January 9, 2024 at 15:00

1 NumPSLA, a program for enumerating pseudoline arrangements and abstract order types

The purpose of this program is to enumerate abstract order types (sometimes also called generalized configurations or pseudoconfigurations) and their duals, the pseudoline arrangements (PSLAs).

The program enumerates the objects without repetition and with negligible storage.

We consider the nondegenerate (*simple*) case only: no three points on a line, and no three curves through a point. We abbreviate *abstract order type* by AOT and *oriented abstract order type* by OAOT. (An *oriented* abstract order type can be distinguished from its mirror image.) As a baseline, we consider everything *oriented*, i.e., the mirror object can be isomorphic or not. In the end, we also check for mirror symmetry, and we can choose to report only one orientation of two mirror types.

1.1 Pseudoline arrangements and abstract order types

A *projective* pseudoline arrangement (PSLA) is a family of centrally symmetric closed Jordan curves on the sphere such that any two curves intersect in two points, and they intersect transversally at these points.

An *affine* PSLA is a family of Jordan curves in the plane that go to infinity at both ends and that intersect pairwise exactly once, and they intersect transversally at these points.

An *x-monotone* PSLA (*wiring diagram*, primitive sorting network) is an affine PSLA with x-monotone curves.

We consider two objects as equivalent under deformation by orientation-preserving isotopies of the sphere, or the plane, respectively. (An x-monotone PSLA must remain x-monotone throughout the deformation.)

A *marked* OAOT is an OAOT with a marked point on the convex hull.

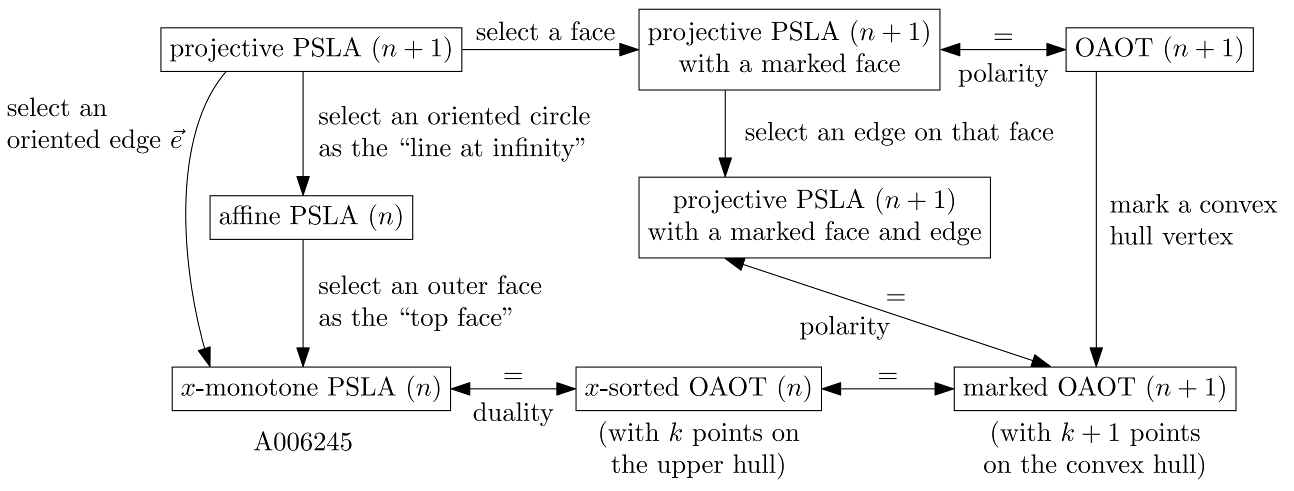

projective PSLA $(n+1)$
select a face
projective PSLA $(n+1)$ with a marked face
=
polarity
OAOT $(n+1)$
select an oriented edge $\vec{e}$
select an oriented circle as the "line at infinity"
select an edge on that face
affine PSLA (n)
projective PSLA $(n+1)$ with a marked face and edge
mark a convex hull vertex
select an outer face as the "top face"
=
polarity
x-monotone PSLA (n)
=
duality
x-sorted OAOT (n)
=
marked OAOT $(n+1)$
A006245
(with k points on the upper hull)
(with $k+1$ points on the convex hull)

Figure 1: Relations between different concepts. There are different paths from the top left to the bottom right, which apply specialization or geometric reinterpretation in different order.

See Aichholzer and Krasser, Abstract order type extension and new results on the rectilinear crossing number. Comput. Geom. 36 (2007), 2–15, Table 1.

