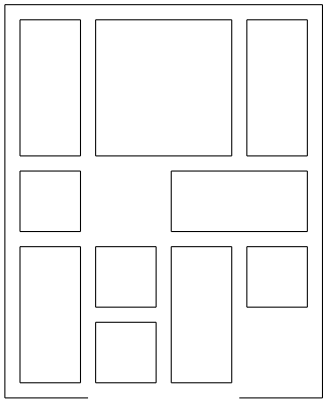
After 17 steps



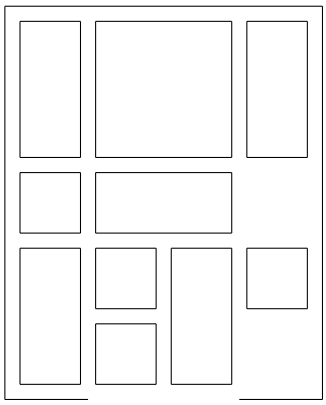
After 18 steps



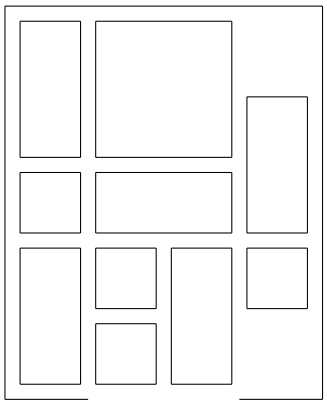
After 19 steps



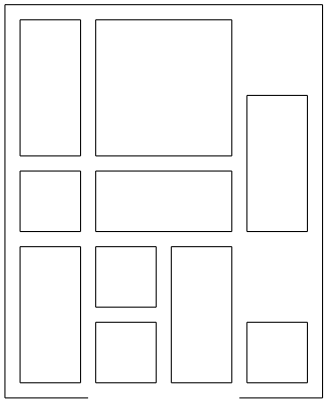
After 20 steps



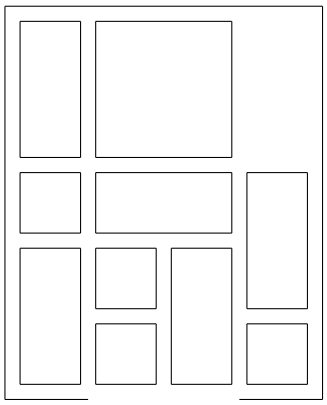
After 21 steps



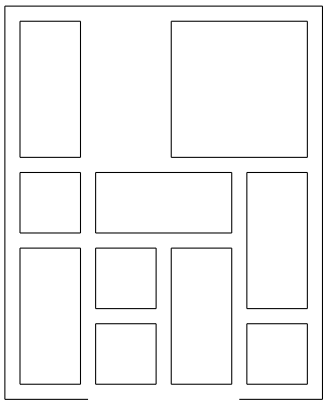
After 22 steps



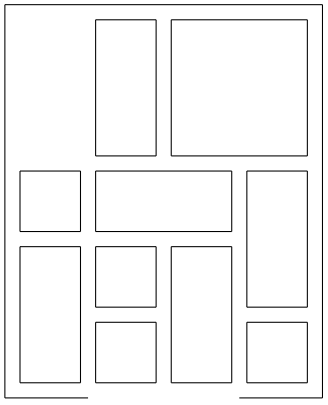
After 23 steps



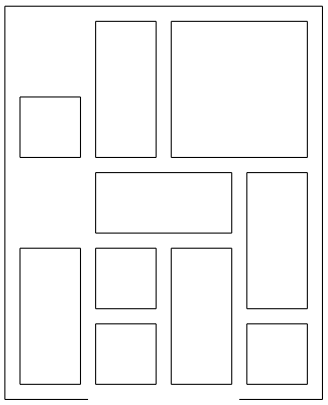
After 24 steps



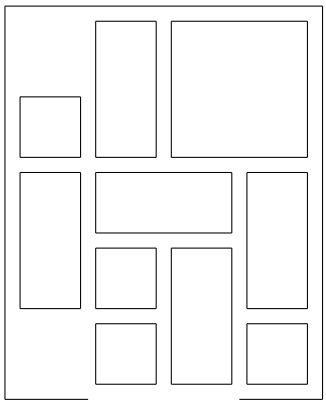
After 25 steps



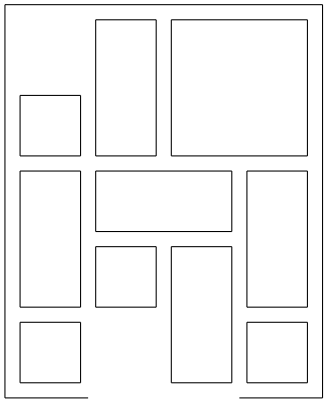
After 26 steps



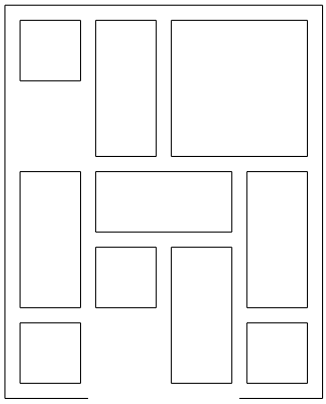
After 27 steps



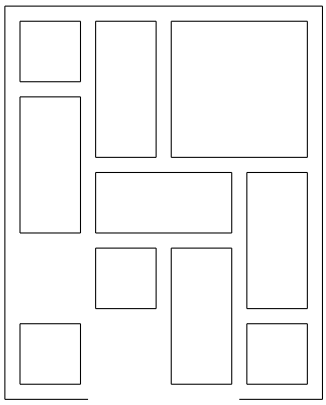
After 28 steps



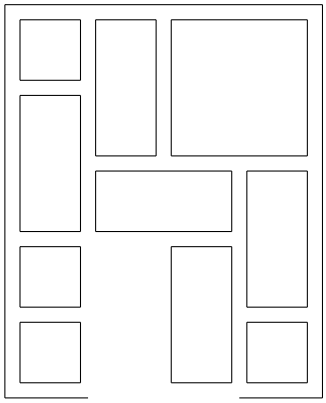
After 29 steps



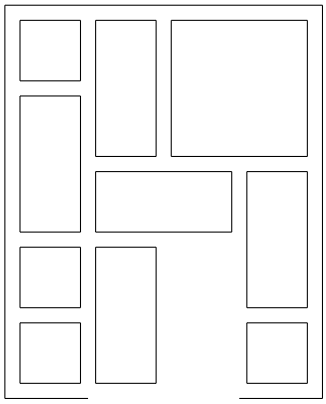
After 30 steps



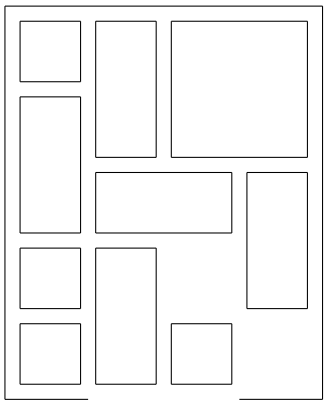
After 31 steps



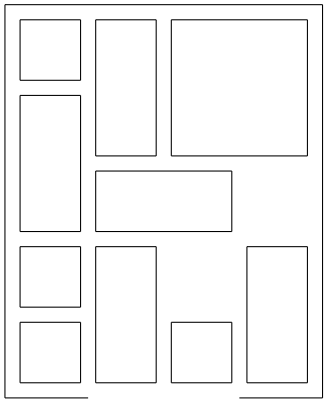
After 32 steps



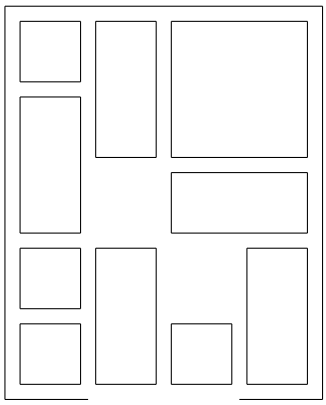
After 33 steps



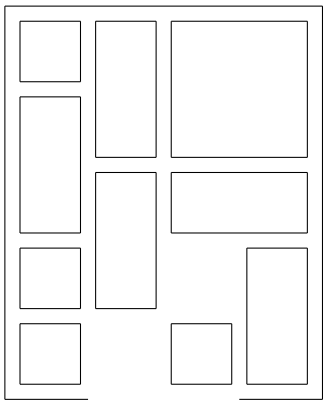
After 34 steps



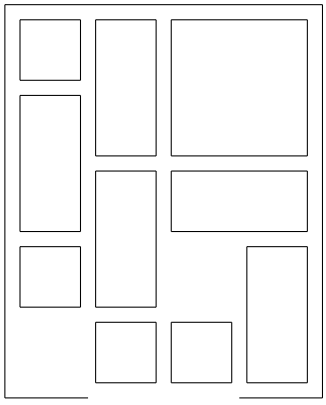
After 35 steps



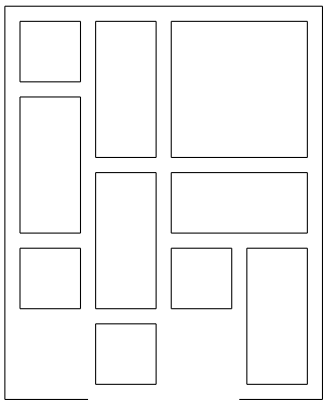
After 36 steps



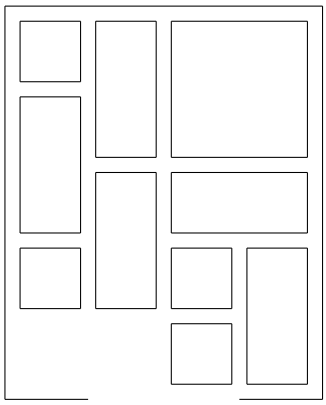
After 37 steps



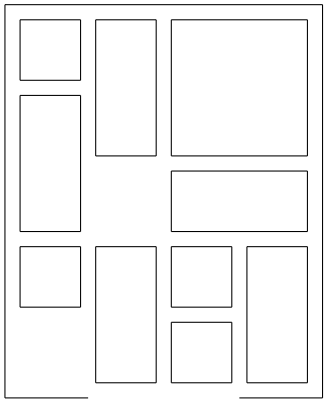
After 38 steps



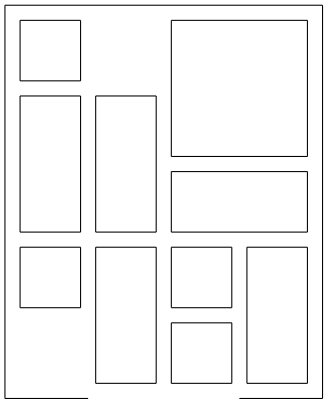
After 39 steps



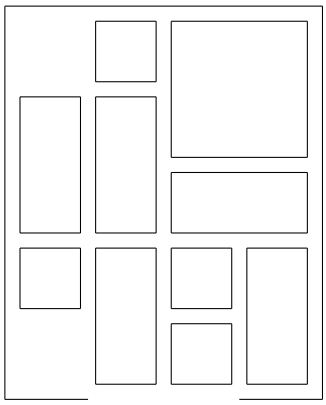
After 40 steps



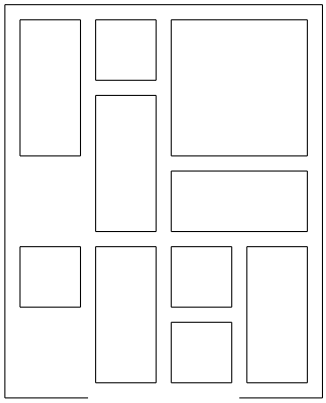
After 41 steps



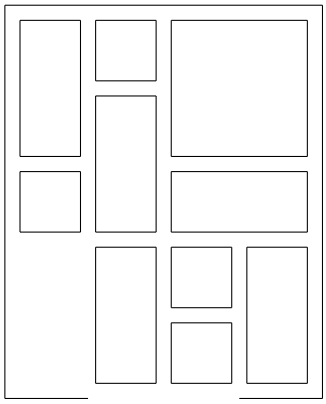
After 42 steps



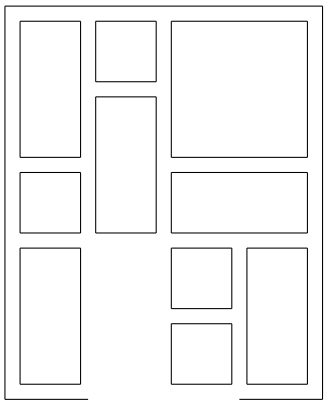
After 43 steps



