Backtracking Algorithms in MCPL using Bit Patterns and Recursion

by

Martin Richards

`mr@uk.ac.cam.cl`

`http://www.cl.cam.ac.uk/users/mr/`

Computer Laboratory
University of Cambridge
February 23, 2009

Abstract

This paper presents example programs, implemented in MCPL, that use bit pattern techniques and recursion for the efficient solution of various tree search problems.

Keywords

Backtracking, recursion, bit-patterns, MCPL, queens, solitaire, pentominoes, nonograms, boolean satisfiability.

Contents

1 Introduction

This report has been written for two reasons. Firstly, to explore various efficient algorithms for solving a variety of backtracking problems using recursion and bit pattern techniques, and, secondly, to demonstrate the effectiveness of MCPL[Ric97] for applications of this sort. MCPL is designed as a successor to BCPL[RWS80]. Like BCPL, it is a simple typeless language, but incorporates features from more modern languages, particularly ML[Pau91], C, and Prolog[CM81].

An implementation of MCPL together with all the programs described in this report are freely available and can be obtained via my World Wide Web Home Page[Ric]. Although the implementation is still under development and somewhat incomplete, it is capable of running all these programs. A manual for MCPL is also available via the same home page.

It is hoped that the MCPL notation is sufficiently comprehensible without explanation, however a brief summary of its syntax has been included in the appendix. For more information consult the MCPL manual.

One of the main attractions of bit pattern techniques is the efficiency of the machine instructions involved (typically, bitwise `AND`, `OR`, `XOR` and shifts), and the speed up obtained by doing 32 (or 64) simple logical operations simultaneously. Sometimes useful results can be obtained by combining conventional arithmetic operations with logical ones. There are many other useful bit pattern operations that are cheap to implement in hardware but are typically not provided by machine designers. These include simple operations such as the bitwise versions of nor (`NOR`), implies (`IMP`) and its complement (`NIMP`), as well as higher level operations such `COMPACT` to remove unwanted bits from a long bit pattern to form a shorter one, its inverse (`SPREAD`), and certain permutation operations. Bit pattern techniques are often even more useful on the 64 bit machines that are now becoming more common.

If a problem can be cast in a form involving small sets then these techniques often help. This report covers a collection of problems that serve to illustrate the bit pattern techniques I wish to present. Some of these are trivial and some less so. Most are useful as benchmark problems for programming languages that purport to be good for this kind of application. It is, indeed, interesting to compare these MCPL programs with possible ML, C, Prolog or LISP translations.

2 The Queens Problem

A well known problem is to count the number of different ways in which eight queens can be placed on an 8×8 chess board without any two of them sharing the same row, column or diagonal. It was, for instance, used as a case study in Niklaus Wirth's classic paper "Program development by stepwise refinement"[Wir71]. In none of his solutions did he use either recursion or bit pattern techniques.

The program given here performs a walk over a complete tree of valid (partial) board states, incrementing a counter whenever a complete solution is found. The root of this tree is said to be at level 0 and represents the empty board. The root has successors corresponding to the board states with one queen placed in the bottom row. These are all said to be at level 1. Each level 1 state has successors that correspond to valid board states with queens placed in the bottom two rows. In general, any valid board state at level i ($i > 0$) contain i queens in the bottom i rows and is a successor of a board state at level $i - 1$. The solutions to the 8-queens problem are the valid board states at level 8. Ignoring symmetries, all these solutions are be distinct.

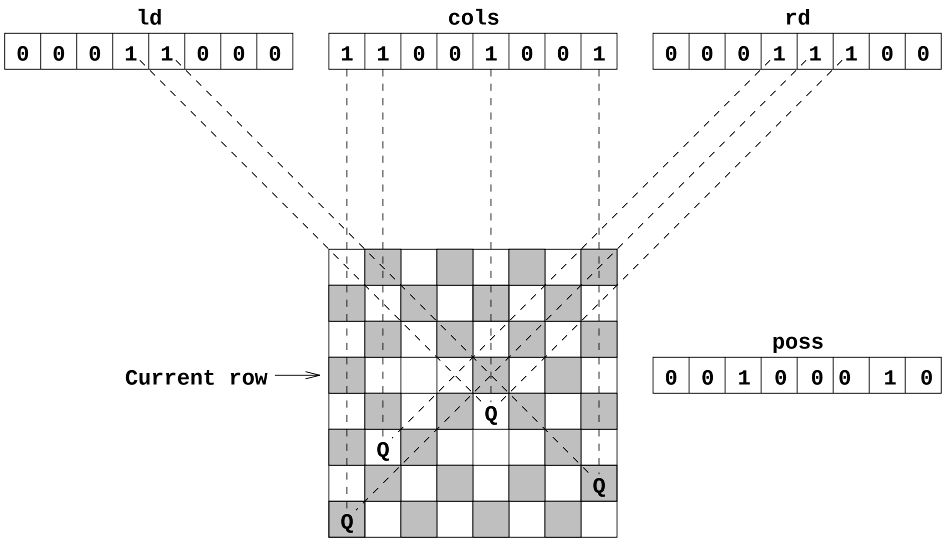


Figure 1: The Eight Queens

The walk over the tree of valid board states can be simulated without physically constructing the tree. This is done using the function `try` whose arguments `ld`, `cols` and `rd` contain sufficient information about the current board state for its successors to be explored. Figure 1 illustrated how `ld`, `cols` and `rd` are used to find where a queen can be validly placed in the current row without being attacked by any queen placed in earlier rows. `cols` is a bit pattern containing a one in for each column that is already occupied. `ld` contains a one for each position attacked along a left going diagonal, while `rd` contains diagonal attacks from the other diagonal. The expression `(ld | cols | rd)` is a bit pattern containing ones in all positions that are under attack from anywhere. When this is complemented and masked with `all`, a bit pattern is formed that gives the positions in the current row where a queen can be placed without being attacked. The variable `poss` is given this as its initial value.

```
LET poss = ~(ld | cols | rd) & all
```

The `WHILE` loop cunningly iterates over these possible placements, only executing the body of the loop as many times as needed. Notice that the expression `poss & -poss` yields the least significant one in `poss`, as is shown in the following example.

```
poss            00100010
       -poss    11011110
                --------
poss & -poss    00000010
```

The position of a valid queen placement is held in `bit` and removed from `poss` by:

```
LET bit = poss & -poss
poss -:= bit
```

and then a recursive call of `try` is made to explore the selected successor state.

```
 try( (ld|bit)<<1, cols|bit, (rd|bit)>>1 )
```

Notice that a left shift is needed for the left going diagonal attacks and a right shift for the other diagonal attacks.

When `cols=all` a complete solution has been found. This is recognised by the pattern:

```
 : ?, =all,  ? => count++
```

which increments the count of solutions.

The main function (`start`) exercises `try` to solve the n-queens problem for $1 \leq n \leq 12$. The output is as follows:

```
20> queens
There are     1 solutions to  1-queens problem
There are     0 solutions to  2-queens problem
There are     0 solutions to  3-queens problem
There are     2 solutions to  4-queens problem
There are    10 solutions to  5-queens problem
There are     4 solutions to  6-queens problem
There are    40 solutions to  7-queens problem
There are    92 solutions to  8-queens problem
There are   352 solutions to  9-queens problem
There are   724 solutions to 10-queens problem
There are  2680 solutions to 11-queens problem
There are 14200 solutions to 12-queens problem
14170>
```

Although the queens problem is commonly in texts on ML, Prolog and LISP, I have seen no solutions written in these languages that approach the efficiency of the one given here.

2.1 The queens program

```
GET "mcpl.h"

STATIC count, all

FUN try
: ?,  =all,  ? => count++

: ld, cols, rd => LET poss = ~(ld | cols | rd) & all
                  WHILE poss DO
                  { LET bit = poss & -poss
                    poss -:= bit
                    try( (ld|bit)<<1, cols|bit, (rd|bit)>>1 )
                  }

FUN start : =>
  all := 1
  FOR n = 1 TO 12 DO
  { count := 0
    try(0, 0, 0)
    writef("There are %5d solutions to %2d-queens problem\n",
                      count,             n )
    all := 2*all + 1
  }
  RETURN 0
```

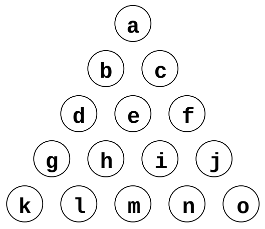
3 Solitaire Problems

Solitaire games are typically played on a board with an arrangement of drilled holes in which pegs can be inserted. If three adjacent positions are in line and have the pattern peg-peg-hole, then a move can be made. This entails moving the first peg into the hole and removing the other peg from the board. The game consists of finding a sequence of moves that will transform the initial configuration of pegs to a required final arrangement. Normally the initial configuration has only one unoccupied position and the final final arrangement is the inverse of this.

In this section, programs for both triangular and conventional solitaire are presented.

3.1 Triangular solitaire

Triangular solitaire is played on a triangular board with 15 holes, labelled as in the diagram.



The initial configurations has pegs in all holes except position `a`, and the final configuration is the inverse of this. A successful game thus consists of a sequence 13 moves. The program described here explores the game tree to find how many different successful games there are. The answer turns out to be 6816.

The tree of reachable board states is similar to the one used in the queens problem above with the root corresponding to the initial configuration and edges corresponding to moves to adjacent positions. However, a major difference is that different paths through the tree can lead to the same position. There are, after all, 6816 ways of reaching the final position. Failure to take this into account leads to a solution that is about 175 times slower.

It is therefore advisable to choose a board representation that makes it easy to determine whether the same board position has been seen before. The method used here is based on the observation that any board position can be specified by 15 boolean values that could well be represented by the least significant 15 bits of a word. Such a word can be used an integer subscript to a vector that holds information about all the 32768 different board configurations. This vector is called `scorev`.

As with the queens problem, a recursive function `try` is used to explore the tree without physically creating it. Its argument represents a board state and its result is the number of different ways of reaching the final state from the given state. Most of the work done by `try` is concerned with finding (and making) all the possible moves from its given state. If this state has been seen before then the appropriate value in `scorev` is returned. This will have been set when this state was first visited. The elements of `scorev` are initialised to the invalid score `-1`, except for the element of corresponding to the final state (`scorev!1`) which is set to `1`.

An important inner loop of the program is concerned with the search for legal moves. There are six possible moves in a direction up and to the right. These are: `d-b-a`, `g-d-b`, `k-g-d`, `h-e-c`, `l-h-e`, and `m-i-f`. There are similarly 6 possible moves in each of the other five directions, making 36 in all. Usually only a small fraction of these are possible from a given state. To test whether the move `d-b-a` can be made using our representation of the board, it is necessary to check whether bits 4 and 2 are set to one and that bit 1 is set to zero.

The `MANIFEST`-constants (`Pa, Pb ,..., Po` are declared to make testing these bit positions more convenient. A somewhat more efficient check for move legality can be made if the state of each board position is represented by a pair of bits, `01` for a peg and `10` for a hole. `MANIFEST`-constants (`Ha, Hb ,..., Ho` provide convenient access to the first digit of the pair.

The function to test and make moves is called `trymove`. Its definition is as follows:

```
FUN trymove
: brd, hhp, hpbits => brd&hhp -> 0,       // Can't make move
                      try(brd XOR hpbits) // Try new position
```

`brd` represents the board using bit pairs and `hhp` is a bit pattern selecting the presence of two holes and one peg. The expression `brd&hhp` yield a non zero value either a hole is found in the first two positions or a peg is found in the third position. A non zero result thus indicates that the specified move cannot be made, causing `trymove` to return zero. Otherwise, `trymove` calls `try` with the representation of the successor board state formed by complementing all 6 bits of the move triplet. This is cheaply computed by the expression `brd XOR hpbits`. Exploration of the move `d-b-a` can thus be achieved by the call:

```
        trymove(brd, Hd+Hb+Pa, Hd+Pd+Hb+Pb+Ha+Pa)
```

It yields the number of ways of reaching the final configuration from the board state `brd` by a path whose first move is `d-b-a`.

To improve the efficiency of the search still further, only moves originating from pegs that are actually on the board are considered. In the function `try`, the variable `poss` is initialised to represent the set of pegs still on the board, and this is used in a way somewhat similar to the iteration in the queens program. The definition of `try` is as follows:

```
FUN try : brd =>
  LET poss  = brd & Pbits
  LET score = scorev!poss
  IF score<0 DO // have we seen this board position before
  { score := 0  // No -- so calculate score for this position.
    WHILE poss DO { LET bit = poss & -poss
                    poss  -:= bit
                    score +:= (fnv!bit) brd
                  }
    scorev!(brd&Pbits) := score  // Remember the score
  }
  RETURN score
```

Pegs at positions `d`, `f` and `m` can potentially make four moves, while pegs at any other positions are limited to two. The function `fa` explores the possible moves of a peg at position `a`. Its definition is as follows:

```
FUN fa : pos => trymove(pos, Ha+Hb+Pd, Pa+Ha+Pb+Hb+Pd+Hd) +
                trymove(pos, Ha+Hc+Pf, Pa+Ha+Pc+Hc+Pf+Hf)
```

The functions (`fb,..., fo`) are defined similarly. These functions are stored (sparsely) in the vector `fnv` so that the expression `(fnv!bit) brd` will efficiently call the search function for the selected peg. The iteration in `try` will thus call only the required search functions and leave the sum of their results in `score`. This score is then saved in the appropriate position of `scorev` removing the need to recomputed it the next time this board state is encountered.

It turns out that only 3016 different states are visited, and of these only 370 are on solution paths. Even so, allocating a 32786 element vector to hold the scores is probably worthwhile.

It is, perhaps, interesting to note that only four one peg positions are reachable from the initial configuration. Which are they?

3.2 The triangular solitaire program

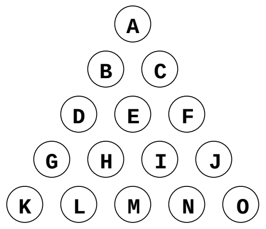
```
GET "mcpl.h"

STATIC    scorev, fnv

MANIFEST  Pbits = #x7FFF, SH = #X10000, Upb = Pbits,

// Peg bits
Pa = 1<<0,  Pb = 1<<1,  Pc = 1<<2,  Pd = 1<<3,  Pe = 1<<4,
Pf = 1<<5,  Pg = 1<<6,  Ph = 1<<7,  Pi = 1<<8,  Pj = 1<<9,
Pk = 1<<10, Pl = 1<<11, Pm = 1<<12, Pn = 1<<13, Po = 1<<14,

// Hole bits
Ha = Pa*SH, Hb = Pb*SH, Hc = Pc*SH, Hd = Pd*SH, He = Pe*SH,
Hf = Pf*SH, Hg = Pg*SH, Hh = Ph*SH, Hi = Pi*SH, Hj = Pj*SH,
Hk = Pk*SH, Hl = Pl*SH, Hm = Pm*SH, Hn = Pn*SH, Ho = Po*SH

FUN start : =>
  initvecs()

  scorev!Pa := 1       // Set the score for the final position

  LET ways = try(       Ha+
                      Pb+Pc+
                     Pd+Pe+Pf+
                   Pg+Ph+Pi+Pj+
                  Pk+Pl+Pm+Pn+Po  )

  writef("Number of solutions = %d\n", ways)
  freevecs()
  RETURN 0

FUN initvecs : => scorev, fnv := getvec Upb, getvec Upb
                  FOR i = 0 TO Upb DO scorev!i := -1

                  fnv!Pa := fa; fnv!Pb := fb; fnv!Pc := fc
                  fnv!Pd := fd; fnv!Pe := fe; fnv!Pf := fe
                  fnv!Pg := fg; fnv!Ph := fh; fnv!Pi := fi
                  fnv!Pj := fj; fnv!Pk := fk; fnv!Pl := fl
                  fnv!Pm := fm; fnv!Pn := fn; fnv!Po := fo

FUN freevecs : => freevec scorev
                  freevec fnv

FUN try : brd =>
  LET poss  = brd & Pbits
  LET score = scorev!poss
  IF score<0 DO // have we seen this board position before
  { score := 0  // No -- so calculate score for this position.
    WHILE poss DO { LET p = poss & -poss
                    poss  -:= p
                    score +:= (fnv!p) brd
                  }
    scorev!(brd&Pbits) := score  // Remember the score
  }
  RETURN score
```

```
FUN trymove
: brd, hhp, hpbits => brd&hhp -> 0,       // Can't make move
                      try(brd XOR hpbits) // Try new position

FUN fa : brd => trymove(brd, Ha+Hb+Pd, Pa+Ha+Pb+Hb+Pd+Hd) +
                trymove(brd, Ha+Hc+Pf, Pa+Ha+Pc+Hc+Pf+Hf)
FUN fb : brd => trymove(brd, Hb+Hd+Pg, Pb+Hb+Pd+Hd+Pg+Hg) +
                trymove(brd, Hb+He+Pi, Pb+Hb+Pe+He+Pi+Hi)
FUN fc : brd => trymove(brd, Hc+He+Ph, Pc+Hc+Pe+He+Ph+Hh) +
                trymove(brd, Hc+Hf+Pj, Pc+Hc+Pf+Hf+Pj+Hj)
FUN fd : brd => trymove(brd, Hd+Hb+Pa, Pd+Hd+Pb+Hb+Pa+Ha) +
                trymove(brd, Hd+He+Pf, Pd+Hd+Pe+He+Pf+Hf) +
                trymove(brd, Hd+Hg+Pk, Pd+Hd+Pg+Hg+Pk+Hk) +
                trymove(brd, Hd+Hh+Pm, Pd+Hd+Ph+Hh+Pm+Hm)
FUN fe : brd => trymove(brd, He+Hh+Pl, Pe+He+Ph+Hh+Pl+Hl) +
                trymove(brd, He+Hi+Pn, Pe+He+Pi+Hi+Pn+Hn)
FUN ff : brd => trymove(brd, Hf+Hc+Pa, Pf+Hf+Pc+Hc+Pa+Ha) +
                trymove(brd, Hf+He+Pd, Pf+Hf+Pe+He+Pd+Hd) +
                trymove(brd, Hf+Hi+Pm, Pf+Hf+Pi+Hi+Pm+Hm) +
                trymove(brd, Hf+Hj+Po, Pf+Hf+Pj+Hj+Po+Ho)
FUN fg : brd => trymove(brd, Hg+Hd+Pb, Pg+Hg+Pd+Hd+Pb+Hb) +
                trymove(brd, Hg+Hh+Pi, Pg+Hg+Ph+Hh+Pi+Hi)
FUN fh : brd => trymove(brd, Hh+He+Pc, Ph+Hh+Pe+He+Pc+Hc) +
                trymove(brd, Hh+Hi+Pj, Ph+Hh+Pi+Hi+Pj+Hj)
FUN fi : brd => trymove(brd, Hi+He+Pb, Pi+Hi+Pe+He+Pb+Hb) +
                trymove(brd, Hi+Hh+Pg, Pi+Hi+Ph+Hh+Pg+Hg)
FUN fj : brd => trymove(brd, Hj+Hf+Pc, Pj+Hj+Pf+Hf+Pc+Hc) +
                trymove(brd, Hj+Hi+Ph, Pj+Hj+Pi+Hi+Ph+Hh)
FUN fk : brd => trymove(brd, Hk+Hg+Pd, Pk+Hk+Pg+Hg+Pd+Hd) +
                trymove(brd, Hk+Hl+Pm, Pk+Hk+Pl+Hl+Pm+Hm)
FUN fl : brd => trymove(brd, Hl+Hh+Pe, Pl+Hl+Ph+Hh+Pe+He) +
                trymove(brd, Hl+Hm+Pn, Pl+Hl+Pm+Hm+Pn+Hn)
FUN fm : brd => trymove(brd, Hm+Hh+Pd, Pm+Hm+Ph+Hh+Pd+Hd) +
                trymove(brd, Hm+Hi+Pf, Pm+Hm+Pi+Hi+Pf+Hf) +
                trymove(brd, Hm+Hl+Pk, Pm+Hm+Pl+Hl+Pk+Hk) +
                trymove(brd, Hm+Hn+Po, Pm+Hm+Pn+Hn+Po+Ho)
FUN fn : brd => trymove(brd, Hn+Hi+Pe, Pn+Hn+Pi+Hi+Pe+He) +
                trymove(brd, Hn+Hm+Pl, Pn+Hn+Pm+Hm+Pl+Hl)
FUN fo : brd => trymove(brd, Ho+Hj+Pf, Po+Ho+Pj+Hj+Pf+Hf) +
                trymove(brd, Ho+Hn+Pm, Po+Ho+Pn+Hn+Pm+Hm)
```