	[A006247]	[A063666]				[A006245]
n	#AOT	#realizable AOT	Δ	relative Δ	#mirror-symmetric AOT	#x-monotonePSLA
3	1	1	0	0	1	2
4	2	2	0	0	2	8
5	3	3	0	0	3	62
6	16	16	0	0	12	908
7	135	135	0	0	28	24,698
8	3,315	3,315	0	0	225	1,232,944
9	158,830	158,817	13	0,01 %	825	112,018,190
10	14,320,182	14,309,547	10,635	0,07 %	13,103	18,410,581,880
11	2,343,203,071	2,334,512,907	8,690,164	0,37 %	76,188	5,449,192,389,984
12	691,470,685,682					2,894,710,651,370,536

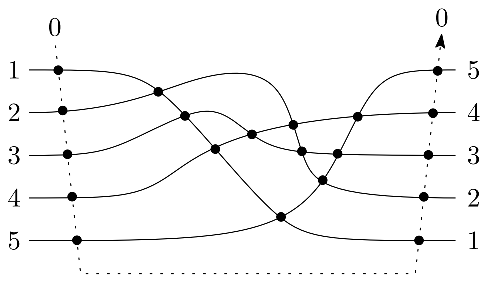
The last column counts the objects that the program actually enumerates one by one (almost, because we try to apply shortcuts). These numbers are known up to $n = 16$. For example, to get the 158,830 AOTs with 9 points, we go through all 1,232,944 PSLAs with 8 pseudolines, and select a subset by a lexicographic comparison, see Sections 11.2 and 11.4.

$$\#\text{OAOT} = 2 \times \#\text{AOT} - \#\text{mirror-symmetric AOT} \qquad [\text{A006246}]$$

#AOT equals the number of simple projective pseudoline arrangements with a marked cell.

2 Representation of a pseudoline arrangement

Here is an x-monotone pseudoline arrangement with $n = 5$ pseudolines, together with a primitive graphic representation of a different pseudoline arrangement as produced by the function *print_wiring_diagram* (see Section 10.1):



```
1-1-1-1 4-4 5-5
       X   X
2 3-3 4 1 5 4-4
 X   X   X
3 2 4 3 5 1 3-3
   X   X   X
4-4 2 5 3-3 1 2
     X       X
5-5-5 2-2-2-2 1
```

Pseudoline 1 starts topmost and ends bottommost. On the right end, the order of all pseudolines is reversed. There is an imaginary pseudoline 0 of very negative slope that intersects all other pseudolines from top to bottom at the very left and again intersects all pseudolines from bottom to top at the very right.

As a projective PSLA, the top face would be enclose by an edge that closes line 0 into a closed loop, from the intersection with 5 to the intersection with 0 (not shown in the picture). We consider this edge as the *starting edge* of this representation. In Section 11.1 we will consider the choice of a different starting edge in the same projective class of PSLAs.

2.1 The P-matrix (local sequences matrix) and its inverse

Here is a representation of the right example as a two-dimensional array, indicating for each pseudoline i the sequence P_i of crossings with the other lines. These sequences are called the *local sequences*. We will refer to the whole matrix as the P-matrix representation of a PSLA.

$$\begin{array}{lll}
P_0 = [1,2,3,4,5] & \bar{P}_0 = [\text{-},0,1,2,3,4] & \\
P_1 = [0,4,5,3,2] & \bar{P}_1 = [0,\text{-},4,3,1,2] & T_1 = [0,0,0,0,0] \\
P_2 = [0,3,4,5,1] & \bar{P}_2 = [0,4,\text{-},1,2,3] & T_2 = [0,0,0,0,1] \\
P_3 = [0,2,4,5,1] & \bar{P}_3 = [0,4,1,\text{-},2,3] & T_3 = [0,1,0,0,1] \\
P_4 = [0,2,3,1,5] & \bar{P}_4 = [0,3,1,2,\text{-},4] & T_4 = [0,1,1,1,0] \\
P_5 = [0,2,3,1,4] & \bar{P}_5 = [0,3,1,2,4,\text{-}] & T_5 = [0,1,1,1,1]
\end{array}$$

The first row and the first column are determined. Each row has n elements. We also use an inverse array $\bar{P}$, which is essentially the inverse permutation of each row. The j-th element of $\bar{P}_i$ gives the position in P_i where the crossing with j occurs. The diagonal entries are irrelevant. The column indices in $\bar{P}$ range from 0 to n; therefore we define the rows to have maximum length `MAXN` $+ 1$.

The binary matrix T is discussed in Section 10.3. It is defined in terms of the P-matrix by the rule $T_i[j] = 1$ if $P_i[j] < i$.