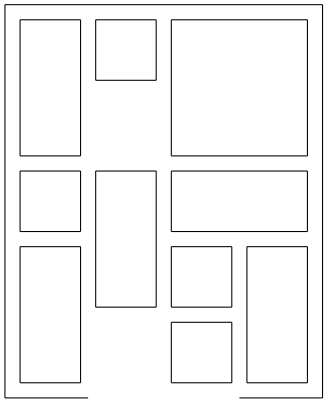
After 44 steps



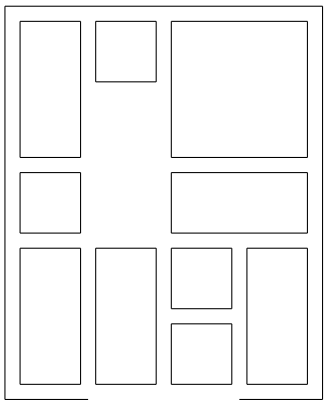
After 45 steps



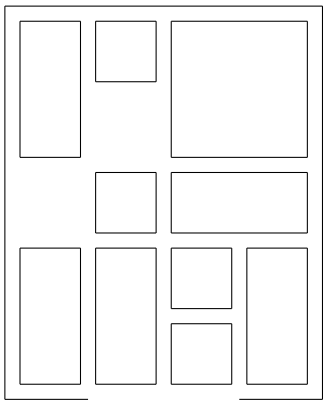
After 46 steps



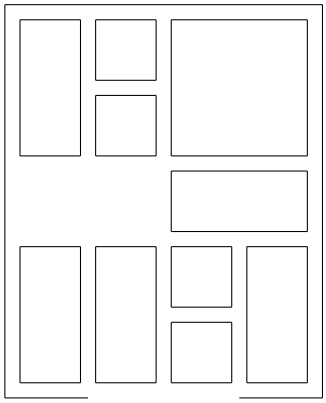
After 47 steps



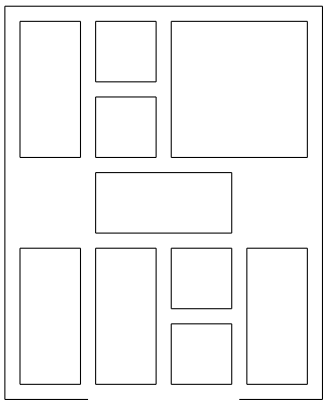
After 48 steps



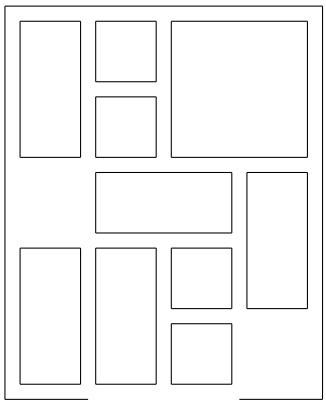
After 49 steps



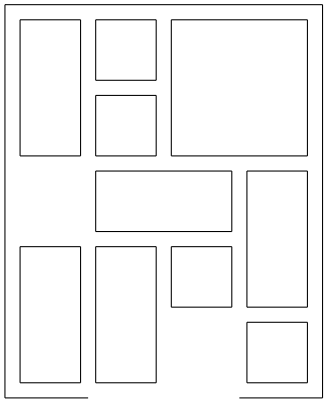
After 50 steps



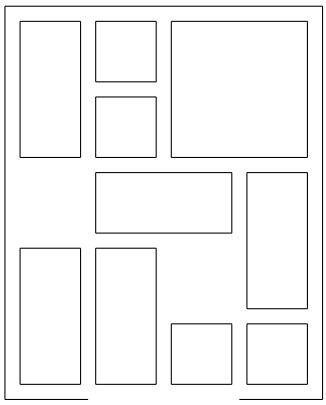
After 51 steps



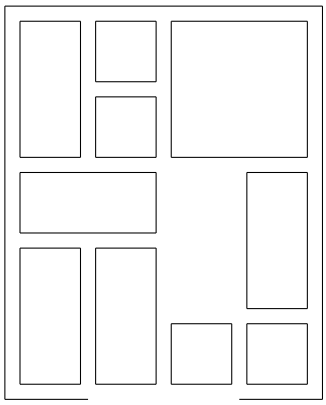
After 52 steps



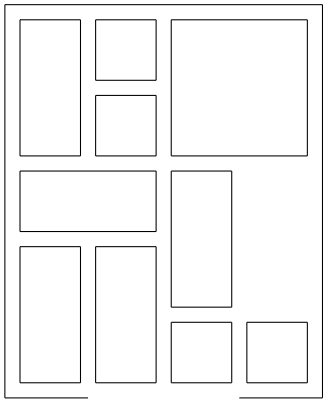
After 53 steps



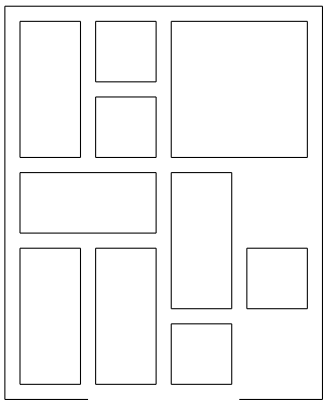
After 54 steps



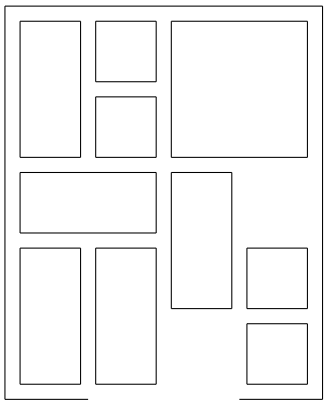
After 55 steps



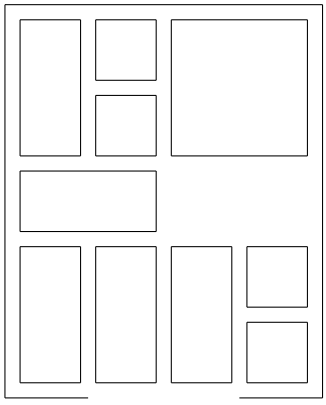
After 56 steps



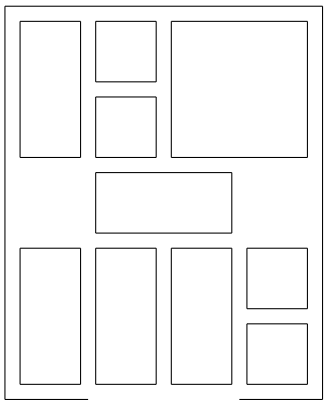
After 57 steps



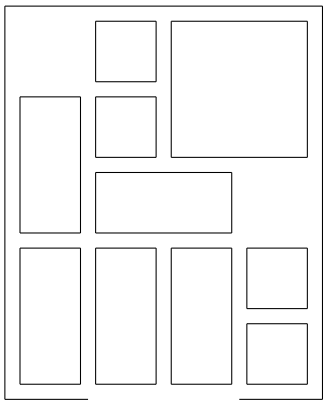
After 58 steps



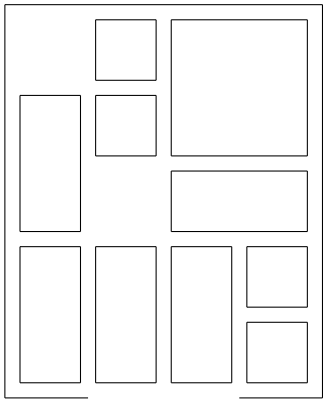
After 59 steps



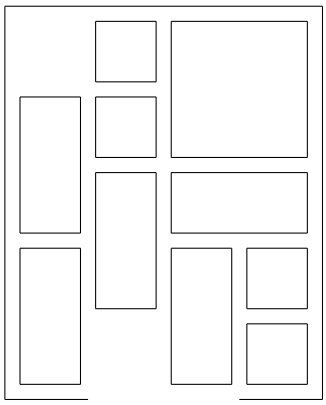
After 60 steps



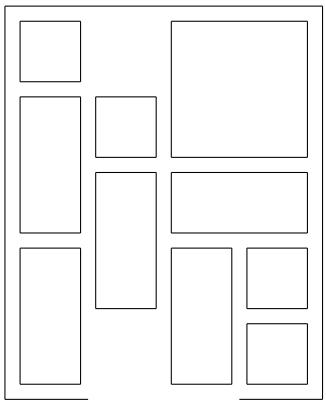
After 61 steps



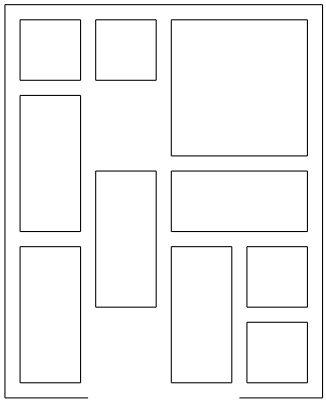
After 62 steps



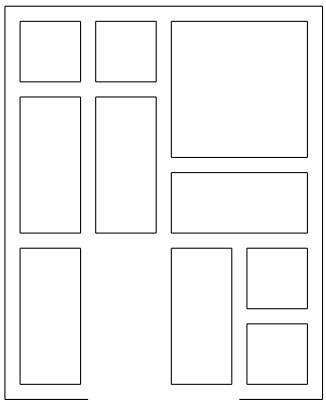
After 63 steps



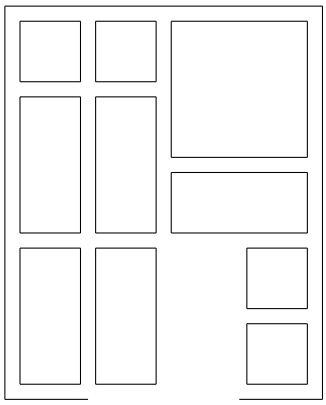
After 64 steps



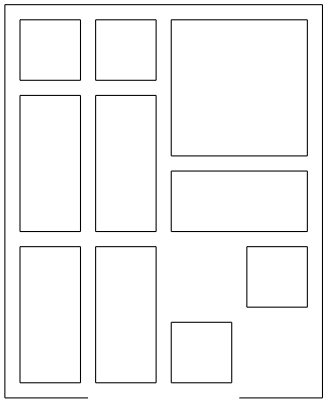
After 65 steps



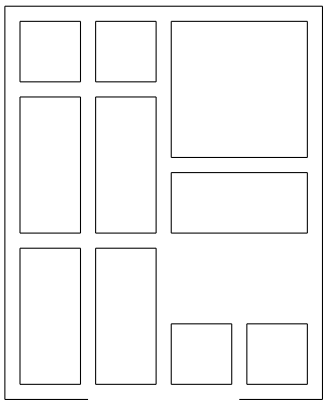
After 66 steps



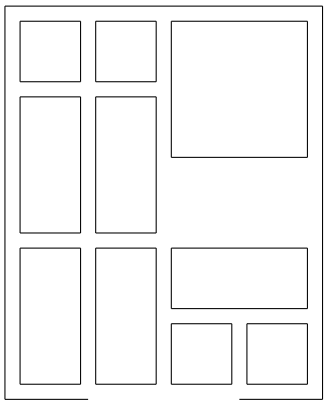
After 67 steps



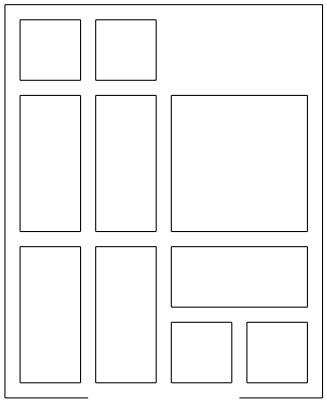
After 68 steps



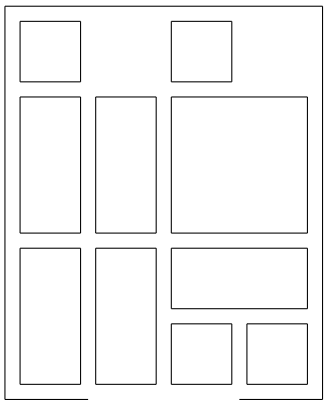
After 69 steps



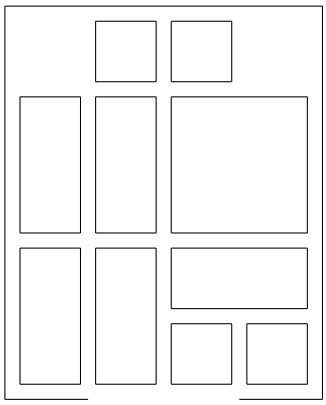
After 70 steps



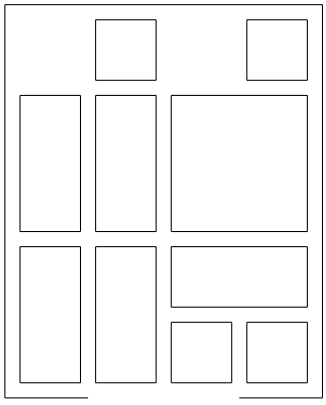
