(See https://cs.stanford.edu/~knuth/programs.html for date.)

1. Introduction. This program finds all ways to pack $2 \times 2 \times 1$ bricks into a $4 \times 4 \times 4$ box in such a way that each face of each brick touches the boundary of the box or the face of another brick. The program is also designed to be readily modified so that it applies to other sorts of pieces in other sorts of boxes.

I'm writing it primarily to gain further experience of the technique of "dancing links," which worked so nicely in the XCOVER routine. Also I'm having fun today; I just finished a long, boring task and I'm rewarding myself by taking time off from other duties.

```
#define n1 4    /* one box dimension */
#define n2 4    /* another */
#define n3 4    /* the last */
#define verbose (argc > 1)
#define very_verbose (argc > 2)
#define very_very_verbose (argc > 3)
#include <stdio.h>
  ⟨ Type definitions 2 ⟩
  ⟨ Global variables 3* ⟩
  tmp()
  {
    printf("tmp");
  }
  main(argc, argv)
      int argc;
      char *argv[];    /* the usual command-line parameters */
  {
    register node *p, *q, *r;
    register int stamp = 0;
    ⟨ Initialize the data structures 4 ⟩;
    ⟨ Backtrack thru all possibilities 11 ⟩;
    ⟨ Report the answers 26 ⟩;
  }
```

2. Data structures. This program deals chiefly with three kinds of lists, representing cells, moves, and constraints.

A move list is a circular list of nodes, one for each cell occupied by a particular placement of a piece. The nodes are doubly linked by *left* and *right* pointers, which stay fixed throughout the algorithm.

A cell list is a circular list consisting of a header node and one additional node for each move that occupies this cell. These nodes are doubly linked by *up* and *down* pointers; thus each node in a move list is also a potential member of a cell list. Nodes leave a cell list when they belong to a move that conflicts with other moves already made. A header node is recognizable by the fact that its *left* pointer is null.

A constraint is a sequence of pointers to cell headers, followed by a null pointer. It represents a set of cells that should not all be empty, based on moves made so far. A constraint list is a sequence of pointers to constraints, followed by a null pointer.

Nodes have a *tag* field that is used in a special "stamping" trick explained later. This field points to an integer; its basic property is that two nodes have the same *tag* if and only if they are part of the same move.

⟨ Type definitions 2 ⟩ ≡

```
typedef struct node_struct {
  struct node_struct *left, *right;  /* adjacent nodes of a move */
  struct node_struct *up, *down;  /* adjacent nodes of a cell */
  char *name;  /* identification of this node for diagnostic printouts */
  struct node_struct ***clist;  /* list of constraint lists for this move */
  int *tag;  /* unique identification of a move */
} node;
```

This code is used in section 1.

3* The sizes of the basic arrays were determined experimentally; originally I just set them to a large number and ran the program.

⟨ Global variables 3* ⟩ ≡

```
node headers[n1][n2][n3];  /* cell header nodes */
node nodes[432];  /* nodes in the move lists */
node *constraints[1674];  /* elements of constraints */
node *special_constraints[2];  /* we'll use this at level 0 */
node **clists[558];  /* elements of constraint lists */
char names[n1 * n2 * n3 * 4];  /* cell names */
int tags[108];  /* the tag fields point into this array */
```

See also section 10.

This code is used in section 1.

4. Here's how we get everything started, when packing bricks as mentioned above.

⟨ Initialize the data structures 4 ⟩ ≡

```
{
  register node *cur_node = &nodes[0], **cur_con = &constraints[0], ***cur_clist = &clists[0];
  register char *cur_name = &names[0];
  register int *cur_tag = &tags[0];
  register int i, j, k;
  ⟨ Make all cell lists empty 5 ⟩;
  ⟨ Initialize all moves that have constant first coordinate 6* ⟩;
  ⟨ Initialize all moves that have constant second coordinate 7* ⟩;
  ⟨ Initialize all moves that have constant third coordinate 8* ⟩;
  printf("This problem involves %d namechars, %d moves, %d nodes,\n",
      (cur_name - &names[0])/4, cur_tag - &tags[0], cur_node - &nodes[0]);
  printf(" %d constraint elements, %d clist elements.\n", cur_con - &constraints[0],
      cur_clist - &clists[0]);
}
```

This code is used in section 1.

5. ⟨ Make all cell lists empty 5 ⟩ ≡

```
for (i = 0; i < n1; i++)
  for (j = 0; j < n2; j++)
    for (k = 0; k < n3; k++) {
      *cur_name = i + '0';
      *(cur_name + 1) = j + '0';
      *(cur_name + 2) = k + '0';
      headers[i][j][k].name = cur_name;
      cur_name += 4;
      headers[i][j][k].up = headers[i][j][k].down = &headers[i][j][k];
    }
```

This code is used in section 4.