4 ⟨Types and data structures 4⟩ ≡
```
typedef int P_matrix[MAXN + 1][MAXN + 1];
```
See also chunks 9, 65, and 73

This code is used in chunk 6.

2.2 Linked representation

For modifying and extending PSLAs, it is best to work with a linked representation.

We will occasionally denote to the crossing point between two lines k and j by $k \times j$ or $j \times k$. Point (j,k) describes the crossing with line k along the line j. `SUCC`(j,k) and `PRED`(j,k) point to the next and previous crossing on line j. For (k,j) we get the corresponding information for the line k. In the example, we have `SUCC`$(2,3) = 5$ and accordingly `PRED`$(2,5) = 3$.

The infinite rays on line j are represented by the additional line 0: `SUCC`$(j,0)$ is the first (leftmost) crossing on line j, and `PRED`$(j,0)$ is the last crossing. The intersections on line 0 are cyclically ordered $1,\ldots,n$. Thus, `SUCC`$(0,i) \leftarrow i+1$ and `SUCC`$(0,n) = 1$.

The program works with a single linked-list representation, which is stored in the global arrays *succ* and *pred*. A single pair of these arrays is sufficient for the whole program.

```
5  #define SUCC(i, j)   succ[i][j]      /* access macros */
   #define PRED(i, j)   pred[i][j]
   #define LINK(j, k1, k2)
           {      /* make crossing with k_1 and k_2 adjacent on line j */
             SUCC(j, k1) ← k2;
             PRED(j, k2) ← k1;
           }
   ⟨ Global variables 5 ⟩ ≡
     int succ[MAXN + 1][MAXN + 1];
     int pred[MAXN + 1][MAXN + 1];
```

See also chunks 13, 21, 39, 50, 51, and 71

This code is used in chunk 6.

3 The main program

Each PSLA for n lines has a unique parent with $n-1$ lines. This defines a tree structure on the PSLAs. The principle of the enumeration algorithm is a depth-first traversal of this tree.

```
6  #define MAXN  15      /* The maximum number of pseudolines for which the program will work. */
     ⟨ Default preprocessor switch settings 7 ⟩
     ⟨ Include standard libaries 10 ⟩
     ⟨ Types and data structures 4 ⟩
     ⟨ Global variables 5 ⟩
     ⟨ Subroutines 27 ⟩
     ⟨ Core subroutines for recursive generation 16 ⟩
     int main(int argc, char *argv[])
     {
       ⟨ Parse the command line 14 ⟩;
   #if readdatabase      /* reading from the database */
       ⟨ Read all point sets of size n_max + 1 from the database and process them 77 ⟩
       return 0;
   #endif
   #if enumAOT
       ⟨ Initialize statistics and open reporting file 52 ⟩;
       ⟨ Start the generation 17 ⟩;
       ⟨ Report statistics 54 ⟩;
   #endif
       return 0;
     }
```

3.1 Preprocessor switches

The program has the enumeration procedure at its core, but it can be configured to perfom different tasks, by setting preprocessor switches at compile-time.

We assume that the program will anyway be modified and extended for specific counting or enumeration tasks, and it makes sense to set these options at compile-time.

(Other options, which are less permanent, can be set by command-line switches, see Section 3.5.)

The compilation can be controlled by defining the following preprocessor macros for be 0 or 1.

7 ⟨ Default preprocessor switch settings 7 ⟩ ≡
```
#ifndef enumAOT
#define enumAOT 1    /* indicates that the program purpose is enumeration of AOTs. This is the default
        action. Other purposes, which would still need to be implemented, might be enumeration of PSLAs. */
#endif
#ifndef readdatabase
#define readdatabase 0    /* version for reading point sets of the order-type */
#endif
#ifndef generatelist
#define generatelist 0    /* List all PSLAs plus their IDs, as preparation for generating exclude-files of
        nonrealizable AOTs, requires enumAOT to be set. */
#endif
#ifndef profile
#define profile 1
      /* gather statistics and profiling information. This does not cost much and should be on by default. */
#endif
```
This code is used in chunk 6.

3.2 On programming style

CWEB provides a good structuring facility while keeping all pieces and the documentation in one place. This leads to a large monolithic program in one file, as opposed to a separation in thematically grouped files that a C-project usually has.

For simplicity, I often use global variables.

Some variations of the program are implemented via preprocessor switches; for others, there is the change-file mechanism of CWEB.

¶ The *boolean* type.