After 71 steps



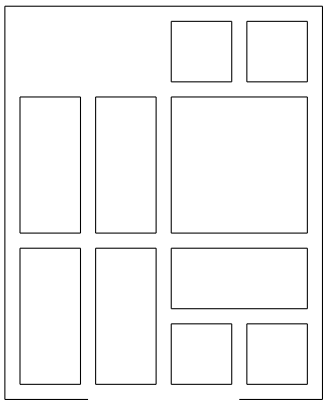
After 72 steps



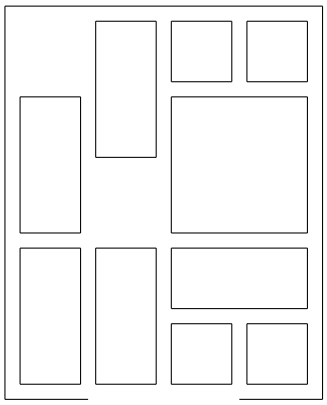
After 73 steps



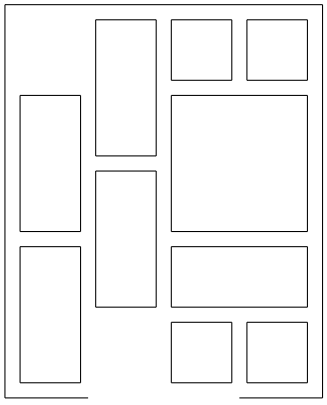
After 74 steps



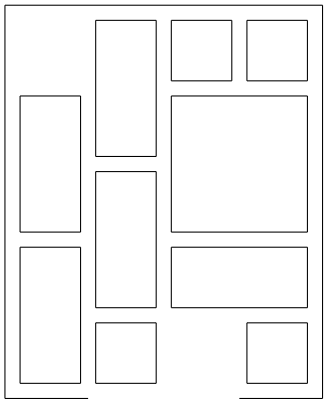
After 75 steps



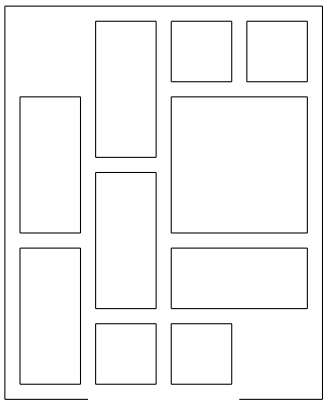
After 76 steps



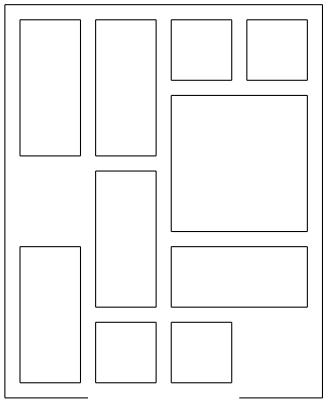
After 77 steps



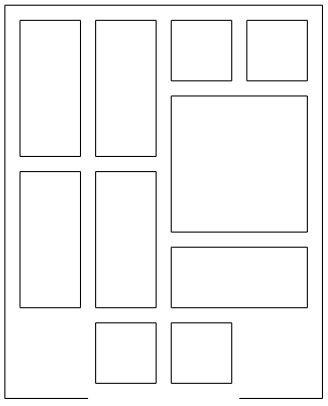
After 78 steps



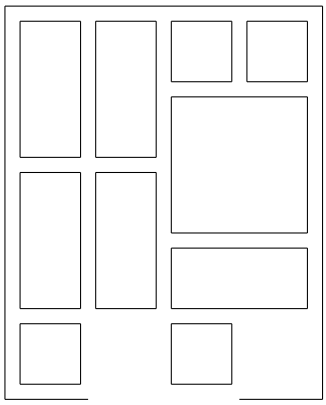
After 79 steps



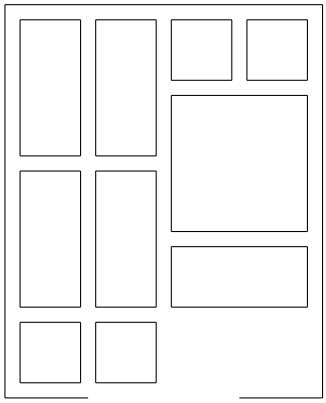
After 80 steps



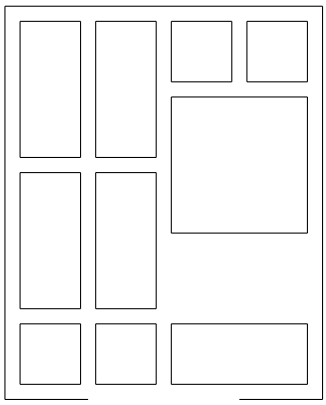
After 81 steps



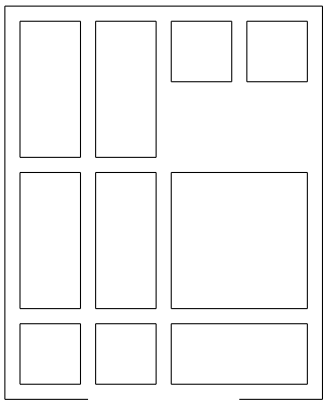
After 82 steps



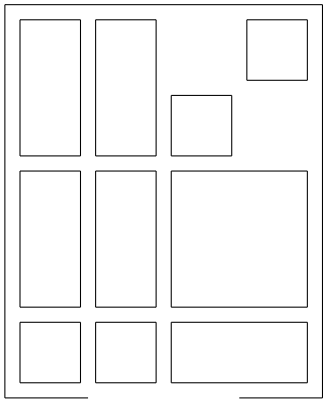
After 83 steps



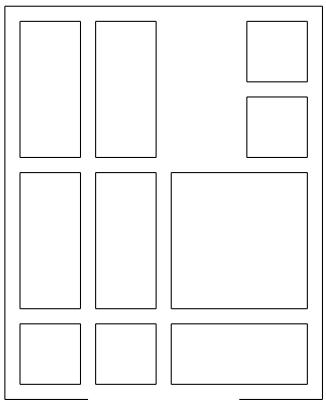
After 84 steps



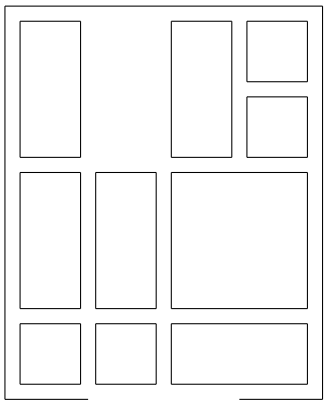
After 85 steps



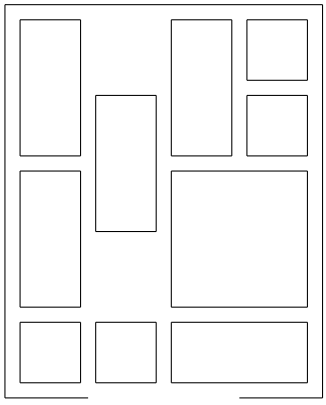
After 86 steps



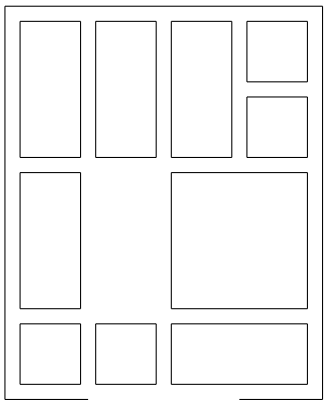
After 87 steps



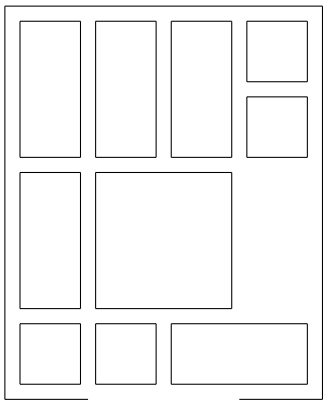
After 88 steps



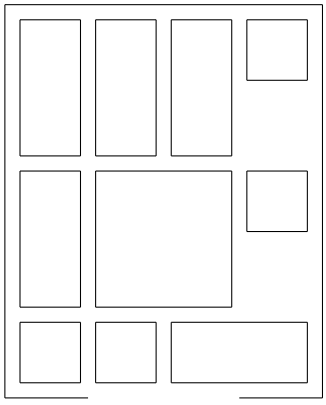
After 89 steps



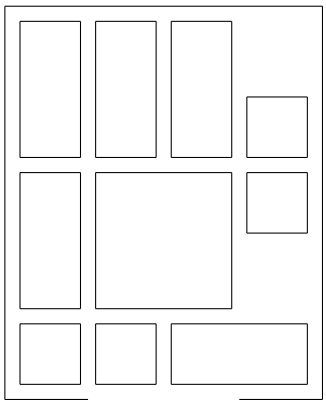
After 90 steps



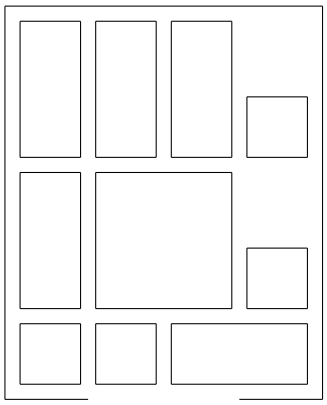
After 91 steps



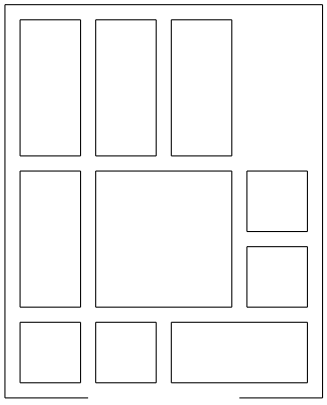
After 92 steps



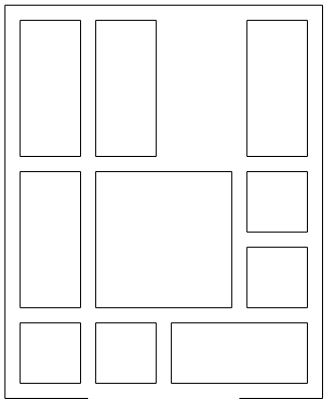
After 93 steps



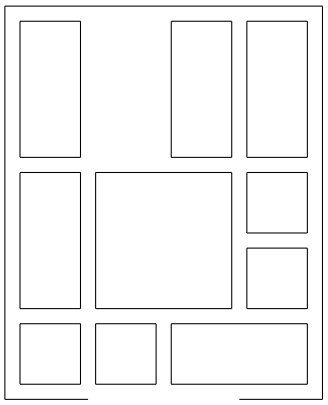
After 94 steps



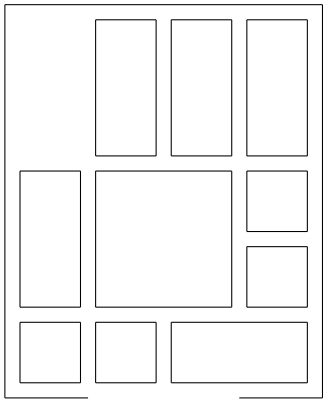
After 95 steps



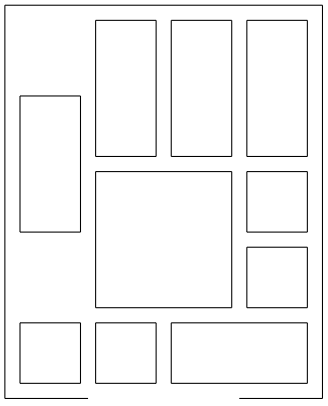
After 96 steps



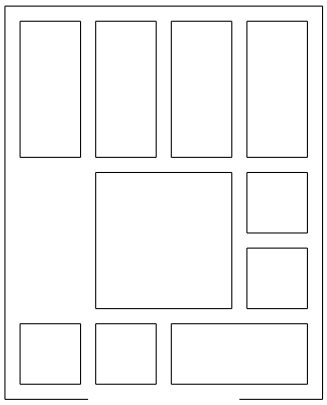
After 97 steps



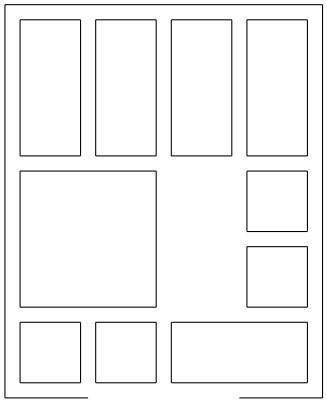