6* **#define** *new_node*(*ii*, *jj*, *kk*)
```
{
  cur_node→right = cur_node + 1; cur_node→left = cur_node − 1;
  p = &headers[ii][jj][kk]; q = p→down;
  cur_node→name = p→name;
  cur_node→up = p; cur_node→down = q; p→down = cur_node; q→up = cur_node;
  cur_node→tag = cur_tag;
  cur_node→clist = cur_clist;
  cur_node++;
}
```
#define *start_con* ∗*cur_clist* = *cur_con* /∗ begin making a constraint list ∗/
#define *new_con*(*ii*, *jj*, *kk*) ∗*cur_con*++ = &*headers*[*ii*][*jj*][*kk*] /∗ add a cell to it ∗/
#define *wrap_con* *cur_con*++, *cur_clist*++ /∗ finish making a constraint list ∗/

⟨ Initialize all moves that have constant first coordinate 6* ⟩ ≡
```
for (i = 0; i < n1; i++)
  for (j = 0; j + 1 < n2; j++)
    for (k = 0; k + 1 < n3; k++) {
      register node *first_node = cur_node;
      if ((i ≡ 0 ∨ i + 1 ≡ n1) ∧ (j ≡ 0 ∨ j + 2 ≡ n2) ∧ (k ≡ 0 ∨ k + 2 ≡ n3)) continue;
      new_node(i, j, k);
      new_node(i, j, k + 1);
      new_node(i, j + 1, k);
      new_node(i, j + 1, k + 1);
      first_node→left = cur_node − 1;
      (cur_node − 1)→right = first_node;
      if (i > 0) {
        start_con;
        new_con(i − 1, j, k);
        new_con(i − 1, j, k + 1);
        new_con(i − 1, j + 1, k);
        new_con(i − 1, j + 1, k + 1);
        wrap_con;
      }
      if (i + 1 < n1) {
        start_con;
        new_con(i + 1, j, k);
        new_con(i + 1, j, k + 1);
        new_con(i + 1, j + 1, k);
        new_con(i + 1, j + 1, k + 1);
        wrap_con;
      }
      if (j > 0) {
        start_con;
        new_con(i, j − 1, k);
        new_con(i, j − 1, k + 1);
        wrap_con;
      }
      if (j + 2 < n2) {
        start_con;
        new_con(i, j + 2, k);
        new_con(i, j + 2, k + 1);
        wrap_con;
```

```
    }
    if (k > 0) {
      start_con;
      new_con(i, j, k − 1);
      new_con(i, j + 1, k − 1);
      wrap_con;
    }
    if (k + 2 < n3) {
      start_con;
      new_con(i, j, k + 2);
      new_con(i, j + 1, k + 2);
      wrap_con;
    }
    cur_clist++;
    cur_tag++;
    if (very_very_verbose) ⟨Print the move that starts with first_node 9⟩;
  }
```

This code is used in section 4.

7*. ⟨Initialize all moves that have constant second coordinate 7*⟩ ≡

```
for (i = 0; i + 1 < n1; i++)
  for (j = 0; j < n2; j++)
    for (k = 0; k + 1 < n3; k++) {
      register node *first_node = cur_node;
      if ((i ≡ 0 ∨ i + 2 ≡ n1) ∧ (j ≡ 0 ∨ j + 1 ≡ n2) ∧ (k ≡ 0 ∨ k + 2 ≡ n3)) continue;
      new_node(i, j, k);
      new_node(i, j, k + 1);
      new_node(i + 1, j, k);
      new_node(i + 1, j, k + 1);
      first_node→left = cur_node − 1;
      (cur_node − 1)→right = first_node;
      if (j > 0) {
        start_con;
        new_con(i, j − 1, k);
        new_con(i, j − 1, k + 1);
        new_con(i + 1, j − 1, k);
        new_con(i + 1, j − 1, k + 1);
        wrap_con;
      }
      if (j + 1 < n2) {
        start_con;
        new_con(i, j + 1, k);
        new_con(i, j + 1, k + 1);
        new_con(i + 1, j + 1, k);
        new_con(i + 1, j + 1, k + 1);
        wrap_con;
      }
      if (i > 0) {
        start_con;
        new_con(i − 1, j, k);
        new_con(i − 1, j, k + 1);
        wrap_con;
      }
      if (i + 2 < n1) {
        start_con;
        new_con(i + 2, j, k);
        new_con(i + 2, j, k + 1);
        wrap_con;
      }
      if (k > 0) {
        start_con;
        new_con(i, j, k − 1);
        new_con(i + 1, j, k − 1);
        wrap_con;
      }
      if (k + 2 < n3) {
        start_con;
        new_con(i, j, k + 2);
        new_con(i + 1, j, k + 2);
        wrap_con;
      }
```

cur_clist ++;
cur_tag ++;
if ($very\_very\_verbose$) ⟨ Print the move that starts with $first\_node$ 9 ⟩;
}

This code is used in section 4.