3.3 A more efficent algorithm for triangular solitaire

This second implementation is based on ideas suggested by Ken Moody [Moo82] and Phil Hazel [Haz82]. It takes advantage of two symmetries that occur in triangular solitaire. One is the left to right symmetry of the board and the other is the forward-backward symmetry based on the observation that the game played backwards from the final position has a lattice of moves that are isomorphic with the original game, and thus only board positions up to the halfway point need be processed.

The peg positions are represented by bit patterns given by the `MANIFEST` constants `A` to `O` with the board having the following layout:



The bit patterns are chosen so that reflecting the board represented by `pos` about the line `A-E-M` is cheaply computed by the expression:

```
        (pos<<1 | pos>>1) & All
```

which essentially swaps adjacent bits. The peg positions on the line of symmetry are represented by pairs of adjacent ones so that the swap operation leaves them unchanged. The inverse board position of `pos`, where pegs are replaced by holes and vice-versa, is cheaply computed by : `pos XOR All`.

Information about board positions is stored as entries in a hash table (`hashtab`) that are built up by means of a breadth first scan. Entries in the hash table have the form: `[chain, pos, k, next]` where `chain` links entries with the same value and `next` links together positions with the same number of pegs on the board, and `pos` represents the board position.

If `pos` represents a symmetric board position (σ, say) then `k` $= N(\sigma)$ – the number of ways of reaching σ from the initial position. If `pos` represents an asymmentric position (α, say) then the entry also holds information about the reflection of α (denoted by $\overline{\alpha}$). For such asymmetric positions, `k` $= N(\alpha) + N(\overline{\alpha})$. Note that the symmetry of the game implies that $N(\alpha) = N(\overline{\alpha})$. Using the same entry for asymmetric pairs reduces the number of table entries by very nearly a factor of two.

The list of board positions reachable after n moves is formed in `poslist` by a call `scanlist(p, addpos)` where `p` is the list of positions reachable after

$n-1$ moves and `addpos` is a function to process each successor position found. For each position (π, say) in `p`, `scanlist` tries all possible moves to find the reachable successors. For each possible move ($\pi \rightarrow \pi'$), `scanlist` calls `addpos(`π'`, k)` to make (or find) the hash table entry for π', and increment its k-value by `k`, where `k` is the k-value associated with π.

Since there is only one hash table entry for each pair of asymetric positions, we need to check that the correct contribution is made to the k-value in all cases.

- π and π' are both symmetric positions.

  The contribution is `k` $= N(\pi)$, which is correct. Note, however, this case never arises with the current definition of triagular solitaire.

- π is symmetric but π' is not.

  The contribution is `k` $= N(\pi)$, but `scanlist` will also find the successor $\overline{\pi'}$ which will cause a second contribution of $N(\pi)$ to be made, as required.

- π is asymmetric and π' is symmetric.

  The contribution is `k` $= N(\pi) + N(\overline{\pi})$, which is correct since it take account of both moves $\pi \longrightarrow \pi'$ and $\overline{\pi} \longrightarrow \pi'$. Since there is only one hash table entry for the position pair $(\pi, \overline{\pi})$, `scanlist` makes no other call of `addpos` relating to this pair.

- π and π' are both asymmetric positions.

  The contribution is `k` $= N(\pi) + N(\overline{\pi})$, which is the required contribution for the pair $(\pi', \overline{\pi'})$, taking into account both moves $\pi \longrightarrow \pi'$ and $\overline{\pi} \longrightarrow \overline{\pi'}$. Since there is only one hash table entry for the position pair $(\pi, \overline{\pi})$, `scanlist` makes no other call of `addpos` relating to this pair.

After calling `scanlist(p, addpos)` for the sixth time, `poslist` is a list containing 4 symmetric positions and 268 asymmetric pairs. All these positions contain 8 pegs and 7 holes. A final call of `scanlist` now explores all moves possible from these positions to positions with 7 pegs and 8 holes. Suppose `scanlist` finds a move $\pi \longrightarrow \pi'$, then, if there is a successful game using this move, by the forward-backward symmetry the inverse of π' will be in `poslist`. The number of ways of reaching the final position from the initial position, using this move, is a simple function of `k`$\times$`k'`, where `k` and `k'` are the $\widehat{\text{k}}$-values associated with π and π'. The final result is the sum of contributions made for each such move. The result is accumulated in `ways` by the call `scanlist(poslist, addways)`. This call will effectively call `addways(`π'`, k)` where `k` is the k-value associated with π

for all moves $\pi \rightarrow \pi'$ that can currently be made. As before, there are four cases to consider:

- π and π' are both symmetric positions.

  The required contribution is $N(\pi) \times N(\pi') =$ `k`$\times$`k'`. As before this case can never arise with the current definition of triagular solitaire.

- π is symmetric but π' is not.

  The required contribution is $N(\pi) \times (N(\pi') + N(\overline{\pi'})) =$ `k`$\times$`k'`. Unfortunately, `scanlist` will call `addways` for the other successor $\overline{\pi'}$ and so in this case `addways` should use `k`$\times$`k'/2` as the contribution each time. Note that `k`$\times$`k'` is an even number so that the division by two is exact.

- π is asymmetric and π' is symmetric.

  The required contribution is $(N(\pi) + N(\overline{\pi})) \times N(\pi') =$ `k`$\times$`k'`. This takes account of both moves $\pi \rightarrow \pi'$ and $\overline{\pi} \rightarrow \pi'$. Since there is only one hash table entry for the position pair $(\pi, \overline{\pi})$, `scanlist` makes no other call of `addways` relating to this pair.

- π and π' are both asymmetric positions.

  The required contribution is $N(\pi) \times N(\pi') + N(\overline{\pi}) \times N(\overline{\pi'})$ which equals `k`$\times$`k'`/2. Since there is only one hash table entry for the position pair $(\pi, \overline{\pi})$, `scanlist` makes no other call of `addways` relating to this pair.

An encoding of `addways` that incorporates these rules is the following:

```
FUN addways : pos, k =>
  LET k1 = lookup(pos XOR All)
  IF k1 TEST symmetric pos THEN ways +:= k * k1
                             ELSE ways +:= k * k1 / 2
```

It turns out that there are no symmetric positions with 7 pegs and 8 holes on any path from the initial position to the final position, and so `addways` could have had an even simpler encoding. However this was not easy to predict.

The following program runs about 9 times faster that the earlier solution given.

3.4 The more efficient program

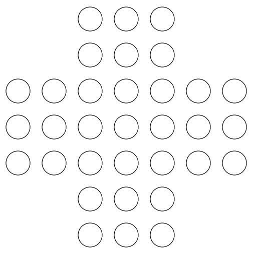
```
GET "mcpl.h"

MANIFEST

// Peg codes (cunningly chosen to ease reflection about A-E-M)

                             A=3<<18,
                        B=1,          C=2,
               D=1<<3,       E=3<<21,      F=2<<3,
          G=1<<6,      H=1<<9,      I=2<<9,       J=2<<6,
   K=1<<12,     L=1<<15,     M=3<<24,     N=2<<15,     O=2<<12,

All       =           A         +
                    B + C       +
                  D + E + F     +
                G + H + I + J   +
              K + L + M + N + O ,

Initpos   = All - A,   // Initial position

Hashtabsize = 541

STATIC
  spacev, spacep, poslist, hashtab, ways

FUN start : =>
  spacev := getvec 50000  // It uses 2012 words
  spacep := spacev
  hashtab := getvec(Hashtabsize-1)
  FOR i = 0 TO Hashtabsize-1 DO hashtab!i := 0

  poslist := 0
  addpos(Initpos, 1)

  FOR i = 1 TO 6 DO
  { LET p = poslist
    poslist := 0
    scanlist(p, addpos)
  }

  ways := 0
  scanlist(poslist, addways)
  writef("Number of solutions = %d\n", ways)

  freevec hashtab
  freevec spacev
  RETURN 0
```

```
FUN scanlist : p, f =>
  WHILE p MATCH p : [chain, pos, k, next] =>
  { UNLESS pos&A DO { IF pos&(B+D)=(B+D) DO f(pos XOR (D+B+A), k)
                      IF pos&(F+C)=(F+C) DO f(pos XOR (F+C+A), k)
                    }
    UNLESS pos&B DO { IF pos&(G+D)=(G+D) DO f(pos XOR (G+D+B), k)
                      IF pos&(I+E)=(I+E) DO f(pos XOR (I+E+B), k)
                    }
    UNLESS pos&C DO { IF pos&(H+E)=(H+E) DO f(pos XOR (H+E+C), k)
                      IF pos&(J+F)=(J+F) DO f(pos XOR (J+F+C), k)
                    }
    UNLESS pos&D DO { IF pos&(F+E)=(F+E) DO f(pos XOR (F+E+D), k)
                      IF pos&(A+B)=(A+B) DO f(pos XOR (A+B+D), k)
                      IF pos&(K+G)=(K+G) DO f(pos XOR (K+G+D), k)
                      IF pos&(M+H)=(M+H) DO f(pos XOR (M+H+D), k)
                    }
    UNLESS pos&E DO { IF pos&(L+H)=(L+H) DO f(pos XOR (L+H+E), k)
                      IF pos&(N+I)=(N+I) DO f(pos XOR (N+I+E), k)
                    }
    UNLESS pos&F DO { IF pos&(A+C)=(A+C) DO f(pos XOR (A+C+F), k)
                      IF pos&(D+E)=(D+E) DO f(pos XOR (D+E+F), k)
                      IF pos&(M+I)=(M+I) DO f(pos XOR (M+I+F), k)
                      IF pos&(O+J)=(O+J) DO f(pos XOR (O+J+F), k)
                    }
    UNLESS pos&G DO { IF pos&(I+H)=(I+H) DO f(pos XOR (I+H+G), k)
                      IF pos&(B+D)=(B+D) DO f(pos XOR (B+D+G), k)
                    }
    UNLESS pos&H DO { IF pos&(J+I)=(J+I) DO f(pos XOR (J+I+H), k)
                      IF pos&(C+E)=(C+E) DO f(pos XOR (C+E+H), k)
                    }
    UNLESS pos&I DO { IF pos&(B+E)=(B+E) DO f(pos XOR (B+E+I), k)
                      IF pos&(G+H)=(G+H) DO f(pos XOR (G+H+I), k)
                    }
    UNLESS pos&J DO { IF pos&(C+F)=(C+F) DO f(pos XOR (C+F+J), k)
                      IF pos&(H+I)=(H+I) DO f(pos XOR (H+I+J), k)
                    }
    UNLESS pos&K DO { IF pos&(M+L)=(M+L) DO f(pos XOR (M+L+K), k)
                      IF pos&(D+G)=(D+G) DO f(pos XOR (D+G+K), k)
                    }
    UNLESS pos&L DO { IF pos&(N+M)=(N+M) DO f(pos XOR (N+M+L), k)
                      IF pos&(E+H)=(E+H) DO f(pos XOR (E+H+L), k)
                    }
    UNLESS pos&M DO { IF pos&(O+N)=(O+N) DO f(pos XOR (O+N+M), k)
                      IF pos&(F+I)=(F+I) DO f(pos XOR (F+I+M), k)
                      IF pos&(D+H)=(D+H) DO f(pos XOR (D+H+M), k)
                      IF pos&(K+L)=(K+L) DO f(pos XOR (K+L+M), k)
                    }
    UNLESS pos&N DO { IF pos&(E+I)=(E+I) DO f(pos XOR (E+I+N), k)
                      IF pos&(L+M)=(L+M) DO f(pos XOR (L+M+N), k)
                    }
    UNLESS pos&O DO { IF pos&(F+J)=(F+J) DO f(pos XOR (F+J+O), k)
                      IF pos&(M+N)=(M+N) DO f(pos XOR (M+N+O), k)
                    }
    p := next
  }
```

```
FUN symmetric : pos => pos = (pos<<1 | pos>>1) & All

FUN minreflect : pos => LET rpos = (pos<<1 | pos>>1) & All
                        IF pos<=rpos RETURN pos
                        RETURN rpos

FUN addpos : pos, k =>
  pos := minreflect pos
  LET hashval = pos MOD Hashtabsize
  LET p = hashtab!hashval
  WHILE p MATCH p : [chain, =pos, n, ?] => n +:= k; RETURN
                  : [chain,    ?, ?, ?] => p := chain
                  .
  p := mk4(hashtab!hashval, pos, k, poslist)
  hashtab!hashval := p
  poslist         := p

FUN lookup : pos =>
  pos := minreflect pos
  LET hashval = pos MOD Hashtabsize
  LET p = hashtab!hashval
  WHILE p MATCH p : [    ?, =pos, n, ?] => RETURN n
                  : [chain,    ?, ?, ?] => p := chain
                  .
  RETURN 0

FUN addways : pos, k =>
  LET k1 = lookup(pos XOR All)
  IF k1 TEST symmetric pos THEN ways +:= k * k1
                           ELSE ways +:= k * k1 / 2

FUN mk4 : a, b, c, d => LET res = spacep
                        !spacep+++ := a
                        !spacep+++ := b
                        !spacep+++ := c
                        !spacep+++ := d
                        RETURN res
```

3.5 Conventional solitaire

Conventional solitaire uses a board of the following shape:



This board has 33 peg positions which is unfortunate for bit pattern algorithms designed to run on a 32 bit implementation of MCPL. The size of the game is such that it is not feasible to count the number of solutions, so the program given here just finds one solution. It uses a vector (`board`) to represent a 9×9 area that contains the board surrounded by a border that is at least one cell wide. This is declared at the beginning of `start` with the aid of the constants `X`, `P` and `H` to represent border, peg and hole positions, respectively. The function `try` searches the move tree until a solution is found, when it raises the exception `Found` that is handled in `start`.

The strategy used by `try` is to find each peg on the board and explore its possible moves. At any stage the vector `movev` holds a packed representation of the current sequence of moves. These are output when the exception `Found` is raised.