After 98 steps



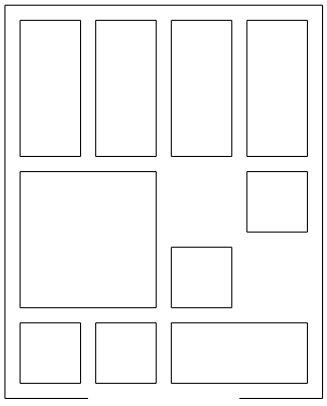
After 99 steps



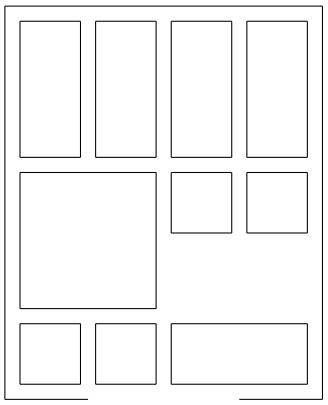
After 100 steps



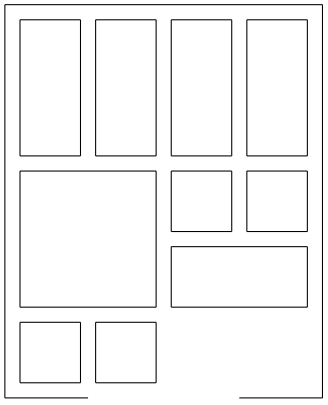
After 101 steps



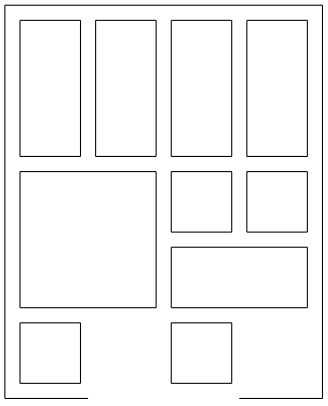
After 102 steps



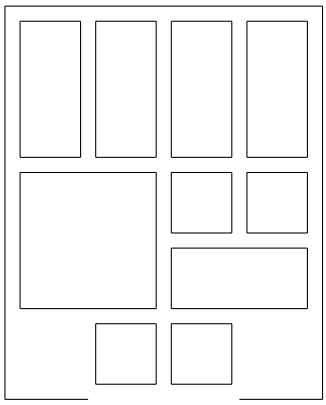
After 103 steps



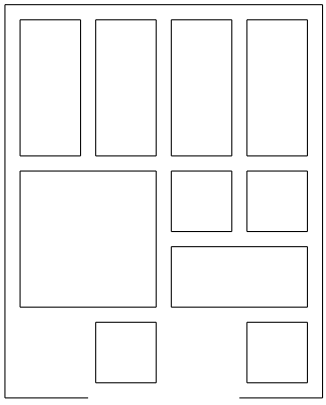
After 104 steps



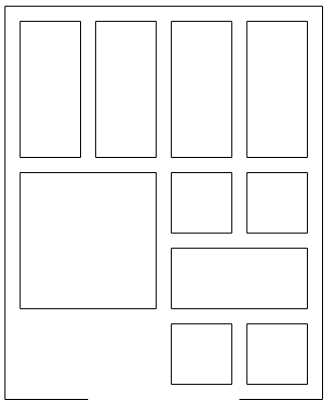
After 105 steps



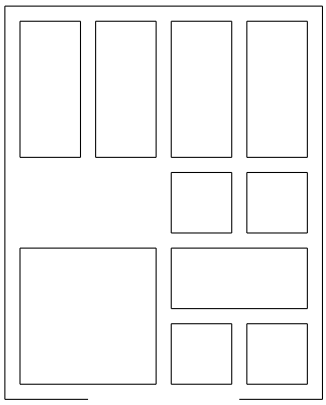
After 106 steps



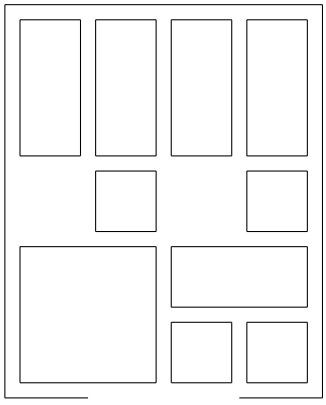
After 107 steps



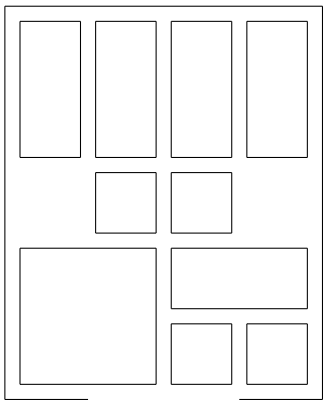
After 108 steps



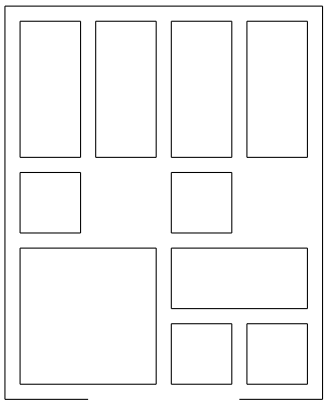
After 109 steps



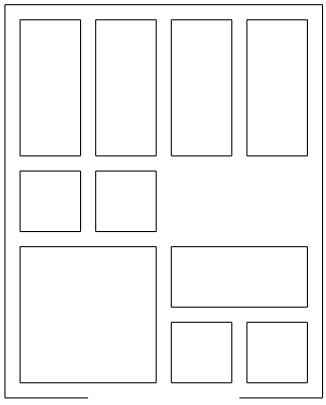
After 110 steps



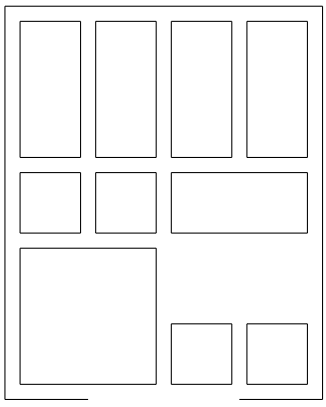
After 111 steps



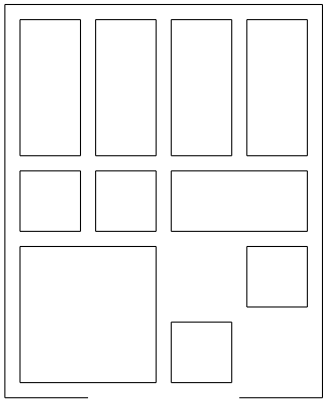
After 112 steps



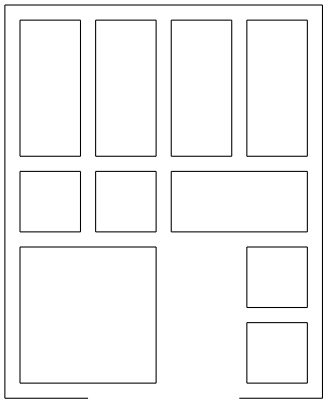
After 113 steps



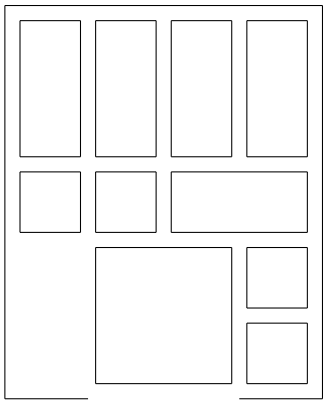
After 114 steps



After 115 steps



After 116 steps



5 Code

```
import qualified Array as A
import qualified Data.List as List
import qualified Data.Map as Map
import qualified Data.Maybe as Maybe
import qualified Data.Set as Set

-- The contents of a square on the game board.
data Square = Empty
            | Single
            | TwoTop | TwoBottom
            | TwoLeft | TwoRight
            | FourTL | FourTR | FourBL | FourBR
              deriving (Eq, Ord, Show)

-- Spaces occupied by a game piece, given its upper-left corner,
-- relative to its upper-left corner.
whole Single = [(0, 0)]
whole TwoTop = [(0, 0), (0, 1)]