8* ⟨Initialize all moves that have constant third coordinate 8*⟩ ≡

for $(i = 0;\ i + 1 < n1;\ i{+}{+})$
  for $(j = 0;\ j + 1 < n2;\ j{+}{+})$
    for $(k = 0;\ k < n3;\ k{+}{+})$ {
      register node $*first\_node = cur\_node$;
      if $((i \equiv 0 \lor i + 2 \equiv n1) \land (j \equiv 0 \lor j + 2 \equiv n2) \land (k \equiv 0 \lor k + 1 \equiv n3))$ **continue**;
      $new\_node(i, j, k)$;
      $new\_node(i + 1, j, k)$;
      $new\_node(i, j + 1, k)$;
      $new\_node(i + 1, j + 1, k)$;
      $first\_node\to left = cur\_node - 1$;
      $(cur\_node - 1)\to right = first\_node$;
      if $(k > 0)$ {
        $start\_con$;
        $new\_con(i, j, k - 1)$;
        $new\_con(i + 1, j, k - 1)$;
        $new\_con(i, j + 1, k - 1)$;
        $new\_con(i + 1, j + 1, k - 1)$;
        $wrap\_con$;
      }
      if $(k + 1 < n3)$ {
        $start\_con$;
        $new\_con(i, j, k + 1)$;
        $new\_con(i + 1, j, k + 1)$;
        $new\_con(i, j + 1, k + 1)$;
        $new\_con(i + 1, j + 1, k + 1)$;
        $wrap\_con$;
      }
      if $(j > 0)$ {
        $start\_con$;
        $new\_con(i, j - 1, k)$;
        $new\_con(i + 1, j - 1, k)$;
        $wrap\_con$;
      }
      if $(j + 2 < n2)$ {
        $start\_con$;
        $new\_con(i, j + 2, k)$;
        $new\_con(i + 1, j + 2, k)$;
        $wrap\_con$;
      }
      if $(i > 0)$ {
        $start\_con$;
        $new\_con(i - 1, j, k)$;
        $new\_con(i - 1, j + 1, k)$;
        $wrap\_con$;
      }
      if $(i + 2 < n1)$ {
        $start\_con$;
        $new\_con(i + 2, j, k)$;
        $new\_con(i + 2, j + 1, k)$;
        $wrap\_con$;
      }

```
      cur_clist++;
      cur_tag++;
      if (very_very_verbose) ⟨Print the move that starts with first_node 9⟩;
    }
```

This code is used in section 4.

9. ⟨Print the move that starts with *first_node* 9⟩ ≡

```
{
  node **p1, ***c1;
  for (p = first_node; ; p = p→right) {
    printf("%s␣", p→name);
    if (p→right ≡ first_node) break;
  }
  printf("=>");
  for (c1 = p→clist; *c1; c1++) {
    for (p1 = *c1; *p1; p1++) printf("%s,", (*p1)→name);
    printf("␣");
  }
  printf("\n");
}
```

This code is used in sections 6*, 7*, and 8*.

10. Backtracking. At level l, we've made l moves, and we assume that we've got to satisfy constraints c for $constr[l] \le c < ctop$. We decide which of those constraints is strongest, in the sense that it a minimal number of moves will satisfy it; we record those moves in an array of pointers m to move nodes, for $first[l] \le m < mtop$, and we try each of them in turn.

```
#define move_stack_size 1000
#define constr_stack_size 1000
#define max_level (((n1 * n2 * n3) >> 2) - 2)
```

⟨ Global variables 3* ⟩ +≡

```
  node *move_stack[move_stack_size];
  node **constr_stack[constr_stack_size];
  node **first[max_level];    /* beginning move on a given level */
  node **move[max_level];    /* current move being explored */
  node ***constr[max_level];    /* first constraint on a given level */
  int totsols[max_level];    /* the number of solutions we found */
```