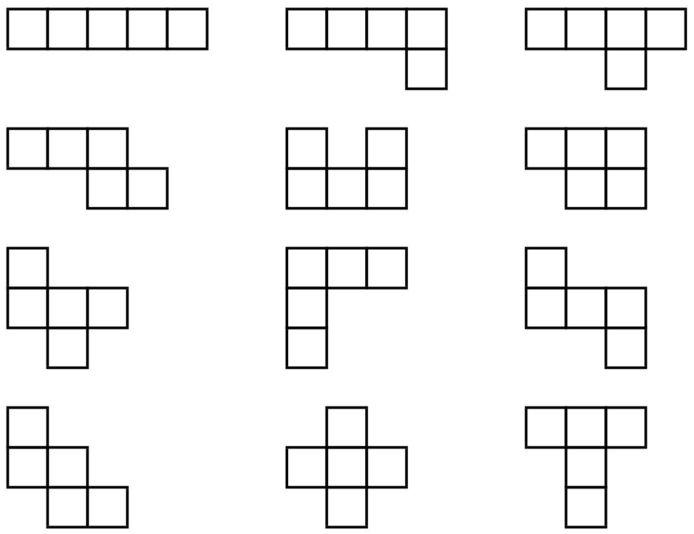
The argument of `try` is the number of pegs still to be removed. When this reaches zero, a solution has been found if the remaining peg is in the centre. If there are still pegs to be removed, moves in each of the four directions are tried for each remaining peg. The board positions used in the move are held in `p`, `p1` and `p2`. If position `p1` holds a peg and position `p2` is unoccupied then the move can be made. This move is saved in `movev`, the board updated appropriately, and a recursive call of `try` used to explores this new board state. On return the previous board state is restored. The time taken to find a solution turns out to be very dependent on the order in which the directions are tried.

3.6 The conventional solitaire program

```
GET "mcpl.h"

MANIFEST
  X, P, H,                            // For boarder, Peg and Hole.
  Centre=4*9+4, Last=9*9-1,
  North=-9, South=9, East=1, West=-1, // Directions
  Found=100                           // An exception

STATIC
  board,
  movev = VEC 31,
  dir   = [ East, South, West, North ]

FUN start : =>

  board := [ X,X,X,X,X,X,X,X,X,
             X,X,X,P,P,P,X,X,X,
             X,X,X,P,P,P,X,X,X,
             X,P,P,P,P,P,P,P,X,
             X,P,P,P,H,P,P,P,X,
             X,P,P,P,P,P,P,P,X,
             X,X,X,P,P,P,X,X,X,
             X,X,X,P,P,P,X,X,X,
             X,X,X,X,X,X,X,X,X
           ]

  try 31                             // There are 31 pegs to remove
  HANDLE : Found => FOR i = 31 TO 1 BY -1 DO
                    { LET m = movev!i
                      writef("Move peg from %2d over %2d to %2d\n",
                              (m>>16)&255, (m>>8)&255, m&255)
                    }
                    RETURN 0
           .
  writef "Not found\n"
  RETURN 0

FUN pack : a, b, c => a<<16 | b<<8 | c

FUN try
: 0 => IF board!Centre= P RAISE Found

: m => FOR p = 0 TO Last IF board!p= P DO  // Find a peg
         FOR k = 0 TO 3 DO
           { LET d = dir!k  // Try a direction
             LET p1 = p  + d
             LET p2 = p1 + d
             IF board!p1= P AND board!p2= H DO // Is move possible?
             { movev!m := pack(p, p1, p2)  // It is, so try making it
               board!p, board!p1, board!p2 :=  H,  H,  P
               try(m-1)                    // Explore new position
               board!p, board!p1, board!p2 :=  P,  P,  H
             }
           }
```

4 The Pentominoes Problem

There are twelve pieces, called pentominoes, that can be formed in two dimensions by joining five unit squares together along their edges. A specimen of each piece is pictured below.



A two dimensional rectangular board six unit wide and ten units long can be entirely covered by these 12 pentominoes without any piece overlapping with any other. This section presents four programs to compute the number of ways in which the pieces can (by rotations and reflections) be fitted on the board.

The problem can be solved by exploring the tree of board states that can be reached by placing the pieces one at time. To ensure that the tree only holds distinct states, each piece placement covers the top leftmost unoccupied square (the *handle*). All four programs discussed here use this strategy.

4.1 Pento

Each piece can be rotated and reflected to give potentially eight variants. It saves time if all the variant forms are precalculated before the search begins.

This first program calculates the variants of each piece for each handle square. The result is a vector `pv` for which the expression `pv!piece!pos` is the list of ways the specified piece can be placed on the board covering handle square `pos`. In forming this list all reflections and rotations of the piece are considered in addition to possible collision with the edge of the board or squares to the left or above the handle square.

This structure is initialised by calling `init` for each variant of each piece. The encoding of `init` is straightforward but, of course, depends on the representation chosen for the board.

The board is essentially represented by a bit pattern of length 60, with occupied positions specified by ones. But, since all positions earlier than the handle are occupied and all positions more than 25 squares ahead are unoccupied, it is possible to represent the board by a 25 bit window and an integer giving the window position. This greatly improves the efficiency on some machines.

The tree of board states is, as usual, searched by a function called `try`. Its first argument (`n`) indicates how many pieces still need to be placed, and the second and third arguments (`p` and `board`) give the current window position and window bits.

Variants for a particular piece and handle square can be a list of 32 bit words, one per variant. With this representation the inner loop of the tree search can be encoded as follows:

```
{ MATCH list : [next, bits] =>
  UNLESS bits & board DO
  { pos!n, bv!n, iv!n := p, bits, id
    try(n-1, p, bits+board)
  }
  list := next
} REPEATWHILE list
```

Here, `list` is a non empty list of possible placements covering the handle square at position `p` on the board. Within a list node, `next` and `bits` give the rest of the list and the bit pattern for this placement, respectively. The result of `bits & board` is zero if the placement is compatible with the current board state, in which case the new state is explored by the recursive call of `try`.

Information about successful placements are saved in the vectors `pos`, `bv`, `iv` so that solutions can be output when found.

4.2 The pento program

```
GET "mcpl.h"

GLOBAL count, spacev, spacep, spacet, pv, idv, pos, bv, iv

FUN setup : =>
   // Initialise the data structure representing
   // rotations, reflections and translations of the pieces.
   spacev := getvec 5000
   spacet := @ spacev!5000
   spacep := spacet

   pv  := getvec 11
   idv := getvec 11
   pos := getvec 11
   bv  := getvec 11
   iv  := getvec 11

   FOR i = 0 TO 11 DO
   { LET v = getvec 59
     FOR p = 0 TO 59 DO v!p := 0
     pv!i, idv!i := v, 'A'+i
     pos!i, bv!i, iv!i := 0, 0, 0 // Solution info
   }

   init(0, #0000000037)  //  * * * * *   *
   init(0, #0101010101)  //              *
                         //              *
                         //              *
                         //              *

   init(1,     #020702)  //    *
                         //  * * *
                         //    *

   init(2,   #03010101)  //    *    *    * *    * *
   init(2,   #03020202)  //    *    *      *    *
   init(2,   #01010103)  //    *    *      *    *
   init(2,   #02020203)  //  * *    * *    *    *

   init(2,       #1701)  //           *    *
   init(2,       #1710)  //     * * * *    * * * *

   init(2,       #0117)  //     * * * *    * * * *
   init(2,       #1017)  //           *    *

   init(3,     #010701)  //      *   *      * * *     *
   init(3,     #040704)  //  * * *   * * *    *       *
   init(3,     #020207)  //      *   *        *     * * *
   init(3,     #070202)
   init(4,       #0703)  //    * *   * *    * * *   * * *
   init(4,       #0706)  //  * * *   * * *    * *   * *
   init(4,       #0307)
   init(4,       #0607)
   init(4,     #030301)  //    *   *     * *   * *
   init(4,     #030302)  //  * *   * *   * *   * *
   init(4,     #010303)  //  * *   * *     *   *
   init(4,     #020303)

   init(5,       #0316)  //  * * *       * * *
   init(5,       #1407)  //      * *   * *

   init(5,       #1603)  //      * *   * *
   init(5,       #0714)  //  * * *       * * *

   init(5,   #01030202)  //  *       *     *   *
   init(5,   #02030101)  //  *       *   * *   * *
   init(5,   #02020301)  //  * *   * *   *       *
   init(5,   #01010302)  //    *   *     *       *

   init(6,     #070101)  //    *   *       * * *   * * *
   init(6,     #070404)  //    *   *           *   *
   init(6,     #010107)  // * **   * * *       *   *
   init(6,     #040407)

   init(7,     #030604)  //  *           *     * *   * *
   init(7,     #060301)  //  * *       * *   * *       * *
   init(7,     #040603)  //    * *   * *     *           *
   init(7,     #010306)

   init(8,     #030103)  //  * *   * *   * * *   *   *
   init(8,     #030203)  //    *   *     *   *   * * *
   init(8,       #0507)  //  * *   * *
   init(8,       #0705)

   init(9,     #010704)  //  *           *   * *       * *
   init(9,     #040701)  //  * * *   * * *     *       *
   init(9,     #030206)  //      *   *         * *   * *
   init(9,     #060203)

   init(10,      #1702)  //      *       *       * * * *   * * * *
   init(10,      #1704)  //  * * * *   * * * *       *     *
   init(10,      #0217)
   init(10,      #0417)
   init(10,  #01030101)  //    *   *       *   *
   init(10,  #02030202)  //    *   *     * *   * *
   init(10,  #01010301)  //  * *   * *     *   *
   init(10,  #02020302)  //    *   *       *   *
```

```
// the comments eliminate reflectively different solutions
   init(11,    #010702)  //    *       *         *   *
// init(11,    #040702)  //  * * *   * * *   * * *   * * *
// init(11,    #020701)  //      *   *         *       *
// init(11,    #020704)
   init(11,    #030602)  //    *       *       * *   * *
// init(11,    #060302)  //  * *       * *   * *       * *
// init(11,    #020603)  //    * *   * *       *       *
// init(11,    #020306)


FUN freespace : =>
   FOR i = 0 TO 11 DO freevec(pv!i)

   freevec pv
   freevec idv
   freevec pos
   freevec bv
   freevec iv
   freevec spacev


FUN mk2 : x, y => !---spacep := y
                  !---spacep := x
                  spacep


FUN init : piece, bits =>
  LET word=bits, height=0
  WHILE word DO { word >>:= 6; height++ }

  LET pat=bits, orig=0
  UNTIL pat&1 DO { pat >>:= 1; orig++ }

  LET v = pv!piece
  FOR p = orig TO orig + 6*(10-height) BY 6 DO
  { LET q   = p
    word   := bits

    { v!q := mk2(v!q, pat)
      IF word & #4040404040 BREAK // can't move left any more
      word <<:= 1                 // move piece left one place
      q++
    } REPEAT
  }
```

```
FUN try
: <0, ?,     ? => writef("Solution %d:\n", ++count); pr()

: n,  p, board =>
  WHILE board&1 DO { p++; board >>:= 1 }
  FOR i = 0 TO n DO
  { LET pvi=pv!i, id=idv!i

    MATCH pvi!p
    : 0    => LOOP
    : list => pv!i, idv!i := pv!n, idv!n

              { MATCH list : [next, bits] =>
                  UNLESS bits & board DO
                  { pos!n, bv!n, iv!n := p, bits, id
                    try(n-1, p, bits+board)
                  }
                  list := next
              } REPEATWHILE list
    .
    pv!i, idv!i, bv!n := pvi, id, 0
  }


FUN start : =>
  setup()
  count := 0
  try(11, 0, 0)
  writef("\nNumber of solutions is %d\n", count)
  freespace()
  RETURN 0

FUN pr : =>
  LET v = VEC 59
  FOR i = 0 TO 59 DO v!i := '-'
  FOR i = 0 TO 11 DO { LET p=pos!i, bits=bv!i, id=iv!i
                       WHILE bits DO
                       { IF bits&1 DO v!p := id
                         bits >>:= 1
                         p++
                       }
                     }
  FOR row = 0 TO 9 DO
  { FOR p = 6*row+5 TO 6*row BY -1 DO writef(" %c", v!p)
    newline()
  }
  newline()
```

4.3 Pento3

This program and the following two are essentially re-implementation of the search strategy used in Fletcher [Fle65]. The method used is to search the neighbourhood of the handle square for connected unoccupied squares. If an area of 5 squares is found, it will correspond to a pentomino which can be placed there, provided it has not already been used. The neighbourhood search can be organised as a tree with 63 leaf nodes all at a depth of 5 with each leaf identifying which pentomino fits the unoccupied area found.

The overall search is controlled by the function `try`. Its first argument indicates how many pentominoes have already been placed. When this reaches 12 a solution has been found. The second argument of `try` is a pointer into a vector reprsenting the board. The first few lines of `try` are as follows:

```
FUN try

: 12, ?                              => count++
                                        pr board

:  n, [              ~=0,a1      ]  => try (n, @a1)

:  n, [           a,a1,a2,a3,a4,
        bz,by,bx, b,b1,b2,b3, ?,
         ?,cy,cx, c,c1,c2, ?, ?,
         ?, ?,dx, d,d1, ?, ?, ?,
         ?, ?, ?, e              ] p  =>
```

The first two pattern test for a solution and search for the handle square, respectively. The third pattern is matched when the second argument (`p`) points to the handle square (`a`). It give names to all the squares that could be covered by a pentomino covering the handle. The `EVERY`-statement tries all possible pentomino placements in turn. The code:

```
EVERY
(   0,  0,  0,  0,  0 )

: =a1,=a2,=a3,=a4,=p2 => a,a1,a2,a3,a4,p2 ALL:= n; try (n, @a1)
                         a,a1,a2,a3,a4,p2 ALL:= 0
: =a1,=a2,=a3, =b,=p3 => a,a1,a2,a3, b,p3 ALL:= n; try (n, @a1)
                         a,a1,a2,a3, b,p3 ALL:= 0
...
```

tests a placement of the long straight piece (`p2`) can be placed, and then one of the L-shaped pieces (`p3`). In the full program all 63 patterns are given. An optimising MCPL compiler would compile these patterns into an efficient binary tree of tests that does not recompute conditions that have already been evaluated.

Notice that, in the definition of `start`, the initial board state is given in a readable form.

The program Pento4 is essentially the same algorithm as Pento3 but with an explicit encoding of the binary search tree.

4.4 The Pento3 program

```
GET "mcpl.h"

STATIC
  board, count=0,
  p1=0, p2=0, p3=0, p4=0, p5=0, p6=0,
  p7=0, p8=0, p9=0, pA=0, pB=0, pC=0

FUN try

: 12, ?                             => count++
                                       pr board

:  n, [              ~=0,a1      ]  => try (n, @a1)

:  n, [           a,a1,a2,a3,a4,
       bz,by,bx, b,b1,b2,b3, ?,
        ?,cy,cx, c,c1,c2, ?, ?,
        ?, ?,dx, d,d1, ?, ?, ?,
        ?, ?, ?, e               ] p  =>

  n++

  EVERY
  (   0,  0,  0,  0,  0 )

  : =a1,=a2,=a3,=a4,=p2 => a,a1,a2,a3,a4,p2 ALL:= n; try (n, @a1)
                           a,a1,a2,a3,a4,p2 ALL:= 0
  : =a1,=a2,=a3, =b,=p3 => a,a1,a2,a3, b,p3 ALL:= n; try (n, @a1)
                           a,a1,a2,a3, b,p3 ALL:= 0
  : =a1,=a2,=a3,=b1,=pB => a,a1,a2,a3,b1,pB ALL:= n; try (n, @a1)
                           a,a1,a2,a3,b1,pB ALL:= 0
  : =a1,=a2,=a3,=b2,=pB => a,a1,a2,a3,b2,pB ALL:= n; try (n, @a1)
                           a,a1,a2,a3,b2,pB ALL:= 0
  : =a1,=a2,=a3,=b3,=p3 => a,a1,a2,a3,b3,p3 ALL:= n; try (n, @a1)
                           a,a1,a2,a3,b3,p3 ALL:= 0
  : =a1,=a2, =b,=bx,=p4 => a,a1,a2, b,bx,p4 ALL:= n; try (n, @a1)
                           a,a1,a2, b,bx,p4 ALL:= 0
  : =a1,=a2, =b,=b1,=p5 => a,a1,a2, b,b1,p5 ALL:= n; try (n, @a1)
                           a,a1,a2, b,b1,p5 ALL:= 0
  ...
  ... Many similar lines
  ...

  :  =b, =c,=c1,=c2,=p8 => a, b, c,c1,c2,p8 ALL:= n; try (n, @a1)
                           a, b, c,c1,c2,p8 ALL:= 0
  :  =b, =c,=c1, =d,=pB => a, b, c,c1, d,pB ALL:= n; try (n, @a1)
                           a, b, c,c1, d,pB ALL:= 0
  :  =b, =c,=c1,=d1,=p4 => a, b, c,c1,d1,p4 ALL:= n; try (n, @a1)
                           a, b, c,c1,d1,p4 ALL:= 0
  :  =b, =c, =d,=dx,=p3 => a, b, c, d,dx,p3 ALL:= n; try (n, @a1)
                           a, b, c, d,dx,p3 ALL:= 0
  :  =b, =c, =d,=d1,=p3 => a, b, c, d,d1,p3 ALL:= n; try (n, @a1)
                           a, b, c, d,d1,p3 ALL:= 0
  :  =b, =c, =d, =e,=p2 => a, b, c, d, e,p2 ALL:= n; try (n, @a1)
                           a, b, c, d, e,p2 ALL:= 0
```

```
FUN pr : =>
    writef("\nSolution number %d", count)
    FOR i = 0 TO 12*8-1 DO
    { LET n  = board!i
      LET ch = '*'
      IF 0<=n<=12 DO ch := ".ABCDEFGHIJKL"%n
      IF i MOD 8 = 0 DO newline()
      writef(" %c", ch)
    }
    newline()

FUN start : =>
    writef "Pento version 3 entered\n"

    LET x = -1
    board := [ x,x,x,x,x,x,x,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,x,x,x,x,x,x,x  ]

    try(0, board)

    writef("\nThe total number of solutions is %d\n", count)

    RETURN 0
```