whole TwoLeft = [(0, 0), (1, 0)]
whole FourTL = [(0, 0), (1, 0), (0, 1), (1, 1)]
whole _ = []

-- Vector addition.
add (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)

-- Initial game board positions.
mkStartPos :: [Square] → A.Array (Int, Int) Square
mkStartPos state = A.array ((0, 0), (3, 4)) $ zip (foldr (++) [] rows) state
    where rows = [[(x, y) | x ∈ [0..3]] | y ∈ [0..4]]
startPos = mkStartPos [TwoTop, FourTL, FourTR, TwoTop,
                       TwoBottom, FourBL, FourBR, TwoBottom,
                       TwoTop, TwoLeft, TwoRight, TwoTop,
                       TwoBottom, Single, Single, TwoBottom,
                       Single, Empty, Empty, Single]

-- Is this a winning position?
isWin p = (p A.! (1, 3)) ≡ FourTL

-- Is the given position on the game board, and empty?
isEmpty p xy = (get p xy) ≡ Just Empty
    where get p xy = if A.inRange (A.bounds p) xy
                     then (Just $ p A.! xy) else Nothing
```

```
-- Try to move the piece at pos in direction dir. Return Just the new
-- position, or Nothing if the move isn't legal.
moveSquare p pos dir =
    let poses = map (add pos) (whole $ p A.! pos)
        poses2 = map (add dir) poses
        new = poses2 List.\\ poses
        old = poses List.\\ poses2
    in if new ≠ [] ∧ all (isEmpty p) new
          then Just $ p A.// ([(add pos dir, p A.! pos) | pos ∈ poses]
                              ++ [(pos, Empty) | pos ∈ old])
          else Nothing

-- Given a list of positions, return the list of positions that are
-- reachable from them in exactly one move.
allMoves ps = Set.fromList $ concatMap allMoves1 ps