11. I'm using **goto** statements, as usual when I backtrack.

⟨ Backtrack thru all possibilities 11 ⟩ ≡

```
{
  register node **mtop = &move_stack[0];
  register node ***ctop = &constr_stack[0];
  register node **pp, ***cc;
  register int l = 0;

  constr[0] = ctop;
  ⟨ Put the initial constraints onto the constraint stack 15* ⟩;
newlevel: first[l] = mtop;
  if (constr[l] ≡ ctop) {
    ⟨ Record a solution 25 ⟩;
    if (l ≡ max_level − 1) goto backtrack;
    ⟨ Put all remaining moves on the move stack 23 ⟩;
  }
  else if (l ≡ max_level − 1) goto backtrack;
  else ⟨ Find a constraint to branch on, and put its moves on the move stack 12 ⟩;
  pp = first[l];
  goto advance;
backtrack: ⟨ Reinstate all moves from this level 22 ⟩;
  mtop = first[l];
  if (l ≡ 0) goto done;
  l−−;
  pp = move[l];
  ⟨ Unmake move *pp 19 ⟩;
  ⟨ Disallow move *pp 21 ⟩;
  pp++;
advance:
  if (pp ≡ mtop) goto backtrack;
  move[l] = pp;
  ⟨ Make move *pp 16 ⟩;
  if (very_verbose) ⟨ Print a progress report 24 ⟩;
  l++;
  goto newlevel;
done: ;
}
```

This code is used in section 1.

12. ⟨ Find a constraint to branch on, and put its moves on the move stack 12 ⟩ ≡

```
{
  register int count;
  node **cbest;
  int best_count = 100000;

  for (cc = constr[l]; cc < ctop; cc++) {
    ⟨ If constraint *cc has smaller count than best_count, set cbest = *cc 13 ⟩;
  }
  ⟨ Put the moves for cbest on the move stack 14 ⟩;
}
```

This code is used in section 11.

13. Here's where the tag fields become important. Pay attention now.

A constraint is a list of cells, at least one of which must be occupied by a future move. We find all ways to satisfy the constraint by going through all moves on those cell lists. But we don't want to count a move twice when it covers more than one cell on the list. So we put a time stamp in the *tag* field of each move, telling us whether we've already seen that move while processing the current constraint.

⟨If constraint $*cc$ has smaller count than *best_count*, set $cbest = *cc$ 13⟩ ≡

```
count = 0;
stamp++;
for (pp = *cc; *pp; pp++)
  for (p = (*pp)→down; p→left; p = p→down)
    if (*(p→tag) ≠ stamp) count++, *(p→tag) = stamp;
if (very_verbose) {
  printf("Constraint␣");
  for (pp = *cc; *pp; pp++) printf("%s,", (*pp)→name);
  printf("␣%d\n", count);
}
if (count < best_count) best_count = count, cbest = *cc;
```

This code is used in section 12.

14.
```
#define panic(s)
    {
      printf("s␣stack␣overflow!\n");
      exit(−1);
    }
```

⟨Put the moves for *cbest* on the move stack 14⟩ ≡

```
stamp++;
for (pp = cbest; *pp; pp++)
  for (p = (*pp)→down; p→left; p = p→down)
    if (*(p→tag) ≠ stamp) *mtop++ = p, *(p→tag) = stamp;
if (mtop ≥ &move_stack[move_stack_size]) panic(move);
```

This code is used in section 12.

15* In this variation, we have omitted all moves that occupy the corners. It's easy to see that it is then necessary to occupy at least one cell next to a corner. So I make that the initial constraint.

⟨Put the initial constraints onto the constraint stack 15*⟩ ≡

```
special_constraints[0] = &headers[0][0][1];
*ctop++ = &special_constraints[0];
```

This code is used in section 11.

16. This step changes *pp*, inside of section ⟨ If constraint $pp = *cc$ is not satisfied, put it on the constraint stack 18 ⟩. (I could have used another variable, but I'm from an older generation that tries to conserve the number of registers used. Silly of me.)

```
⟨ Make move *pp 16 ⟩ ≡
  if (stamp ≡ 1620) tmp( );
  for (p = *pp; ; p = p→right) {
    ⟨ Remove all other moves in the cell list containing p from their other cell lists 17 ⟩;
    if (p→right ≡ *pp) break;
  }
  constr[l + 1] = ctop;
  for (cc = constr[l]; cc < constr[l + 1]; cc++)
    ⟨ If constraint pp = *cc is not satisfied, put it on the constraint stack 18 ⟩;
  for (cc = p→clist; *cc; cc++) ⟨ If constraint pp = *cc is not satisfied, put it on the constraint stack 18 ⟩;
  if (ctop ≥ &constr_stack[constr_stack_size]) panic(constraint);
```

This code is used in section 11.