4.5 The Pento4 program

```
GET "mcpl.h"

STATIC
  depth, p, board, count, trycount,
  p1,p2,p3,p4,p5,p6,p7,p8,p9,pA,pB,pC

FUN put
:[?,y], [square(=0)], [piece(=TRUE)] => square, piece := depth, FALSE
                                        TEST depth=12
                                        THEN { count++; pr() }
                                        ELSE try (@y)
                                        square, piece := 0, TRUE

:  => RETURN


FUN try

:    [             ~=0,a1      ]  => try(@a1)

: sq [           a,a1,a2,a3,a4,
       bz,by,bx, b,b1,b2,b3, ?,
        ?,cy,cx, c,c1,c2, ?, ?,
        ?, ?,dx, d,d1, ?, ?, ?,
        ?, ?, ?, e             ]  =>

  depth++

  a := depth

  IF a1=0 DO { a1 := depth
               IF a2=0 DO { a2 := depth
                            IF a3=0 DO { a3 := depth; put(sq,@a4,@p2)
                                                      put(sq, @b,@p3)
                                                      put(sq,@b1,@pB)
                                                      put(sq,@b2,@pB)
                                                      put(sq,@b3,@p3)
                                         a3 := 0
                                       }
                            IF  b=0 DO { b := depth;  put(sq,@bx,@p4)
                                                      put(sq,@b1,@p5)
                                                      put(sq,@b2,@p7)
                                                      put(sq, @c,@p8)
                                         b := 0
                                       }
                            IF b1=0 DO { b1 := depth; put(sq,@b2,@p5)
                                                      put(sq,@c1,@p6)
                                         b1 := 0
                                       }
                            IF b2=0 DO { b2 := depth; put(sq,@b3,@p4)
                                                      put(sq,@c2,@p8)
                                         b2 := 0
                                       }
                            a2 := 0
                          }
```

```
             IF  b=0 DO { b := depth
                          IF bx=0 DO { bx := depth; put(sq,@by,@p4)
                                                    put(sq,@cx,@p9)
                                                    put(sq,@b1,@p5)
                                                    put(sq, @c,@p1)
                                       bx := 0
                                     }
                          IF b1=0 DO { b1 := depth; put(sq,@b2,@p5)
                                                    put(sq, @c,@p5)
                                                    put(sq,@c1,@p5)
                                       b1 := 0
                                     }
                          IF  c=0 DO { c := depth;  put(sq,@cx,@pC)
                                                    put(sq,@c1,@p7)
                                                    put(sq, @d,@p3)
                                       c := 0
                                     }
                          b := 0
                        }
             IF b1=0 DO { b1 := depth
                          IF b2=0 DO { b2 := depth; put(sq,@b3,@p4)
                                                    put(sq,@c2,@p9)
                                                 // put(sq,@c1,@p1)
                                       b2 := 0
                                     }
                          IF c1=0 DO { c1 := depth; put(sq, @c,@p7)
                                                    put(sq,@c2,@pC)
                                                    put(sq,@d1,@p3)
                                       c1 := 0
                                     }
                          b1 := 0
                         }
             a1 := 0
           }
IF  b=0 DO { b := depth
             IF bx=0 DO { bx := depth
                          IF by=0 DO { by := depth; put(sq,@bz,@p3)
                                                    put(sq,@cy,@pC)
                                                    put(sq,@b1,@pB)
                                                    put(sq, @c,@p6)
                                                    put(sq,@cx,@p1)
                                       by := 0
                                     }
                          IF cx=0 DO { cx := depth; put(sq,@cy,@p9)
                                                    put(sq, @c,@p5)
                                                    put(sq,@dx,@p4)
                                                 // put(sq,@b1,@p1)
                                       cx := 0
                                     }
                          IF b1=0 DO { b1 := depth; put(sq,@b2,@pB)
                                                    put(sq, @c,@pA)
                                                 // put(sq,@c1,@p1)
                                       b1 := 0
                                     }
                          IF  c=0 DO { c := depth;  put(sq, @d,@pB)
                                                 // put(sq,@c1,@p1)
                                       c := 0
                                     }
                          bx := 0
                        }
             IF b1=0 DO { b1 := depth
                          IF b2=0 DO { b2 := depth; put(sq,@a2,@p7)
                                                    put(sq,@b3,@p3)
                                                    put(sq, @c,@p6)
                                                    put(sq,@c2,@pC)
                                                    // put(sq,@c1,@p1)
                                       b2 := 0
                                     }
                          IF  c=0 DO { c := depth;  put(sq,@c1,@p5)
                                                    put(sq, @d,@pB)
                                                 // put(sq,@cx,@p1)
                                       c := 0
                                     }
                          IF c1=0 DO { c1 := depth; put(sq,@c2,@p9)
                                                    put(sq,@d1,@p4)
                                       c1 := 0
                                     }
                          b1 := 0
                        }
             IF  c=0 DO { c := depth
                          IF cx=0 DO { cx := depth; put(sq,@cy,@p8)
                                                    put(sq,@dx,@p4)
                                                    put(sq,@c1,@p6)
                                                    put(sq, @d,@pB)
                                       cx := 0
                                     }
                          IF c1=0 DO { c1 := depth; put(sq,@c2,@p8)
                                                    put(sq, @d,@pB)
                                                    put(sq,@d1,@p4)
                                       c1 := 0
                                     }
                          IF  d=0 DO { d := depth;  put(sq,@dx,@p3)
                                                    put(sq,@d1,@p3)
                                                    put(sq, @e,@p2)
                                       d := 0
                                     }
                          c := 0
                        }
             b := 0
           }
a := 0
depth--
```

```
FUN pr : =>
    writef("\nSolution number %d", count)
    FOR i = 0 TO 12*8-1 DO
    { LET n  = board!i
      LET ch = '*'
      IF 0<=n<=12 DO ch := ".ABCDEFGHIJKL"%n
      IF i MOD 8 = 0 DO newline()
      writef(" %c", ch)
    }
    newline()

FUN start : =>
    writef "Pento version 4 entered\n"

    LET x = -1
    board := [ x,x,x,x,x,x,x,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,0,0,0,0,0,0,x,
               x,x,x,x,x,x,x,x  ]

  // Set all pieces initially unused
  p1,p2,p3,p4,p5,p6,p7,p8,p9,pA,pB,pC ALL:= TRUE

  depth, count := 0, 0

  try board

  writef("\nThe total number of solutions is %d\n", count)

  RETURN 0
```

4.6 Pento6

This is an alternative implementation of Pento3 using bit patterns. As usual the search is done by the function `try`. The state of the board is represented using a scheme similar that used in the first pentominoes program, that is by an integer (`p`) to identify a position near the handle square and a bit pattern (`brd`) that contains occupancy information about this neighbourhood of the board. Each row of the board uses 7 bits — six for the board and one for the boundary. Manifest constants such as `A, A1, ...` identify positions near the handle square and provide a convenient means of constructing bit patterns for the various pentomino shapes. For instance, the two possible orientations of the long straight piece are represented by `A+A1+A2+A3+A4` and `A+B+C+D+E`.

The function `try` takes five arguments: `bits` representing a pentomino shape, `piece` identifies which pentomino is being tried, `p` is the position of the handle square, `brd` is the current state of the board relative to this position and `used` is a bit pattern indicating which pentominoes have already been used. The statement:

```
IF brd&bits OR used&piece RETURN
```

causes a return from `try` if the attempted placement conflicts with the edge or a previous placement, or if the pentomino has already been used. If the placement is legal the variables `brd` and `used` are updated, and a test performed to see if a complete solution has been found. If not, an new handle square if found by:

```
WHILE brd&1 DO { brd>>:=1; p++ }
```

and the new border bits inserted by: `brd |:= border!p`, where `border` is an explicitly declared vector giving the border patterns for each possible handle square. What follows is a sequence of 63 calls of `try` to test all possible placements. The efficiency is improved by breaking these tests into 6 groups qualified by cheap feasibility tests.

This implementation is easily the most efficient of the ones described so far.

4.7 The program

```
GET "mcpl.h"

MANIFEST
  P1=1,    P2=P1*2, P3=P2*2, P4=P3*2, P5=P4*2, P6=P5*2,
  P7=P6*2, P8=P7*2, P9=P8*2, Pa=P9*2, Pb=Pa*2, Pc=Pb*2,

  All=P1+P2+P3+P4+P5+P6+P7+P8+P9+Pa+Pb+Pc,

                        A=1,   A1=A<<1,A2=A<<2,A3=A<<3,A4=A<<4,
Bz=A<<4,By=A<<5,Bx=A<<6,B=A<<7,B1=B<<1,B2=B<<2,B3=B<<3,
        Cy=B<<5,Cx=B<<6,C=B<<7,C1=C<<1,C2=C<<2,
                Dx=C<<6,D=C<<7,D1=D<<1,
                        E=D<<7

STATIC
  border = [
    #1004020100, #0402010040, #0201004020, #0100402010,  // 0
                 #0040201004, #4020100402, #2010040201,
    #1004020100, #0402010040, #0201004020, #0100402010,  // 1
                 #0040201004, #4020100402, #2010040201,
    #1004020100, #0402010040, #0201004020, #0100402010,  // 2
                 #0040201004, #4020100402, #2010040201,
    #1004020100, #0402010040, #0201004020, #0100402010,  // 3
                 #0040201004, #4020100402, #2010040201,
    #1004020100, #0402010040, #0201004020, #0100402010,  // 4
                 #0040201004, #4020100402, #2010040201,
    #1004020100, #0402010040, #0201004020, #0100402010,  // 5
                 #0040201004, #4020100402, #2010040201,
    #7004020100, #7402010040, #7601004020, #7700402010,  // 6
                 #7740201004, #7760100402, #7770040201,
    #7774020100, #7776010040, #7777004020, #7777402010,  // 7
                 #7777601004, #7777700402, #7777740201,
    #7777760100, #7777770040, #7777774020, #7777776010,  // 8
                 #7777777004, #7777777402, #7777777601,
    #7777777700, #7777777740, #7777777760, #7777777770,  // 9
                 #7777777774, #7777777776, #7777777777 ],
  count=0

FUN start : =>
  count := 0
  try(0, 0, 0, border!0, 0)
  writef("\nThe total number of solutions is %d\n", count)
  RETURN 0
```

```
FUN try : bits, piece, p, brd, used =>

  IF brd&bits OR used&piece RETURN

  brd, used +:= bits, piece

  IF used=All DO { count++;
                   writef("solution %4d\n", count)
                   RETURN
                 }

  WHILE brd&1 DO { brd>>:=1; p++ }
  brd |:= border!p

  UNLESS (A1+A2)&brd DO
  { try(A+A1+A2+A3+A4, P2, p, brd, used)
    try(A+A1+A2+A3+ B, P3, p, brd, used)
    try(A+A1+A2+A3+B1, Pb, p, brd, used)
    try(A+A1+A2+A3+B2, Pb, p, brd, used)
    try(A+A1+A2+A3+B3, P3, p, brd, used)
    try(A+A1+A2+ B+Bx, P4, p, brd, used)
    try(A+A1+A2+ B+B1, P5, p, brd, used)
    try(A+A1+A2+ B+B2, P7, p, brd, used)
    try(A+A1+A2+ B+ C, P8, p, brd, used)
    try(A+A1+A2+B1+B2, P5, p, brd, used)
    try(A+A1+A2+B1+C1, P6, p, brd, used)
    try(A+A1+A2+B2+B3, P4, p, brd, used)
    try(A+A1+A2+B2+C2, P8, p, brd, used)
  }
  UNLESS (A1+ B)&brd DO
  { try(A+A1+ B+Bx+By, P4, p, brd, used)
    try(A+A1+ B+Bx+Cx, P9, p, brd, used)
    try(A+A1+ B+Bx+B1, P5, p, brd, used)
    try(A+A1+ B+Bx+ C, P1, p, brd, used)
    try(A+A1+ B+B1+B2, P5, p, brd, used)
    try(A+A1+ B+B1+ C, P5, p, brd, used)
    try(A+A1+ B+B1+C1, P5, p, brd, used)
    try(A+A1+ B+ C+Cx, Pc, p, brd, used)
    try(A+A1+ B+ C+C1, P7, p, brd, used)
    try(A+A1+ B+ C+ D, P3, p, brd, used)
  }
  UNLESS (A1+B1)&brd DO
  { try(A+A1+B1+B2+B3, P4, p, brd, used)
    try(A+A1+B1+B2+C2, P9, p, brd, used)
//  try(A+A1+B1+B2+C1, P1, p, brd, used)
    try(A+A1+B1+C1+ C, P7, p, brd, used)
    try(A+A1+B1+C1+C2, Pc, p, brd, used)
    try(A+A1+B1+C1+D1, P3, p, brd, used)
  }
  UNLESS  (B+Bx)&brd DO
  { try(A+ B+Bx+By+Bz, P3, p, brd, used)
    try(A+ B+Bx+By+Cy, Pc, p, brd, used)
    try(A+ B+Bx+By+B1, Pb, p, brd, used)
    try(A+ B+Bx+By+ C, P6, p, brd, used)
    try(A+ B+Bx+By+Cx, P1, p, brd, used)
    try(A+ B+Bx+Cx+Cy, P9, p, brd, used)
    try(A+ B+Bx+Cx+ C, P5, p, brd, used)
    try(A+ B+Bx+Cx+Dx, P4, p, brd, used)
//  try(A+ B+Bx+Cx+B1, P1, p, brd, used)
    try(A+ B+Bx+B1+B2, Pb, p, brd, used)
    try(A+ B+Bx+B1+ C, Pa, p, brd, used)
//  try(A+ B+Bx+B1+C1, P1, p, brd, used)
    try(A+ B+Bx+ C+ D, Pb, p, brd, used)
//  try(A+ B+Bx+ C+C1, P1, p, brd, used)
  }
  UNLESS  (B+B1)&brd DO
  { try(A+ B+B1+B2+A2, P7, p, brd, used)
    try(A+ B+B1+B2+B3, P3, p, brd, used)
    try(A+ B+B1+B2+ C, P6, p, brd, used)
    try(A+ B+B1+B2+C2, Pc, p, brd, used)
//  try(A+ B+B1+B2+C1, P1, p, brd, used)
    try(A+ B+B1+ C+C1, P5, p, brd, used)
    try(A+ B+B1+ C+ D, Pb, p, brd, used)
//  try(A+ B+B1+ C+Cx, P1, p, brd, used)
    try(A+ B+B1+C1+C2, P9, p, brd, used)
    try(A+ B+B1+C1+D1, P4, p, brd, used)
  }
  UNLESS  (B+ C)&brd DO
  { try(A+ B+ C+Cx+Cy, P8, p, brd, used)
    try(A+ B+ C+Cx+Dx, P4, p, brd, used)
    try(A+ B+ C+Cx+C1, P6, p, brd, used)
    try(A+ B+ C+Cx+ D, Pb, p, brd, used)
    try(A+ B+ C+C1+C2, P8, p, brd, used)
    try(A+ B+ C+C1+ D, Pb, p, brd, used)
    try(A+ B+ C+C1+D1, P4, p, brd, used)
    try(A+ B+ C+ D+Dx, P3, p, brd, used)
    try(A+ B+ C+ D+D1, P3, p, brd, used)
    try(A+ B+ C+ D+ E, P2, p, brd, used)
  }
```

4.8 The two player pentomino game

A game of pentominoes between two people can be played on a chess board. The players play alternately and the first who is unable to move loses. A draw is clearly not possible in this game. It has been shown by Orman [Orm96] that the first player can force a win. Various winning first moves were verified by a program that exhaustively searched the game tree. The program presented here performs a simple version of such a search. It could easily be augmented to include the heuristics used by Orman to improve its efficiency but this has not been done here since it obscures the bit pattern techniques which are the purpose of this example.

A piece placement can be represented a pattern of 76 bits composed of 64 bits to identify the board squares used and 12 bits to identify the piece. Two placements are mutually compatible if the intersection of their bit patterns is empty.