    where allMoves1 p =
              Maybe.catMaybes[moveSquare p pos dir |
                              pos ∈ A.indices p,
                              dir ∈ [(0, -1), (0, 1), (-1, 0), (1, 0)]]

-- Add position p to the visited map with the given number of steps.
-- Return the updated map and Just p if p wasn't previously visited,
-- Nothing otherwise.
addPosition :: (Ord k, Ord a) => (Map.Map k a) → a → k → (Map.Map k a, Maybe k)
addPosition visited steps p = (visited', q)
    where (old_steps, visited') = Map.insertLookupWithKey f p steps visited
          f _ old new = min old new
          q = case old_steps of
                Nothing → Just p
                Just _ → Nothing

addPositions :: (Ord k, Ord a) => (Map.Map k a) → a → [k] → (Map.Map k a, [k])
addPositions visited steps [] = (visited, [])
addPositions visited steps (p:ps) = (visited'', qs)
    where qs = case q of Just p' → p':ps'
                         Nothing → ps'
          (visited', ps') = addPositions visited steps ps
          (visited'', q) = addPosition visited' steps p

-- Given the map of visited positions, the current step, and the list
-- of current positions, return an updated map and list of current
-- positions.
newPositions visited steps old_pos =
```

```
    addPositions visited steps (Set.toList $ allMoves old_pos)

-- A list of all reachable positions: at list index k is a list of all
-- positions reachable in exactly k steps.
listPositions start = go (Map.singleton start 0) 1 [start]
    where go visited steps pos = pos : go visited' (steps + 1) pos'