17. When a cell is occupied by the move at level l, we put $l + 1$ into the *right* field of its header node. That way we can tell if the cell is occupied.

The "dancing links" trick is used here: When node r is removed from its list, we don't change *r→up* and *r→down*, and we don't lose the links that led us to r. That means it will be easy to restore the list when backtracking.

```
⟨ Remove all other moves in the cell list containing p from their other cell lists 17 ⟩ ≡
  for (q = p→down; q ≠ p; q = q→down) {
    if (q→left ≡ Λ) q→right = (node *)(l + 1);
    else
      for (r = q→left; r ≠ q; r = r→left) {
        r→up→down = r→down;
        r→down→up = r→up;
      }
  }
```

This code is used in section 16.

18.
```
⟨ If constraint pp = *cc is not satisfied, put it on the constraint stack 18 ⟩ ≡
  {
    for (pp = *cc; *pp; pp++)
      if ((*pp)→right) break;
    if (¬*pp) *ctop++ = *cc;
  }
```

This code is cited in section 16.

This code is used in section 16.

19. The links do their dance in this step. We have to reconstruct the lists in exact reverse order of the way we constructed them. (That's why I provided both *left* and *right* links in the move lists. Otherwise the program would try to insert a node into its list twice.)

The significant aspect to note about dancing links in this algorithm is the order in which moves are disallowed and reinstated, as well as the order in which they are make and unmade.

```
⟨Unmake move *pp 19⟩ ≡
  for (p = (*pp)→left; ; p = p→left) {
    ⟨Unremove all other moves in the cell list containing p from their other cell lists 20⟩;
    if (p ≡ *pp) break;
  }
  ctop = constr[l + 1];
```

This code is used in section 11.

20. ⟨Unremove all other moves in the cell list containing p from their other cell lists 20⟩ ≡

```
  for (q = p→up; q ≠ p; q = q→up) {
    if (q→left ≡ Λ) q→right = Λ;
    else
      for (r = q→right; r ≠ q; r = r→right) {
        r→up→down = r;
        r→down→up = r;
      }
  }
```

This code is used in section 19.

21. ⟨Disallow move $*pp$ 21⟩ ≡

```
  for (p = (*pp)→right; ; p = p→right) {
    q = p→down;
    r = p→up;
    q→up = r;
    r→down = q;
    if (p ≡ *pp) break;
  }
```

This code is used in section 11.

22. ⟨Reinstate all moves from this level 22⟩ ≡

```
  for (pp = mtop − 1; pp ≥ first[l]; pp−−)
    for (p = (*pp)→right; ; p = p→right) {
      q = p→down;
      r = p→up;
      q→up = r→down = p;
      if (p ≡ *pp) break;
    }
```

This code is used in section 11.

23. ⟨ Put all remaining moves on the move stack 23 ⟩ ≡

```
{
  stamp++;
  for (p = &headers[0][0][0]; p < &headers[n1][0][0]; p++)
    if (¬p→right)
      for (q = p→down; q ≠ p; q = q→down)
        if (*(q→tag) ≠ stamp) *mtop++ = q, *(q→tag) = stamp;
}
```

This code is used in section 11.

24. ⟨ Print a progress report 24 ⟩ ≡

```
{
  printf("Move␣%d:", l + 1);
  for (p = (*move[l])→right; ; p = p→right) {
    printf("␣%s", p→name);
    if (p ≡ *move[l]) break;
  }
  printf("␣(%d)\n", stamp);
}
```

This code is used in section 11.

25. ⟨ Record a solution 25 ⟩ ≡

```
totsols[l]++;
if (verbose) {
  int ii, jj, kk;
  printf("%d.%d:", l, totsols[l]);
  for (ii = 0; ii < n1; ii++) {
    printf("␣");
    for (jj = 0; jj < n2; jj++)
      for (kk = 0; kk < n3; kk++) {
        register int c = (int) headers[ii][jj][kk].right;
        printf("%c", c > 9 ? c − 10 + 'a' : c + '0');
      }
  }
  printf("\n");
}
```

This code is used in section 11.

26. ⟨ Report the answers 26 ⟩ ≡

```
printf("Total␣solutions␣found:\n");
{
  register int lev;
  for (lev = 0; lev < max_level; lev++)
    if (totsols[lev]) printf("␣␣level␣%d,␣%d\n", lev, totsols[lev]);
}
```

This code is used in section 1.

27*. Index.

The following sections were changed by the change file: 3, 6, 7, 8, 15, 27.

ANTISLIDE-NOCORNER