The program first precomputes the complete set of 2308 possible placements as a list of triplets placing them between the pointers `p1` and `q1`. This is done by the code:

```
p1 := stackp
mappieces addallrots
q1 := stackp
```

which calls `init` 12 times, passing it `addallrots` and bit patterns giving the shape and identity of each pentomino. `init` computes all translations, both horizontally and vertically, of its given pentomino passing the 76 bit representation to `addallrots` by means of the call: `f(w1, w0, piece)`, `f` being the first argument of `init`. `addallrots` calls `addpos` for each of the 8 possible rotations and reflections the placement can have. Right to left reflection of the 8×8 board represented by a pair of 32 bit words is done by the following function:

```
FUN reflect : [w1, w0] =>
  w0 := (w0&#x01010101)<<7 | (w0&#x80808080)>>7 |
        (w0&#x02020202)<<5 | (w0&#x40404040)>>5 |
        (w0&#x04040404)<<3 | (w0&#x20202020)>>3 |
        (w0&#x08080808)<<1 | (w0&#x10101010)>>1

  w1 := (w1&#x01010101)<<7 | (w1&#x80808080)>>7 |
        (w1&#x02020202)<<5 | (w1&#x40404040)>>5 |
        (w1&#x04040404)<<3 | (w1&#x20202020)>>3 |
        (w1&#x08080808)<<1 | (w1&#x10101010)>>1
```

Its argument is a pointer to the pair of 32 bit words `w1` and `w0` that represent the lower and upper half of the board. Each half is reflected by simple (if tedious) assignments. The rotate function is perhaps slightly more subtle. It definition is as follows:

```
FUN rotate : [w1, w0] =>
  LET a = (w0&#x0F0F0F0F)<<4 | w1&#x0F0F0F0F
  LET b = (w1&#xF0F0F0F0)>>4 | w0&#xF0F0F0F0

  a  := (a & #X00003333)<<2 | (a & #X0000CCCC)<<16 |
        (a & #XCCCC0000)>>2 | (a & #X33330000)>>16

  b  := (b & #X00003333)<<2 | (b & #X0000CCCC)<<16 |
        (b & #XCCCC0000)>>2 | (b & #X33330000)>>16

  w0 := (a & #X00550055)<<1 | (a & #X00AA00AA)<<8  |
        (a & #XAA00AA00)>>1 | (a & #X55005500)>>8

  w1 := (b & #X00550055)<<1 | (b & #X00AA00AA)<<8  |
        (b & #XAA00AA00)>>1 | (b & #X55005500)>>8
```

Here the rotation is done in three stages, by first moves the four 4×4 corners cyclicly round one position, then the 16 2×2 sized squares are moved in smaller cycles, and finally, the individual bits of these 2×2 squares are rotated. The mechanism is efficient since many of the individual bit movements are done in simultaneously.

The function `addpos` pushes a 76 bit placement represented by it arguments `w1`, `w0` and `piece` onto the placement stack provided it is distinct from all those already present. This check is done with the aid of a closed hash table of size 4001. The hash function: `ABS((w1+1)*(w0+3)) MOD Hashtabsize` was chosen with care to achieve reasonable efficiency. I was not able to devise a satisfactory perfect hashing function for the job.

The set of 296 distinct first moves (all placements with rotational and reflective symmetries removed) are calculated initially and placed between `p0` and `q0`. the code to do this is:

```
p0 := stackp
mappieces addminrot
q0 := stackp
```

where `addminrot` is a function adds only a carefully selected "minimum" of the 8 possible rotations and reflections of each placement it is given.

4.9 Exploring the move tree

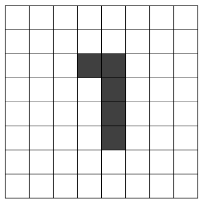
Having constructed the sets of initial moves and placements, the exploration of the move tree is initiated by the code:

```
TEST try76(1, p0, q0, p1, q1)
THEN writes "\nFirst player can force a win\n"
ELSE writes "\nFirst player cannot force a win\n"
```

The first argument (`1`) is the move number being considered which must be selected from the moves bracketed between `p0` and `q0`, and the last two arguments bracket the set of placements from which the replies can be selected is specified.

The function `try76` selects a move from its given move set placing it 76 bit representation in `w1`, `w0` and `piece`. It then constructs the set of possible replies that do not conflict with the selected move, pushing them onto a stack. If no replies are possible then the selected move is a winner and `try76` return `TRUE` to indicate this success. If, however, replies are possible `try76` typically calls itself recursively to explore the tree of moves from the new board state.

A minor complication in the encoding of `try76` is to allow it to follow a user specified path in the move tree. This allows the program to explore the move tree from a particular state, such as just as after the following first move:



which, as Osman states, can force a win. This move turns out to have 1181 possible replies, and so each of these can be separately explored.

Each move uses one pentomino and 5 board squares so after two moves, 10 squares and two pentominoes will have been used. The corresponding 12 bit positions will thus be zero for every possible reply. These replies can thus be compressed from 76 bits to 64 bits without loss of information. This compression clearly improves the efficiency of the subsequent search.

The compression is done by the function `cmp64`, whose arguments are: the move number (`n`), whose 76 bit possible piece placements are bracketed by the arguments `p` and `q`. The remaining arguments `w1bits`, `w0bits` and `pbits` identify the board squares and pieces used by the available placements. These were computed in `try76` while the set of replies were being formed.

The function `cmpput64` is called by `cmp64` to compact each 76 bit placement to its 64 bits form by packing the remaining piece bits into available board positions, pushing the resulting word pair onto the placement stack. The encoding of `cmpput64` is straightforward.

Having completed the compression to 64 bit form, `cmp64` calls `try64` to resume the tree exploration using this new representation.

The stucture of `try64` is the same as that of `try76`, but is more efficient since it uses word pairs rather than triplets and also does not include the code to following a user provided path.

After 8 moves, a further compression to 32 bits is possible since $32 > 76-8\times6$. In fact this compression can often be done after 7 or even 6 moves, since sufficient unoccupied squares may be unreachable using any of the remaining pieces. This compression is triggered by testing:

```
bits w1bits + bits w0bits <= 32
```

where `bits` counts the number of ones present in its argument. It definition is as follows:

```
FUN bits : 0 => 0
         : w => 1 + bits(w&(w-1))
```

Notice that the expression `w&(w-1)` removes a single one from `w`.

The compaction from 64 bits to 32 is done by packing the bits in the senior half into available positions in the junior half, and it is done by the function `cmp32` aided by `cmpput32`. These functions a somewhat similar to `cmp64` and `cmpput64`

The final tree exploration using 32 bit representations is then carried out by `try32` which is similar but even simpler than `try64`.

Information concerning the compression is saved in `w1bits64`, `w0bits64`, `pbits64`, `prn64`, `w1bits32`, `w0bits32` and `prn32` so that the compressed representations can be expanded to 76 bits when printing the board for debugging purposes.

The function to print the board state is called `pr` and it uses `exp32` and `exp64` to expand the board representation from 32 and 64 bit representations, respectively.

4.10 The program

```
GET "mcpl.h"

GLOBAL count:200,
       stackv, stackp, stackt, p0, q0, p1, q1

MANIFEST
  P1=1,    P2=1<<1, P3=1<<2, P4 =1<<3, P5 =1<<4,  P6 =1<<5,
  P7=1<<6, P8=1<<7, P9=1<<8, P10=1<<9, P11=1<<10, P12=1<<11

FUN setup : =>
   // Initialise the set of possible piece placements on
   // the 8x8 board allowing for all rotations, reflections
   // and translations. The generate the set of truly
   // distinct first moves.

   stackv := getvec 500000
   stackt := @ stackv!500000
   stackp := stackv

   p1 := stackp
   mappieces addallrots
   q1 := stackp
   writef("\nThere are %4d possible first moves", (q1-p1)/(3*Bpw))
   p0 := stackp
   mappieces addminrot
   q0 := stackp
   writef("\nof which  %4d are truly distinct\n", (q0-p0)/(3*Bpw))

FUN mappieces : f =>
   hashtab := getvec Hashtabsize
   FOR i = 0 TO Hashtabsize DO hashtab!i := 0

   init(f,     #x1F, P1)  //  * * * * *     *
   init(f, #x020702, P2)  //              * * *
                          //                *

   init(f,   #x010F, P3)  //  * * * *         *
   init(f, #x010701, P4)  //        *     * * *
                          //                  *

   init(f,   #x0703, P5)  //    * *       * * *
   init(f,   #x030E, P6)  //  * * *           * *

   init(f, #x070101, P7)  //      *       *
   init(f, #x030604, P8)  //      *       * *
                          //  * * *         * *

   init(f,   #x0507, P9)  //  * * *       *
   init(f, #x010704, P10) //  *   *       * * *
                          //                  *

   init(f,   #x0F02, P11) //      *          *
   init(f, #x010702, P12) //  * * * *      * * *
                          //                   *
   freevec hashtab
```

```
FUN freestack : => freevec stackv

FUN addminrot : w1, w0, piece =>
  LET mw1=w1, mw0=w0

  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  reflect(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  rotate(@w1)
  IF w1<mw1 OR w1=mw1 AND w0<mw0 DO mw1, mw0 := w1, w0
  addpos(mw1, mw0, piece)

FUN addallrots : w1, w0, piece =>
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)
  reflect(@w1)
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)
  rotate(@w1)
  addpos(w1, w0, piece)

FUN init : f, word0, piece =>
  LET word1 = 0

  { LET w1=word1, w0=word0

    { f(w1, w0, piece)
      IF (w0|w1) & #x80808080 BREAK // can't move left any more
      w1, w0 <<:= 1, 1              // move piece left one place
    } REPEAT

    IF word1 & #xFF000000 RETURN
    word1 := word1<<8 + word0>>24
    word0 <<:= 8
  } REPEAT
```

```
STATIC   hashtab

MANIFEST Hashtabsize = 4001 // Large enough for 2308 entries

FUN addpos : w1, w0, piece =>
  LET hashval = ABS((w1+1)*(w0+3)) MOD Hashtabsize

  { LET p = hashtab!hashval

    UNLESS p DO { hashtab!hashval := stackp // Make new entry
                  !stackp+++ := w1
                  !stackp+++ := w0
                  !stackp+++ := piece
                  RETURN
                }

    IF p!0=w1 AND p!1=w0 RETURN             // Match found

    hashval++
    IF hashval>Hashtabsize DO hashval := 0
  } REPEAT


FUN reflect : [w1, w0] =>
  w0 := (w0&#x01010101)<<7 | (w0&#x80808080)>>7 |
        (w0&#x02020202)<<5 | (w0&#x40404040)>>5 |
        (w0&#x04040404)<<3 | (w0&#x20202020)>>3 |
        (w0&#x08080808)<<1 | (w0&#x10101010)>>1

  w1 := (w1&#x01010101)<<7 | (w1&#x80808080)>>7 |
        (w1&#x02020202)<<5 | (w1&#x40404040)>>5 |
        (w1&#x04040404)<<3 | (w1&#x20202020)>>3 |
        (w1&#x08080808)<<1 | (w1&#x10101010)>>1

FUN rotate : [w1, w0] =>
  LET a = (w0&#x0F0F0F0F)<<4 | w1&#x0F0F0F0F
  LET b = (w1&#xF0F0F0F0)>>4 | w0&#xF0F0F0F0

  a  := (a & #X00003333)<<2 | (a & #X0000CCCC)<<16 |
        (a & #XCCCC0000)>>2 | (a & #X33330000)>>16

  b  := (b & #X00003333)<<2 | (b & #X0000CCCC)<<16 |
        (b & #XCCCC0000)>>2 | (b & #X33330000)>>16

  w0 := (a & #X00550055)<<1 | (a & #X00AA00AA)<<8  |
        (a & #XAA00AA00)>>1 | (a & #X55005500)>>8

  w1 := (b & #X00550055)<<1 | (b & #X00AA00AA)<<8  |
        (b & #XAA00AA00)>>1 | (b & #X55005500)>>8

FUN bits : 0 => 0
         : w => 1 + bits(w&(w-1))
```

```
STATIC path = VEC 12,
        w1v = VEC 12,
        w0v = VEC 12,
        mvn = VEC 12,
        mvt = VEC 12

FUN start : =>
  LET argv = VEC 50
  LET stdout = output()

  IF rdargs(",,,,,,,,,,,,TO/K", argv, 50)=0 DO
  { writef "Bad arguments\n"
    RETURN 0
  }

  FOR i = 0 TO 11 DO path!(i+1) := argv!i -> str2numb(argv!i), -1

  UNLESS argv!12=0 DO selectoutput(findoutput(argv!12))

  setup()

  TEST try76(1, p0, q0, p1, q1)
  THEN writes "\nFirst player can force a win\n"
  ELSE writes "\nFirst player cannot force a win\n"

  freestack()

  UNLESS argv!12=0 DO endwrite()

  RETURN 0

FUN try76 : n, p, q, np, nq =>
  LET s=stackp, t=p, lim=q

  UNLESS path!n<0 DO { t := @p!(3*(path!n-1)); IF t<lim DO lim := t+1 }

  WHILE t < lim DO
  { LET w1    = !t+++            // Choose a move
    LET w0    = !t+++
    LET piece = !t+++

    w1v!n, w0v!n := w1, w0       // Save the move for printing
    mvn!n, mvt!n := (t-p)/(3*Bpw), (q-p)/(3*Bpw)

    IF path!n>=0 AND path!(n+1)<0 DO
    { writef "\nConsidering board position:"
      FOR i = 1 TO n DO writef(" %d/%d", mvn!i, mvt!i)
      newline(); newline(); pr n
    }
    LET r=np, w1bits=0, w0bits=0, pbits=0
    stackp := s

    UNTIL r>=nq DO      // Form the set of of possible replies
    { LET a = !r+++
      LET b = !r+++
      LET c = !r+++
      UNLESS a&w1 OR b&w0 OR c&piece DO { !stackp+++ := a
                                          !stackp+++ := b
                                          !stackp+++ := c
                                          w1bits |:= a
                                          w0bits |:= b
                                          pbits  |:= c
                                        }
    }

    // The possible replies are stored between s and stackp

    IF s=stackp RETURN TRUE // The chosen move is a winner

    // Explore the possible replies
    TEST n>=2 AND path!(n+1)<0
    THEN UNLESS cmp64(n+1, s, stackp, w1bits, w0bits, pbits) RETURN TRUE
    ELSE UNLESS try76(n+1, s, stackp, s, stackp  )           RETURN TRUE
  }

  // We cannot find a winning move from the available moves
  stackp := s
  RETURN FALSE

FUN cmp64 : n, p, q, w1bits, w0bits, pbits =>
  LET s = stackp

  w1bits64, w0bits64, pbits64, prn64 := w1bits, w0bits, pbits, n

  // Compress the representation of the moves from 76 to 64 bits.
  UNTIL p>=q DO { LET w1    = !p+++
                  LET w0    = !p+++
                  LET piece = !p+++
                  cmpput64(w1, w0, piece)
                }

  LET res = try64(n, s, stackp)

  prn64 := 20
  stackp := s
  RETURN res
```

```
FUN try64 : n, p, q =>
  LET s=stackp, t=p

  WHILE t < q DO
  { stackp := s

    LET w1 = !t+++     // Choose a move
    LET w0 = !t+++

    w1v!n, w0v!n := w1, w0

    mvn!n, mvt!n := (t-p)/(2*Bpw), (q-p)/(2*Bpw)

    IF n=4 DO
    { writef("\n\nTrying Move %d: %3d/%d:\n", n, mvn!n, mvt!n)
      pr n
    }
    IF n=5 DO newline()
    IF n=6 DO
    { FOR i = 1 TO n DO writef("%3d/%d ", mvn!i, mvt!i)
      writes "       \^m"
    }

    LET r=p, w1bits=0, w0bits=0

    UNTIL r>=q DO      // Form the set of of possible replies
    { LET a = !r+++
      LET b = !r+++
      UNLESS a&w1 OR b&w0 DO { !stackp+++ := a
                               !stackp+++ := b
                               w1bits, w0bits |:= a, b
                             }
    }

    // The possible replies are stored between s and stackp

    IF s=stackp RETURN TRUE // Move n is a winner

    // See if this move n was a winner
    TEST bits w1bits + bits w0bits <= 32
    THEN UNLESS cmp32(n+1, s, stackp, w1bits, w0bits) RETURN TRUE
    ELSE UNLESS try64(n+1, s, stackp)                 RETURN TRUE
  }

  // We cannot find a winning move from the available moves
  stackp := s
  RETURN FALSE
```

```
FUN cmp32 : n, p, q, w1bits, w0bits =>
  LET s = stackp
  w1bits32, w0bits32, prn32 := w1bits, w0bits, n

  // Compact the representation of the moves from 64 to 32 bits.
  UNTIL p>=q DO { LET w1 = !p+++
                  LET w0 = !p+++
                  cmpput32(w1, w0)
                }

  LET res = try32(n, s, stackp)

  prn32 := 20
  stackp := s
  RETURN res


FUN try32 : n, p, q =>
  LET s=stackp, t=p

  WHILE t < q DO
  { LET w0 = !t+++     // Choose a move

//  w0v!n := w0
//  newline(); pr n

    LET r = p
    stackp := s

    UNTIL r>=q DO      // Form the set of possible replies
    { LET a = !r+++
      UNLESS a&w0 DO !stackp+++ := a
    }

    IF s=stackp RETURN TRUE      // Move n is a winner
    IF n=11 LOOP                 // Move n is a loser
    UNLESS try32(n+1, s, stackp) RETURN TRUE
  }

  // We cannot find a winning move from the available moves
  stackp := s
  RETURN FALSE
```

```
STATIC
  chs = CVEC 64,
  w1bits64, w0bits64, pbits64, prn64=20,
  w1bits32, w0bits32,          prn32=20,
  prw1, prw0

FUN pr : n =>
  FOR i = 1 TO 64 DO chs%i := '.'

  FOR p = 1 TO n DO
  { LET ch = 'A'+p-1
    IF p=n DO ch := '*'
    prw1, prw0 := w1v!p, w0v!p

    IF p>=prn32 DO exp32()  // expand from 32 to 64 bits
    IF p>=prn64 DO exp64()  // expand from 64 to 76 bits
    // prw1 and prw0 now contain the board bits

    FOR i = 1 TO 64 DO  // Convert to and 8x8 array of chars
    { IF prw0&1 DO chs%i := ch
      prw0 >>:= 1
      UNLESS i MOD 32 DO prw0 := prw1
    }
  }

  FOR i = 1 TO 64 DO    // Output the 8x8 array
  { writef(" %c", chs%i)
    IF i MOD 8 = 0  DO newline()
  }
  newline()
```

```
FUN cmpput64 : w1, w0, piece =>
  LET w1bits=~w1bits64, w0bits=~w0bits64, pbits=pbits64
  LET pbit = ?
  WHILE pbits AND w0bits DO
  { LET w0bit = w0bits & -w0bits
    pbit  := pbits  & -pbits
    IF piece&pbit DO w0 |:= w0bit // Move a piece bit into w0
    pbits  -:= pbit
    w0bits -:= w0bit
  }
  WHILE pbits AND w1bits DO
  { LET w1bit = w1bits & -w1bits
    pbit  := pbits  & -pbits
    IF piece&pbit DO w1 |:= w1bit // Move a piece bit into w1
    pbits  -:= pbit
    w1bits -:= w1bit
  }
  !stackp+++ := w1
  !stackp+++ := w0

FUN cmpput32 : w1, w0 =>
  LET w1bits=w1bits32, w0bits=~w0bits32
  WHILE w1bits AND w0bits DO
  { LET w1bit = w1bits & -w1bits
    LET w0bit = w0bits & -w0bits
    IF w1&w1bit DO w0 |:= w0bit // Move a w1 bit into w0
    w1bits -:= w1bit
    w0bits -:= w0bit
  }
  !stackp+++ := w0

FUN exp64 : =>
  prw1 &:= w1bits64  // Remove the piece bits from
  prw0 &:= w0bits64  // the w1 and w0 bit patterns

FUN exp32 : => // Move various bits from prw0 into prw1
  LET w1bits=w1bits32, wobits=~w0bits32
  prw1 := 0
  WHILE w1bits AND w0bits DO
  { LET w1bit = w1bits & -w1bits
    LET w0bit = w0bits & -w0bits
    IF prw0&w0bit DO { prw0 -:= w0bit; prw1 |:= w1bit }
    w1bits -:= w1bit
    w0bits -:= w0bit
  }
```

5 The Cardinality of D_3

This is a program to show that there are 120549 elements in the domain D_3 as described on pages 113–115 of "Denotational Semantics" by J.E.Stoy[Sto77].