              where (visited', pos') = newPositions visited steps pos

-- Given a list of reachable positions (such as produced by
-- listPositions), return the number of steps to one of the first
-- winning positions, and the position itself.
firstWin posList = maybe Nothing (λ(i, ps) → Just (i, filter isWin ps))
                   $ List.find winner $ zip [0..] posList
    where winner (i, ps) = any isWin ps

-- Given a position p and a list of candidate positions, return one of
-- the candidates that is also a legal move.
backtrack p candidates = head $ Set.toList (Set.intersection (allMoves [p])
                                                (Set.fromList candidates))

backtrackAll p [] = []
backtrackAll p (c:cs) = p' : backtrackAll p' cs
    where p' = backtrack p c

-- Given a starting position, return a list of positions that goes
-- from the starting position to one of the closest winning positions,
-- one move at a time.
winSequence start = let posList = listPositions start
                        Just (winStep, winPositions) = firstWin posList
                        winPos = head winPositions
                        revPosList = reverse $ take winStep posList
                        revWinSeq = winPos : backtrackAll winPos revPosList
                    in reverse revWinSeq

-- Return a list of strings. Each string is one line in a LaTeX
-- fragment that draws the given position.
drawPos p steps = ["\\subsection*{After " ++ (show steps) ++ " steps}",
                   "\\begin{center}",
                   "\\setlength{\\unitlength}{1cm}",
                   "\\begin{picture}(4.5,5.5)(-0.25,-0.25)"]