We start with a base domain (D_0) having just the two elements $\perp$ and $\top$ satisfying the relation $\perp \sqsubseteq \top$.

The domain $D_1 = D_0 \rightarrow D_0$ is the domain of monotonic functions from D_0 to D_0. It contains three elements denoted by $\perp$, 1 and $\top$ satisfying the relations $\perp \sqsubseteq 1$ and $1 \sqsubseteq \top$.

The domain $D_2 = D_1 \rightarrow D_1$ is the domain of monotonic functions from D_1 to D_1. It contains ten elements that we will denote by the letters $A \ldots J$, satisfying relations that form the lattice shown in figure 2.

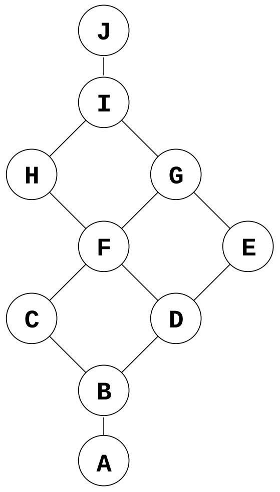


Figure 2: The D_2 lattice

Finally, D_3 is defined to be the domain of monotonic functions $D_2 \rightarrow D_2$, and the problem is to compute the number of elements in D_3.

A function $f \epsilon D_2$ can be denoted by a sequence of ten elements $abcdefghij$ giving the values of $f(A), f(B), \ldots, f(J)$, respectively. The program searches for all such functions satisfying the monotonicity constraint:

$$x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$$

It does this by successively selecting values for j, $i \ldots a$, passing, on at each stage, the contraints about future selections. All possible values for j are tried by the call:

```
try(fJ, A+B+C+D+E+F+G+H+I+J)
```

For each possible setting (x, say) it calls `fJ(tab!`x`)` whose argument represents the set of elements that can be assigned to i, the vector `tab` providing the mapping between an element and the set of elements smaller than it.

Sometimes it is neccesary to pass two contraint sets to `try`. An example is the call:

```
try(fF, a&b, a)
```

Here,`a` is the set of elements smaller than the one already chosen for g and `b` is the set of elements smaller than that already chosen for h. All possible values for f are thus in `a&b`. The last argument of this call provides the set of possible values that e can have. The definitions of the functions `fA` to `fJ` are thus straightforward encodings of the monotonicity constraints resulting from the D_2 lattice.

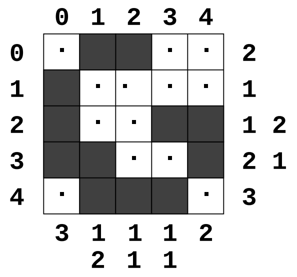
5.1 The program

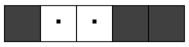
```
GET "mcpl.h"

MANIFEST  A=1,     B=1<<1, C=1<<2, D=1<<3, E=1<<4,
          F=1<<5,  G=1<<6, H=1<<7, I=1<<8, J=1<<9

STATIC    tab = VEC J,
          count = 0

FUN start : => tab!J := A+B+C+D+E+F+G+H+I+J
               tab!I := A+B+C+D+E+F+G+H+I
               tab!H := A+B+C+D  +F  +H
               tab!G := A+B+C+D+E+F+G
               tab!F := A+B+C+D  +F
               tab!E := A+B  +D+E
               tab!D := A+B  +D
               tab!C := A+B+C
               tab!B := A+B
               tab!A := A

// tab!e = the set of elements <= e in the lattice D2

               try(fJ, A+B+C+D+E+F+G+H+I+J)
               writef("Number of elements in D3 = %d\n", count)
               RETURN 0

FUN try : f, a, b => UNTIL a=0 DO { LET x = a & -a
                                    a -:= x
                                    f(tab!x, b)
                                  }

                                  //        J
FUN fJ : a     => try(fI, a)      //        |a
                                  //        I
FUN fI : a     => try(fH, a, a)   //        |a
                                  //       / \
                                  //      H   |
FUN fH : a, b  => try(fG, b, a)   //     a|   |b
                                  //      |   G
FUN fG : a, b  => try(fF, a&b, a) //     b|   |a
                                  //       \ / \
                                  //        F   |
FUN fF : a, b  => try(fE, b, a)   //       a|   |b
                                  //        |   E
FUN fE : a, b  => try(fD, a&b, b) //       b|   |a
                                  //       / \ /
                                  //      |   D
FUN fD : a, b  => try(fC, b, a)   //     b|   |a
                                  //      C   |
FUN fC : a, b  => try(fB, a&b)    //     a|   |b
                                  //       \ /
                                  //        B
FUN fB : a     => try(fA, a)      //        |a
                                  //        A
FUN fA :       => count++         //        |
```

6 Nonograms

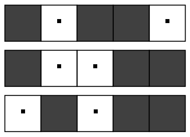
A nonogram is an ancient Japanese puzzle in which a pixel map has to be found that fits into a rectangular grid satisfying constraints consisting of numbers given at the end of each row and at the bottom of each column. Such puzzles have appeared for some time in the London Sunday Telegraph.

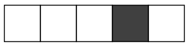
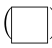
A solution to a typical 5×5 nonogram is given below.

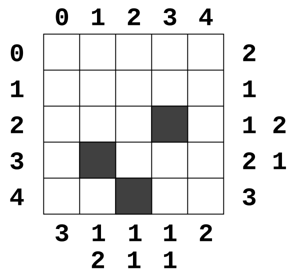


The marked squares form contiguous groups whose lengths must match the numbers at the end of the row or column. For instance, row 2 has the pattern  consisting of contiguous regions of lengths 1 and 2 agreeing with the numbers at the end of that row. Contiguous regions must be separated by at least one dotted square.

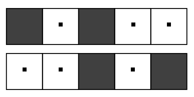
The puzzle can be solved by considering each row in turn and deducing from its numerical constraint whether any of its squares have forced values. For instance, for row 2, the only possible arrangements are:



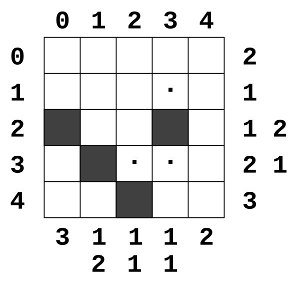
and hence that row must be of the form  where the empty squares () denote unresolved positions. Applying this method to all the rows tells us that any solution must have the form:



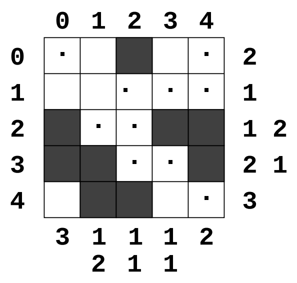
We now apply the same method to the columns. For instance, the only arrangements now possible for column 3 are:



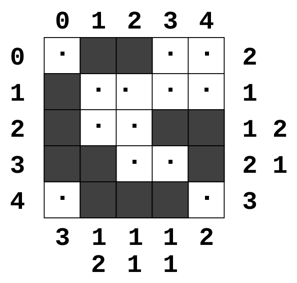
from which we can deduce that this column has the form □⊡■⊡□. After processing each column the state of the board is:



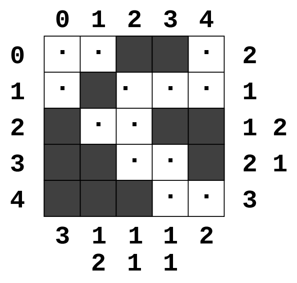
If we continue to apply this process to the rows and columns we eventually reach a state where no further resolution is possible. For this puzzle, this state is:



To continue from here, we have to try both alternative settings (■ or ⊡) for one of the unresolved squares and, for each setting, continue the resolution as before. For this puzzle, setting row 0 column 1 to ■ results in the solution:



and setting it to a dotted square (⊡) another solution is obtained:



6.1 Implementation

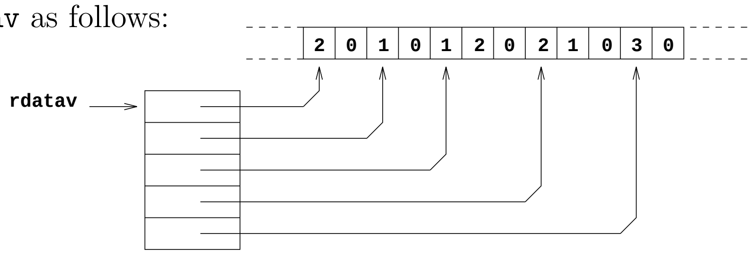
A program to implement this algorithm can clearly take advantage of bit pattern techniques for the representation of the board and the filtering of the constraints, while recursion can be used as a convenient method to control the backtracking.

To solve problems for boards of up to 32×32, we can use two 32-bit words to represent each row. One identifying which positions have known values and other specifying what the known values are. The bit patterns for each row are held in the vectors `knownv` and `boardv`. Rows are numbered from `0` to `rupb` and columns are numbered from `0` to `cupb`. The least significant position of the bit patterns hold information about column 0.

The specification of the puzzle is given by a data file such as the following (which specifies the problem discussed above):

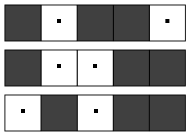
```
row 2
row 1
row 1 2
row 2 1
row 3

col 3
col 1 2
col 1 1
col 1 1
col 2
```

This is read in by the function `readdata` whose argument is the title of the data file. It initialises to vectors `rdatav` and `cdatav` to hold pointers to zero terminated vectors holding the constraint data. On reading the above data, it leaves `rdatav` as follows:



It initialises `cdatav` similarly. A check is made to ensure that number of squares that must be marked to satisfy the row constraints equals the number required to satisfy the column constraints.

The *freedom* of a constraint is the number of different positions the rightmost marked region of each row can have. For row 2, the freedom is 2 since the possible arrangements are:



The freedom of row `r` can be calculated by the call `freedom(rdata!r,cupb)`, where `freedom` is defined as follows:

```
FUN freedom : p, upb => IF !p=0 RETURN 0
                        LET free = upb+2
                        free -:= !p + 1 REPEATWHILE !+++p
                        RETURN free
```

As can be seen, the freedom of a constraint depends on the length of the row, the number of regions and the sum of their lengths.

The function `readdata` calculates the freedom of all constraints saving the results in the appropriate elements of `rfreedomv` for the rows and `cfreedomv` for the columns.

The search for solutions is made by calling `allsolutions` after initialising the elements of `knownv` and `boardv` to zero to indicate that all board positions to unknown. From this state, the rows and columns are processed by a call of `solve` which returns when no more resolution is possible. It returns `FALSE` if it reached a dead-end state from which no solution is possible, otherwise it returns `TRUE`. When no unresolved squares remain, a solution has been found, but otherwise the position of an unresolved square is chosen and placed in `row` and `bit`. The current state of `knownv` and `boardv` is saved in local vectors `kv` and `bv`. The selected square is set to ■ by:

```
        knownv!row, boardv!row |:= bit, bit
```

This branch of the search tree is then explored by the recursive call of `allsolutions`. On return, `knownv` and `boardv` is restored from `kv` and `bv` and the selected square set to the other possible value (⊡) by:

```
        knownv!row |:= bit
```

A tail-recursive call of `allsolutions` is then made by means of the repeat loop. This explores the other branch of the search tree.

The function `solve` processes the rows using the function `dorows` and then flips the board about a diagonal so the the columns can processed by a second call of `dorows`. The flip operation is performed by `flip` which calls `flipbits` twice, once to flip the 32×32 bitmap `boardv` and once for `knownv`. `flipbits` swaps bits (i, j) with (j, i) for all positions in its given bitmap — essentially 1024 bit moves. However, the algorithm does this is 5 stages, swapping square areas of sizes 16, 8, 4, 2, and finally 1, using five calls of `xchbits`. The definitions of these functions are as follows:

```
FUN flipbits : v => xchbits(v, 16, #x0000FFFF)
                    xchbits(v,  8, #x00FF00FF)
                    xchbits(v,  4, #x0F0F0F0F)
                    xchbits(v,  2, #x33333333)
                    xchbits(v,  1, #x55555555)

FUN xchbits
: v, n, mask => LET i = 0
                { FOR j = 0 TO n-1 DO
                  { LET q= @ v!(i+j)
                    LET a=q!0, b=q!n
                    q!0 := a & mask | b<<n &~mask
                    q!n := b &~mask | a>>n & mask
                  }
                  i +:= n+n
                } REPEATWHILE i<32
```

The first argument of `xchbits` is the bitmap vector, the second argument (`n`) give the size of square regions being moved, and the third argument (`mask`) identifies which row bits are involved. The process is closely analagous to the discrete fast Fourier transform algorithm in that it takes $\log n$ stages each of which applies a butterfly operation n times.

The function `dorows` tries (using `try`) all possible mark arrangements that are compatible with the row constraints and the current known state of the board. If it finds that more is now known it sets `change` to true, to cause another iteration (within `solve`) of the resolution process.

It first argument of `try` is a pointer to the size of the next contiguous region to place. If this is zero then the arrangement is complete, and is legal if the call of `ok` returns true. The bitwise OR of all legal arrangements are accumulated in `orsets`, and `andsets` accumulates the corresponding intersection. These are used in `dorows` to determine whether new information has be discovered.

If the first argument of `try` points to a non zero value, it attempts to place a region of this size in each remaining possible positions, calling `try` recursively to place the remaining regions.

6.2 Observation

Although this algorithm works well enough, it efficiency could be improved by not reprocessing any row whose known information has not changed since it was last processed.