                  ++ border
                  ++ concat (map (λ(xy, square) → draw xy square) (A.assocs p))
                  ++ ["\\end{picture}",
                      "\\end{center}"]
```

```
 where hline x0 x1 y = "\\put(" ++ (show $ min x0 x1) ++ "," ++ (show y)
                       ++ "){\\line(1,0){" ++ (show $ abs (x0 - x1)) ++ "}}"
       vline x y0 y1 = "\\put(" ++ (show x) ++ "," ++ (show $ min y0 y1)
                       ++ "){\\line(0,1){" ++ (show $ abs (y0 - y1)) ++ "}}"
       border = [hline (-0.1) 1 (-0.1), hline 3 4.1 (-0.1),
                 hline (-0.1) 4.1 5.1,
                 vline (-0.1) (-0.1) 5.1, vline 4.1 (-0.1) 5.1]
       rect (x0, y0) (x1, y1) = [hline x0 x1 y0, hline x0 x1 y1,
                                 vline x0 y0 y1, vline x1 y0 y1]
       shrinkRect (x0, y0) (x1, y1) = rect (x0' + s, y0' + s) (x1' - s, y1' - s)
           where x0' = min x0 x1
                 x1' = max x0 x1
                 y0' = min y0 y1
                 y1' = max y0 y1
                 s = 0.1
       coordTrans (x, y) = (fromInteger $ toInteger x,
                            fromInteger $ toInteger $ 5 - y) :: (Double, Double)
       pRect xy0 xy1 = shrinkRect (coordTrans xy0) (coordTrans xy1)
       draw (x, y) Single = pRect (x, y) (x + 1, y + 1)
       draw (x, y) TwoTop = pRect (x, y) (x + 1, y + 2)
       draw (x, y) TwoLeft = pRect (x, y) (x + 2, y + 1)
       draw (x, y) FourTL = pRect (x, y) (x + 2, y + 2)
       draw _ _ = []

main = mapM_ (λ(i, p) → putStrLn (concat $ [c ++ "\n" | c ∈ (drawPos p i)]))
       $ zip [0..] $ winSequence startPos
```