6.3 The program

```
GET "mcpl.h"

STATIC
  cupb, rupb, spacev, spacet, spacep, boardv, knownv,
  cdatav, rdatav, cfreedomv, rfreedomv,
  rowbits, known, orsets, andsets,
  change, count, tracing

FUN start : =>
  LET argv = VEC 50
  LET datafile = "nonograms/demo"

  IF rdargs("DATA,TO/K,TRACE/S", argv, 50)=0 DO
  {  writef "Bad arguments for NONOGRAM\n"
     RETURN 20
  }

  UNLESS argv!0=0 DO datafile := argv!0
  UNLESS argv!1=0 DO
  { LET out = findoutput(argv!1)
    IF out=0 DO
    { writef("Cannot open file %s\n", argv!1)
      RETURN 20
    }
    selectoutput(out)
  }

  tracing := argv!2

  UNLESS initdata() DO
  { writes "Cannot allocate workspace\n"
    UNLESS argv!1=0 DO endwrite()
    retspace()
    RETURN 20
  }

  UNLESS readdata datafile DO
  { writes "Cannot read the data\n"
    UNLESS argv!1=0 DO endwrite()
    retspace()
    RETURN 20
  }

  count := 0
  allsolutions()

  writef("%d solution%s found\n", count, count=1 -> "", "s")

  UNLESS argv!1=0 DO endwrite()
  retspace()
  RETURN 0
```

```
FUN initdata : =>            // returns TRUE if successful
  spacev   := getvec 100000
  spacet   := @ spacev!100000
  spacep   := spacev
  cupb     := 0
  rupb     := 0
  boardv   := getvec 31
  knownv   := getvec 31
  cdatav   := getvec 31
  rdatav   := getvec 31
  cfreedomv:= getvec 31
  rfreedomv:= getvec 31

  IF spacev=0 OR boardv=0 OR knownv=0 OR cdatav=0 OR rdatav=0 OR
     cfreedomv=0 OR rfreedomv=0 RETURN FALSE

  FOR i = 0 TO 31 DO
  { boardv!i    := 0
    knownv!i    := 0
    cdatav!i    := 0
    rdatav!i    := 0
    cfreedomv!i := 0
    rfreedomv!i := 0
  }

  RETURN TRUE

FUN retspace : =>
  IF spacev    DO freevec spacev
  IF boardv    DO freevec boardv
  IF knownv    DO freevec knownv
  IF cdatav    DO freevec cdatav
  IF rdatav    DO freevec rdatav
  IF cfreedomv DO freevec cfreedomv
  IF rfreedomv DO freevec rfreedomv

FUN readdata : filename =>      // Returns TRUE if successful
  LET stdin = input()
  LET data  = findinput filename

  IF data=0 DO
  { writef("Unable to open file %s\n", filename)
    RETURN FALSE
  }

  selectinput data

  LET argv = VEC 200
  cupb, rupb := -1, -1
```

```
{ LET ch = rdch()
  WHILE ch='\s' OR ch='\n' DO ch := rdch()
  IF ch=Endstreamch BREAK
  unrdch()

  IF rdargs("ROW/S,COL/S,,,,,,,,,,,,,,,,,,,", argv, 200)=0 DO
  { writes("Bad data file\n")
    endread()
    selectinput stdin
    RETURN FALSE
  }

  IF argv!0 = argv!1 DO
  { writes "Expecting ROW or COL in data file\n"
    endread()
    selectinput stdin
    RETURN FALSE
  }

  IF argv!0 DO rdatav!++rupb := spacep
  IF argv!1 DO cdatav!++cupb := spacep

  FOR i = 2 TO 20 DO
  { IF argv!i = 0 BREAK
    !spacep+++ := str2numb(argv!i)
  }
  !spacep+++ := 0

} REPEAT

FOR x = 0 TO cupb DO cfreedomv!x := freedom(cdatav!x, rupb)
FOR y = 0 TO rupb DO rfreedomv!y := freedom(rdatav!y, cupb)

IF tracing DO
{ FOR x = 0 TO cupb DO writef("cfreedom!%2d = %2d\n", x, cfreedomv!x)
  FOR y = 0 TO rupb DO writef("rfreedom!%2d = %2d\n", y, rfreedomv!y)
}

endread()

selectinput stdin

UNLESS marks(cdatav, cupb)=marks(rdatav, rupb) DO
{ writes("Data sumcheck failure\n")
  writef("X marks = %d\n", marks(cdatav,cupb))
  writef("Y marks = %d\n", marks(rdatav,rupb))
  RETURN FALSE
}

RETURN TRUE
```

```
FUN marks : v, upb =>
  LET res = 0
  FOR i = 0 TO upb DO { LET p = v!i
                        UNTIL !p=0 DO res +:= !p+++
                      }
  RETURN res


FUN freedom : p, upb => IF !p=0 RETURN 0
                        LET free = upb+2
                        free -:= !p + 1 REPEATWHILE !+++p
                        RETURN free



FUN allsolutions : =>
{ UNLESS solve() RETURN // no solutions can be found from here

  LET row=0, bit=0

  FOR i = 0 TO rupb DO
  { LET unknown = ~ knownv!i
    IF unknown DO { row, bit := i, unknown & -unknown
                    BREAK
                  }
  }

  // test to see if a solution has been found
  IF bit=0 DO
  { writef("\nSolution %d\n\n", ++count)
    prboard()
    RETURN
  }

  // There may be a solution from here.
  // Try both possible settings of the unresolved square
  // given by pos and bit.

  IF tracing DO
  { writes "\nNo more direct resolution available in the following:\n"
    prboard()
  }
  { LET bv = VEC 31
    LET kv = VEC 31

    // save current state
    FOR i = 0 TO 31 DO bv!i, kv!i := boardv!i, knownv!i

    knownv!row, boardv!row |:= bit, bit

    IF tracing DO
    { writes "So, try setting an unresolved square to mark\n"
      prboard()
    }

    allsolutions()

    // restore saved state
    FOR i = 0 TO 31 DO boardv!i, knownv!i := bv!i, kv!i
  }

  // Space for bv and kv is released at this point so that the
  // tail recursive call of allsolutions is more economical.

  knownv!row |:=  bit

  IF tracing DO
  { writes "Try setting a unresolved square to blank\n"
    prboard()
  }
} REPEAT

// solve returns FALSE is no solution possible from here
FUN solve : =>
  { change := FALSE
    UNLESS dorows() RETURN FALSE
    flip()
    UNLESS dorows() DO { flip(); RETURN FALSE }
    flip()
  } REPEATWHILE change

  RETURN TRUE
```

```
// dorows returns FALSE if no solution possible from current state
FUN dorows : =>
  FOR row = 0 TO rupb DO
  { orsets, andsets  := 0, #xFFFFFFFF
    rowbits, known := boardv!row, knownv!row
    try(rdatav!row, 0, 0, rfreedomv!row)

    UNLESS (andsets & orsets) = andsets RETURN FALSE
    rowbits, known |:= andsets, ~orsets | andsets
    IF known=knownv!row LOOP
    boardv!row, knownv!row, change := rowbits, known, TRUE
    IF tracing DO { newline(); prboard() }
  }
  RETURN TRUE

FUN try
: [0], set, ?, ?  =>  // end of piece list

    IF ok(set, cupb+1) DO  // Have we found a valid setting
    { IF tracing DO        // Yes, we have.
      { FOR col = 0 TO cupb DO
          writef(" %c",  set>>col & 1 -> '*', '.')
        writes "  possible line\n"
      }
      orsets  |:= set      // Accumulate the "or" and
      andsets &:= set      // "and" sets.
    }

: [size, next], set, col, free =>

    LET piece = 1<<size - 1
    FOR i = 0 TO free DO
    { LET nset = set | piece<<(col+i)
      LET ncol = col+i+size+1
      IF ok(nset, ncol) DO try(@ next, nset, ncol, free-i)
    }

// ok returns TRUE if the given mark placement is
// compatible with the current known board settings
FUN ok : set, npos =>
  LET mask = known & (1<<npos - 1)
  RETURN (set XOR rowbits) & mask = 0
```

```
// flip will flip the nonogram about a diagonal axis from
// the top left of the picture.
// Remember that the top left most position is represented
// by the least significant bits of boardv!0 and knownv!0

FUN flip : =>
  cdatav,    rdatav    := rdatav,    cdatav
  cfreedomv, rfreedomv := rfreedomv, cfreedomv
  cupb,      rupb      := rupb,      cupb

  flipbits boardv
  flipbits knownv

// flipbits swaps bit (i,j) with bit (j,i) for
// all bits in a 32x32 bitmap. It does it in 5 stages
// by swapping square areas of sizes 16, 8, 4, 2 and
// finally 1.

FUN flipbits : v => xchbits(v, 16, #x0000FFFF)
                    xchbits(v,  8, #x00FF00FF)
                    xchbits(v,  4, #x0F0F0F0F)
                    xchbits(v,  2, #x33333333)
                    xchbits(v,  1, #x55555555)

FUN xchbits
: v, n, mask => LET i = 0
                { FOR j = 0 TO n-1 DO
                  { LET q= @ v!(i+j)
                    LET a=q!0, b=q!n
                    q!0 := a & mask | b<<n &~mask
                    q!n := b &~mask | a>>n & mask
                  }
                  i +:= n+n
                } REPEATWHILE i<32

FUN  prboard : =>
  FOR y = 0 TO rupb DO
  { LET row=boardv!y, known=knownv!y
    FOR x = 0 TO cupb DO
      TEST (known>>x & 1)=0
      THEN writes(" ?")
      ELSE TEST (row>>x & 1)=0
           THEN writes(" .")
           ELSE writes(" M")
    newline()
  }
  newline()
```

7 Boolean Satisfiability

The program described here is an implementation of a variant of the Davis-Putman algorithm[DP60] to enumerates the settings of propositional variables that will cause a given boolean expression to be satisfied.

The boolean expression is given in conjunctive normal form held in a file. Positive and negative integers represent positive and negated propositional variables, respectively. The terms (or clauses) are given as sequences of integers enclosed in parentheses and the sequence of terms is terminated by the end of file. Thus, for example, the expression: $(A \vee \overline{B}) \wedge (A \vee B \vee C) \wedge (\overline{A} \vee B \vee C)$ could be represented by the file:

```
(1 -2)
(1 2 3)
(-1 2 3)
```

For simplicity, only variables numbered 1 to 32 are allowed.

For the expression to be satisfied each of its terms must be satisfied, and for a term to be satisfied either one of its positive variables must be set to true, or one of its negated variables must be false. The algorithm essentially explores a binary tree whose nodes represent boolean expressions with edges leading to expressions in which a selected variable is set either to true or false. A leaf of the tree is occurs when either no terms remain to be satisfied or when an empty (and therefore unsatisfiable) term is found. The efficiency is greatly affected by the choice of which variable to set at each stage.

A term can be empty, a singleton, a doublet or a term containing more than two variables. An expression containing an empty term cannot be satisfied. If it contains a singleton then the singleton variable has a forced value. If the expression contains neither empty nor singleton terms, then a variable is selected and set successively to its two possible boolean values. If any doublets are present, the positive or negated variable that occurs most frequently in doublets is chosen, otherwise the most frequently used positive or negated variable occurring in any term is preferred. The advantage of doublet variables is that one of their alternative settings will generate singletons. It is probably best to set the variable first to that value that causes its terms to be satisfied, since this tends to lead to a solution earlier. If there are no solutions, or we are enumerating all of them, then the order in which the two tree branches are searched has no effect on the total time taken to complete the task.

This strategy can be implemented conveniently using bit pattern representions of the terms. In this implementation, a term is represented by two 32 bit values, the first identifying its positive variables and the second the negated ones. The function `readterms` reads the file of terms pushing them, as word pairs, onto a term stack whose free end is pointed to by `stackp`. The number of variables used is returned in `varcount`. This is calculated by evaluating `bits(all)`, where `all` identifies all variables used in the expression. The definition of `bits` has been described already on page 38.

The call `try(p, q, tset, fset)` explores a node of the tree. The arguments `p` and `q` bracket the region of the term stack containing the terms of the expression belonging to this node, and the arguments, `tset` and `fset`, indicate which variables have already been set to true and false, respectively. It first searches for an unsatisfied term that is now either empty or a singleton.

During this scan each term is successively placed in `pterm` and `nterm`. Notice that a term is already satified if either `pterm&tset` or `nterm&fset` is non zero. If the term is unsatisfied, a simplified version of it is formed in `tposs` and `fposs` by removing from `pterm` and `nterm` all variables that have known settings. The code to do this is:

```
avail := ~(tset|fset)
tposs := pterm & avail // Remaining pos vars in this term.
fposs := nterm & avail // Remaining neg vars in this term.
```

The variables still active in the term are placed in `vars` by the assignment: `vars := tposs|fposs`. If `vars` is zero, an empty term has been found, indicating that the current expression is unsatisfiable. A variable can be removed from `vars` by the assignment: `vars &:= vars-1`. If after this, `vars` is zero then the term was a singleton and `tset` or `fset` updated appropriately. This process repeats until no empty or singleton terms are remain.

At this stage `count` holds the number of larger terms still to be satisfied. If more than a third of the current terms are satisfied, the data is compressed by filtering them out, by a call of `filter`. This ensures that, at all times, a high proportion of the terms under consideration are still active, whilst guaranteeing that the term stack will never needs to hold more than three times the number of terms in the original expression.

The program now counts how many times each variable is used in both doublets and larger terms in both the positive and negated forms. It does this using longitudinal arithmetic.

7.1 Longitudinal arithmetic

In longitudinal arithmetic, 32 counters are packed into the elements of a vector `p`, say. The element `p!0` holds the least significant bit of each counter, and the more significant bits are held in `p!1, p!2,...,p!n`. The function `inc` can be used to increment several of the counters simultaneously. Its definition is as follows:

```
FUN inc : p, w => WHILE w DO { !p, w := !p XOR w, !p & w; p+++ }
```

Here, `w` is a bit pattern indicating which counters are to be incremented. The replacement for the least significant word is therefore: `!p XOR w` and the carry bits are: `!p & w`. The while loop ensures that the carry is propagated as far as necessary. It is easy to show that, if only one counter is being incremented, then the body of the while loop is executed twice per call on average. If two or more counters are simultaneous incremented this average is only slightly larger.

The function `val` can be used to convert a longitudinal counter to a conventional integer. It definition is as follows:

```
FUN val : p, n, bit => LET res = 0
                       UNTIL n<0 TEST bit & p!n--
                                 THEN res := 2*res + 1
                                 ELSE res := 2*res
                       RETURN res
```

The arguments `p` and `n` give the vector and its upper bound, and `bit` identifies which counter is to be converted. The result is accumulated, from the most significant end, in `res`.

In this application, four vectors, all with upperbound 16, are used to hold counters. They are:

- `p2` – to hold the counts of positive variables occurring in doublets,
- `n2` – to hold the counts of negated variables occurring in doublets,
- `p3` – to hold the counts of positive variables occurring in larger terms, and
- `n3` – to hold the counts of negated variables occurring in larger terms.

They are incremented appropriately for each term by the code:

```
TEST bits(tposs|fposs) = 2 // Is the term a doublet or larger?
THEN { inc(p2, tposs); inc(n2, fposs) } // A doublet
ELSE { inc(p3, tposs); inc(n3, fposs) } // A larger term
```

The program now searches for a suitable variable to select by first looking at the doublet counters:

```
LET pv=p2, nv=n2, k=16
UNTIL pv!k OR nv!k OR k<0 DO k--   // Search the doublet counts
```

and if no doublet variables are found, it looks through the counters for larger terms:

```
IF k<0 DO { pv, nv, k := p3, n3, 16
            UNTIL pv!k OR nv!k DO k-- // Search the larger terms
          }
```

At this stage, one or both of `pv!k` or `nv!k` will be non zero, indicating which variable have the larger counts. Variables with non maximal counts are filtered out by:

```
LET pbits=pv!k, nbits=nv!k
UNTIL --k<0 DO { LET pw=pbits & pv!k, nw=nbits & nv!k
                 IF pw|nw DO pbits, nbits := pw, nw
               }
```

Variables with maximal counts are left in `pbits` for positive occurrences and `nbits` for negative ones. If any positive variables has a maximal count, one is chosen (`pbits & -pbits`) and `try` called twice, first setting this variable true and then to false. Similar code is used when a negated variable is chosen.

7.2 Comment

Although this implementation is limited to expressions with no more than 32 distinct variables, it can easily be extended to deal with more. It should, however, be noted that the algorithm can be applied as soon as the expression under consideration becomes simple enough. If this strategy is adopted, it is probably worth using a 64 bit implementation, provided the available hardware is suitable.

7.3 The program

```
GET "mcpl.h"

STATIC
  stackv, stackp, stackt, varcount,
  p2 = VEC 16, n2 = VEC 16, p3 = VEC 16, n3 = VEC 16

FUN bits : 0 => 0
         : w => 1 + bits( w&(w-1))

FUN start : =>
  LET filename = "cnfdata"
  LET argv = VEC 50

  UNLESS rdargs("DATA,TO/K", argv,50) DO
  { writef "Bad arguments for SAT\n"
    RETURN 20
  }

  IF argv!0 DO filename := argv!0

  stackv := getvec 500000
  stackp := stackv
  stackt := @stackv!500000

  IF argv!1 DO selectoutput(findoutput(argv!1))

  IF readterms filename DO
  { writef("Solving SAT problem: %s\n", filename)
    writef("It has %d variables and %d terms\n\n",
                   varcount,        (stackp-stackv)/(2*Bpw))
    try(stackv, stackp, 0, 0)
  }
  IF argv!1 DO endwrite()
  freevec stackv
  RETURN 0
```

```
// readterms reads a file representing a cnf expression.
// Typical data is as follows:
//        (1 -2) (1 2 3) (-1 -2 3)
// It return TRUE if successful.

FUN readterms : filename =>
  LET stdin=input(), data=findinput filename
  LET ch=0,          all=0

  IF data=0 DO { writef("Can't find file: %s\n", filename)
                 RETURN FALSE
               }

  selectinput data

  { // Skip to start of next term (if any).
    ch := rdch() REPEATUNTIL ch='(' OR ch=Endstreamch

    IF ch=Endstreamch DO { endread()
                           selectinput stdin
                           varcount := bits all
                           RETURN TRUE
                         }

    LET pterm=0, nterm=0

    { LET var = readn()      // Read a variable.
      MATCH var
      :      0    => BREAK   // No more variables in this term.
      :   1 .. 32 => pterm |:= 1<<( var-1)
      : -32 .. -1 => nterm |:= 1<<(-var-1)
      :      ?    => writef("Var %4d out of range\n", var)
    } REPEAT

    // Test the term for validity.
    UNLESS pterm|nterm DO writef "An empty term found\n"
    IF     pterm&nterm DO writef "A tautologous term found\n"

    all |:= pterm|nterm

    !stackp+++ := pterm // Insert into the term stack
    !stackp+++ := nterm
  } REPEAT

FUN filter : p, q, tset, fset =>
  UNTIL p>=q DO
  { LET pterm = !p+++  // Get a term
    LET nterm = !p+++

    // If it is unsatisfied push it onto the term stack.

    UNLESS pterm&tset OR nterm&fset DO { !stackp+++ := pterm
                                         !stackp+++ := nterm
                                       }
  }
```

```
FUN try : p, q, tset, fset =>

// p       points to the first term
// q       points to just beyond the last
// tset    variables currently set true
// fset    variables currently set false

  LET t, tcount, count
  LET pterm, nterm, avail, tposs, fposs

  // Scan for empty or singleton terms
  { t, tcount, count := p, 0, 0
    UNTIL t>=q DO
    { pterm := !t+++
      nterm := !t+++

      IF pterm&tset OR nterm&fset LOOP // Term already satisfied.

      avail := ~(tset|fset)
      tposs := pterm & avail // Remaining pos vars in this term.
      fposs := nterm & avail // Remaining neg vars in this term.

      LET vars = tposs|fposs
      IF vars=0 RETURN   // An empty term can't be satified.
      vars &:= vars-1    // Remove one variable.
      TEST vars=0 THEN { tcount++      // A singleton term found.
                         tset |:= tposs
                         fset |:= fposs
                       }
                  ELSE count++         // A larger term found.
    }
  } REPEATWHILE tcount  // Repeat until no singletons found.

  UNLESS count DO { writef("Solution found:\n")
                    prterm(tset, fset)
                    newline()
                    RETURN
                  }

  LET s = stackp

  IF count < (q-p)/(3*Bpw) DO // Filter if less than 2/3 remain.
  { filter(p, q, tset, fset)
    p, q := s, stackp
  }

  FOR n = 0 TO 16 DO p2!n, n2!n, p3!n, n3!n ALL:= 0

  t := p
  // Scan for doublet or larger terms
  UNTIL t>=q DO
  { pterm := !t+++
    nterm := !t+++

    IF pterm&tset OR nterm&fset LOOP // Term already satisfied

    avail := ~(tset|fset)
    tposs := pterm & avail // remaining pos vars in this term
    fposs := nterm & avail // remaining neg vars in this term

    TEST bits(tposs|fposs) = 2 // Is the term a doublet or larger?
    THEN { inc(p2, tposs); inc(n2, fposs) } // A doublet
    ELSE { inc(p3, tposs); inc(n3, fposs) } // A larger term
  }

  LET pv=p2, nv=n2, k=16
  UNTIL pv!k OR nv!k OR k<0 DO k--      // Search the doublet counts

  IF k<0 DO { pv, nv, k := p3, n3, 16
              UNTIL pv!k OR nv!k DO k-- // Search the larger terms
            }
  // pv!k ~= 0 or nv!k ~= 0 or both.

  // Find variable(s) with maximal count (at least one exists).
  LET pbit, nbit = pv!k, nv!k
  UNTIL --k<0 DO { LET pw=pbit & pv!k, nw=nbit & nv!k
                   IF pw|nw DO pbit, nbit := pw, nw
                 }
  TEST pbit
  THEN { pbit &:= -pbit              // Choose just one variable
         try(p, q, tset+pbit, fset) // Try setting it to set true
         try(p, q, tset, fset+pbit) // Try setting it to set false
       }
  ELSE { nbit &:= -nbit              // Choose just one variable
         try(p, q, tset, fset+nbit) // Try setting it to set false
         try(p, q, tset+nbit, fset) // Try setting it to set true
       }
  stackp := s

FUN prterm : tset, fset => // Print the setting, eg:
  LET i = 0                // 2 -3 5 6 -11
  WHILE tset|fset DO { i++
                       IF tset&1 DO writef(" %d", i)
                       IF fset&1 DO writef(" %d", -i)
                       tset, fset >>:= 1, 1
                     }

FUN inc : p, w => WHILE w DO { !p, w := !p XOR w, !p & w; p+++ }

FUN val : p, n, bit => LET res = 0
                       UNTIL n<0 TEST bit & p!n--
                                 THEN res := 2*res + 1
                                 ELSE res := 2*res
                       RETURN res
```

8 Summary of Bit Pattern Techniques Used

This section highlights the main bit pattern techniques used in this report.

8.1 `poss&-poss`

This selects of one element from a set. See pages 3, 7, 50, 59 and 66. Notice that the assignment: `pbit &:= -pbit` uses this mechanism.

8.2 `bits&(bits-1)`

This removes of one element from a set. See page 38. Notice that the assignment: `bits &:= bits-1` uses this mechanism.

8.3 `(pos<<1|pos>>1)&All`

With a carefully chosen representation of the triangular solitaire board, this expression reflects the left and right halves of the board. See pages 10.

8.4 `brd&hhp`

Using two separate bits to represent each peg position in solitaire allows a cheap test (`brd&hhp`) to determine whether a move is legal. See page 6.

8.5 `(fnv!bit) brd`

This expression provides a quick way of calling a function that depends on which bit occurs in `bit`. It is used on page 7. It is efficient provided the vector `fnv` is not too large. Other possible solutions include the use of a MATCH statement or, more subtly, the use of a perfect hashing function as in: `((fnv!(bit MOD Fnvsize))(...)`, but `Fnvsize` must be carefully chosen! As an aside, the well known birthday problem (23 people typically do not have distinct birth dates) leads one to believe that a perfect hash function is likely to require a wastefully sparse hash table. However, the function `hash` defined as follows:

```
FUN hash : bit => (bit>>1 MOD 29)
```

is a perfect hash function for the 29 values: `1<<0, ..., 1<<28`. It might be called perfectly perfect since all 29 entries in the hash table are used. An analogous result holds when the divisor is 19. A few other good values suitable for wordlengths upto 64 bits are 2, 4, 5, 9, 11, 13, 19, 25, 29, 37, 53, 59, 61 and 67. These work almost well if the right shift is omitted from the hash function. Unfortunately, on modern machines `MOD` is relatively expensive. A faster hashing function could be based on a hardware implementation of some variant of $\lfloor \log_2 w \rfloor$.

8.6 Flipping a 32×32 bit map

The function `flipbits`, described on page 55, flips a bit map about a diagonal.

8.7 `reflect` and `rotate`

These functions, described on section 4.8, reflect and rotate an 8×8 bitmap used in the two player pentomino game.

8.8 Compacting a sparse bit patterns

This was done by the function `cmpt64` to compact a 76 bit pattern to 64 bits, and by `cmpt32` to compact 64 bits to 32 bits. Both these functions are described on page 37.

8.9 Longitudinal arithmetic

The functions `inc` and `val`, described on page 65, illustrate the use of longitudinal arithmetic.

A Summary of MCPL

In the syntactic forms given below

E	denotes an expression,
K	denotes a constant expression,
A	denotes an argument list,
P	denotes a pattern,
N	denotes a variable name,
M	denotes a manifest name.

A.1 Outermost level declarations

These are the only constructs that can occur at the outermost level of the program.

`MODULE` N

This directive must occur first, if present.

`GET` *string*

Insert the file named *string* at this point.

`FUN` N `:` P `=>` E `:..:` P `=>` E `.`

The main procedure has name: `start`. Functions may only be defined at the outermost level, hence they have no dynamic free variables.

`EXTERNAL` N `:` *string* `,..,` N `:` *string*

The "`:` *string*"s may be omitted.

`MANIFEST` M `=` K `,..,` M `=` K

The "`=` K"s are optional. When omitted the next available integer is used.

`STATIC` N `=` K `,..,` N `=` K

The "`=` K"s are optional, and when omitted the corresponding variable is not initialised. The Ks may include strings, tables and functions.

`GLOBAL` N `:` K `,..,` N `:` K

The "`:` K"s may be omitted, and when omitted the next available integer is used.

A.2 Expressions

N Eg: `abc v1 a s_err`

These are used for variable and function names. They start with lower case letters.

M Eg: `Abc B1 A S_for`

These are used for manifest constant names. They start with upper case letters.

inumb Eg: `1234 #x7F_0001 #377 #b_0111_1111_0000`

`?`

This yields an undefined value.

`TRUE` `FALSE`

These are constants equal to `-1` and `0`, respectively.

char Eg: `'A' '\n' 'XYZ'`

The characters are packed into a word as consecutive bytes. The rightmost character being placed in the least significant byte position. Such constants can be thought of as base 256 integers.

string Eg: `"abc" "Hello\n"`

Strings are zero terminated for compatibility with C.

`TABLE [` *E* `,..,` *E* `]`

This yields an initialised static vector. The elements of the vector are not necessarily re-initialised on each evaluation of the table, particularly if the initial values are constants.

`[` *E* `,..,` *E* `]`

This yields an initialised local vector. The space allocated in current scope.

`VEC` *K*

This yields an uninitialised local word vector with subscripts from 0 to *K*. The space is allocated on entry to the current scope.

`CVEC` *K*

This yields an uninitialised local byte vector with subscripts from 0 to *K*. The space is allocated on entry to the current scope.

`(` *E* `)`

Parentheses are used to group an expression that normally yields a result.

`{` *E* `}`

Braces are used to group an expression that normally has no result.

EA

This is a function call. To avoid syntactic ambiguity, *A* must be a name (*N* or *M*), a constant (*inumb*, `?`, `TRUE`, `FALSE`, *char* or *string*), or it must start with `(` or `[`.

`@` *E*

This returns the address of *E*. *E* must be either a variable name (*N*) or a subscripted expression (*E*`!`*E*, *E*`%`*E*, `!`*E* or `%`*E*).

E `!` E `!` E

This is the word subscription operator. The left operand is a pointer to the zeroth element of a word vector and the right hand operand is an integer subscript. The form `!`E is equivalent to E`!0`.

E `%` E `%` E

This is the byte subscription operator. The left operand is a pointer to the zeroth element of a byte vector and the right hand operand is an integer subscript. The form `%`E is equivalent to E`%0`.

`++` E `+++` E `--` E `---` E

Pre increment or decrement by 1 or `Bpw` (bytes per word).

E `++` E `+++` E `--` E `---`

Post increment or decrement by 1 or `Bpw`.

`~` E `+` E `-` E `ABS` E

These are monadic operators for bitwise NOT, plus, minus and absolute value, respectively.

E `<<` E E `>>` E

These are logical left and right shift operators, respectively.

E `*` E E `/` E E `MOD` E E `&` E

These are dyadic operators for multplication, division, remainder after division, and bitwise AND, respectively.

E `+` E E `-` E E `|` E

These are dyadic operators for addition, subtraction, and bitwise OR, respectively.

E `XOR` X

This returns the bitwise exclusive OR of its operands.

E *relop* E *relop* ... E

where *relop* is any of `=`, `~=`, `<`, `<=`, `>` or `>=`. It return `TRUE` only if all the individual relations are satisfied. Each E is evaluated atmost once.

`NOT` E E `AND` E E `OR` E

These are the truth value operators.

E `->` E, E

This is the conditional expression construct.

E , . . , E `:=` E , . . , E E , . . , E `ALL:=` E

This is the simultaneous assignment operator. All the expressions are evaluated then all the assignments done.

E `,..,` *E* *op*`:=` *E* `,..,` *E*

Where *op*`:=` can be any of the following: `>>:=`, `<<:=`, `*:=`, `/:=`, `MOD:=`, `&:=`, `+:=`, `-:=`, or `XOR:=`.

`RAISE` *A*

This transfers control to the the currently active HANDLE. Up to three arguments can be passed.

`TEST` *E* `THEN` *E* `ELSE` *E*

`IF` *E* `DO` *E*

`UNLESS` *E* `DO` *E*

These are the conditional commands. They are less binding than assignment and typically do not yield results.

`WHILE` *E* `DO` *E*

`UNTIL` *E* `DO` *E*

E `REPEATWHILE` *E*

E `REPEATUNTIL` *E*

E `REPEAT`

`FOR` *N* `=` *E* `TO` *E* `BY` *K* `DO` *E*

`FOR` *N* `=` *E* `TO` *E* `DO` *E*

`FOR` *N* `=` *E* `BY` *K* `DO` *E*

`FOR` *N* `=` *E* `DO` *E*

These are the repetitive commands. The FOR command introduces a new scope for locals, and *N* is a new variable within this scope.

`VALOF` *E*

This introduces a new scope for locals and defines the context for RESULT commands within *E*.

`MATCH` *A* `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

`EVERY` *A* `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

E `HANDLE` `:` *P* `=>` *E* `:..:` *P* `=>` *E* `.`

In each of these construct, the dot (`.`) is optional. The arguments (*A*) are matched against the patterns (*P*), and control passed to the first expression whose patterns match. For the EVERY construct, all guarded expressions whose patterns match are evaluated. The HANDLE construct defines the context for RAISE commands. A RAISE command will supply the arguments to be matched by HANDLE.

`RESULT` *E*   `RESULT`

Return from current `VALOF` expression with a value. `RESULT` with no argument is equivalent to `RESULT ?`.

`EXIT` E `EXIT`

Return from the current function or MATCH, EVERY or HANDLE construct with a given value. `EXIT` with no argument is equivalent to `EXIT ?`.

`RETURN` E `RETURN`

Return from current function with a value. `RETURN` with no argument is equivalent to `RETURN ?`.

`BREAK` `LOOP`

Respectively, exit from, or loop in the current repetitive expression.

E `;..;` E

Evaluate expressions from left to right. The result is the value of the last expression. Any semicolon at the end of a line is optional.

`LET` N `=` E `,..,` N `=` E

This construct declares and possibly initialises some new local variables. The allocation of space for them is done on entry to the current scope. New local scopes are introduced by `FUN`, `MATCH`, `EVERY`, `HANDLE`, `=>`, `VALOF`, and `FOR`. The "`=`E"s are optional, but, if present, cause the corresponding variable to be initialised when the LET contruct is reached.

A.3 Constant expressions

These are used in MANIFEST, STATIC and GLOBAL declarations, in VEC expressions, in the step length of FOR commands, and in patterns.

The syntax of constant expressions is the same as that of ordinary expressions except that only constructs that can be evaluated at compile time are permitted. These are:

M, *inumb*, `?`, `TRUE`, `FALSE`, *char*,
`(` K `)`, `{` K `}`,
`~` K, `+` K, `-` K, `ABS` K,
K `<<` K, K `>>` K,
K `*` K, K `/` K, K `MOD` K, K `&` K,
K `+` K, K `-` K, K `|` K,
K `XOR` K,
K *relop* K *relop* ... K,
`NOT` K, K `AND` K, K `OR` K,
K `->` K, K
`TEST` K `THEN` K `ELSE` K

A.4 Patterns

Patterns are used in function definitions, MATCH, EVERY and HANDLE constructs. Patterns are matched against parameter values held in consecutive storage locations. Pattern matching is applied from left to right, except that any assignments are done at the end and only if the entire match was successful.

N

The current location is given name *N*.

`?`

This will always match the current location.

K

The value in the current location must equal *K*.

K`..`*K*

The value in the current location must greater than or equal to the first *K* and less than or equal to the second *K*.

`(` *P* `)`

Parentheses are used for grouping.

P `,..,` *P*

The current location and adjacent ones are matched by the corresponding *P*s.

`[` *P* `,..,` *P* `]`

The value of the current location is a pointer to consective locations matched by the *P*s.

PP

The value in the current location is matched by both *P*s.

P `OR` *P*

One or other pattern must match. The patterns must only be constants (*K*) or ranges (*K*`..`*K*).

`<` *E* `<=` *E* `>` *E* `>=` *E* `=` *E* `~=` *E*

The value of the current location must be `<`*E*, `<=`*E*, etc.

`:=` *E*

If the entire match is successful, the current location is updated with the value of *E*.

op`:=` *E*

If the entire match is successful, the current location is modified by the specified operation with *E*.

A.5 Arguments

Arguments are used in function calls and in MATCH, EVERY, GOTO and RAISE commands. They cause a number of expressions to be evaluated and placed in consecutive locations ready to be matched by one or more patterns.

E
`( )`
`(` *E* `, . . . ,` *E* `)`

An argument list is either an expression, or a, possibly empty, list of expressions separated by commas and enclosed in parentheses. The argument in a function call must start with `(` or `[`, or be a name, a constant, `?`, `TRUE` or `FALSE`.

References

[CM81] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, Berlin, 1981.

[DP60] Martin Davis and Hilary Putman. A computing procedure for quatification theory. *Journal of the Association for Computing Machinery*, 7(3):201–215, July 1960.

[Fle65] John G. Fletcher. A program to solve the pentomino problem by the recursive use of macros. *Communications of the ACM*, 8(10):621–623, October 1965.

[Haz82] P. Hazel. Private communication. Cambridge University Computer Laboratory, 1982.

[Moo82] J.K.M. Moody. Private communication. Cambridge University Computer Laboratory, 1982.

[Orm96] H.K. Orman. Pentominoes: A first player win. In R.J.Nowakowski, editor, *Games of No Chance.* Cambridge University Press, 1996.

[Pau91] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, 1991.

[Ric] M. Richards. *My WWW Home Page.* www.cl.cam.ac.uk/users/mr/.

[Ric97] M. Richards. *MCPL Programming Manual.* Technical Report No 431, Cambridge University Computer Laboratory, July 1997.

[RWS80] M. Richards and C. Whitby-Strevens. *BCPL - the language and its compiler.* Cambridge University Press, Cambridge, 1980.

[Sto77] J. Stoy. *Denotational Semantics: The Scott-Stratchey Approach to Programming Language Theory*, pages 113–115. MIT Press, Cambridge Mass., 1977.

[Wir71] N. Wirth. Program development by stepwise refinement. *Comm. ACM*, 14:221–227, 1971.