9 ⟨ Types and data structures 4 ⟩ +≡
```
  typedef enum { false, true } boolean;
```

¶ Standard libraries

10 ⟨ Include standard libaries 10 ⟩ ≡
```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>
```
See also chunk 75.
This code is used in chunk 6.

3.3 Auxiliary macro for for-loops

I don't want to write x three times.

11
```
#define for_int_from_to(x, first, last) for (int x ← first; x ≤ last; x++)
  format for_int_from_to for
```

3.4 Parallelization

The user can parallelize the enumeration by specifying a *split level*, usually 8 or 9. The program will then work normally up to this level of the tree, that is, it will enumerate all 1,232,944 PSLAs with 8 lines, but it will only expand a selection of these PSLAs. The selection is determined as follows. As the PSLAs with 8 lines are enumerated, a running counter is incremented, thus implicitly assigning a number between 1 and 1,232,944 to each PSLA. We choose a modulus $m = parts$ and a value $k = part$. Then the program will explore the subtree of only those nodes whose number is congruent to k modulo m. By running the program for $k = 1, \ldots, m$, the work is split into m roughly equal parts.

This usually leads to a quite balanced partition of the work. For example, when enumerating AOTs with up to 13 points, the program had to visit all PSLAs with 12 pseudolines. We split the computation into 400 parts at level 9. The number of PSLAs with 12 lines that were visited in each part was in the range between 5,938,178,249,794 and 5,894,432,599,972 PSLAs, a variation of 0.74 %.

The variation of the runtime was more substantial: between 180 and 200 hours per run, and it may be due to differences between processors on which the tasks were run. The total running time was 3173 CPU days.

3.5 Command-line arguments

```
13  #define PRINT_INSTRUCTIONS fprintf(stderr,
            "Usage:␣%s␣n␣[-exclude␣excludefile]␣[splitlevel␣parts␣part]␣[fileprefix]\n",
            argv[0]);
    ⟨ Global variables 5 ⟩ +≡
      int n_max, split_level ← 0;
      unsigned int parts ← 1000, part ← 0;      /* default values */
      char *fileprefix ← "reportPSLA";     /* default name for the report-file */
      char *exclude_file_name ← 0;
      char fname[200] ← "";
      FILE *reportfile ← 0;
```

```
14  ¶⟨ Parse the command line 14 ⟩ ≡
      if (argc < 2) n_max ← 7;
      else {
        if (argv[1][0] ≡ '-') {      /* first argument "--help" gives help message. */
          PRINT_INSTRUCTIONS;
          exit(0);
        }
        n_max ← atoi(argv[1]);
      }
      printf("Enumeration␣up␣to␣n␣=␣%d␣pseudolines,␣%d␣points.\n", n_max, n_max + 1);
      if (n_max > MAXN) {
        fprintf(stderr, "The␣largest␣allowed␣value␣for␣n␣is␣%d.␣Aborting.\n", MAXN);
        exit(1);
      }
      int argshift ← 0;
      if (argc ≥ 3) {
        if (strcmp(argv[2], "-exclude") ≡ 0) {
          if (argc ≥ 4) {
            exclude_file_name ← argv[3];
            argshift ← 2;
            printf("Excluding␣entries␣from␣file␣%s.\n", exclude_file_name);
            ⟨ Open the exclude-file and read first line 23 ⟩
          }
          else {
            PRINT_INSTRUCTIONS;
            exit(1);
          }
        }
      }
      if (argc ≥ 3 + argshift) {
        split_level ← atoi(argv[2 + argshift]);
        if (split_level ≡ 0) {
          if (argv[3 + argshift][0] ≠ '-') fileprefix ← argv[3 + argshift];
          snprintf(fname, sizeof (fname) − 1, "%s-%d.txt", fileprefix, n_max);
          parts ← 1;
        }
        else {
          if (exclude_file_name ≠ 0) {
```

```
        fprintf(stderr, "The␣-exclude␣option␣with␣a␣positive␣splitlevel␣%d␣is␣not␣im\
            plemented.␣Aborting.\n", split_level);
        exit(1);
      }
      if (argc ≥ 4 + argshift) parts ← atoi(argv[3 + argshift]);
      if (argc ≥ 5 + argshift) part ← atoi(argv[4 + argshift]);
      part ← part % parts;
      if (argc ≥ 6 + argshift) fileprefix ← argv[5 + argshift];
      snprintf(fname, sizeof (fname) − 1, "%s-%d-S%d-part_%d_of_%d.txt", fileprefix, n_max, split_level,
          part, parts);
      printf("Partial␣enumeration:␣split␣at␣level␣n␣=␣%d.␣Part␣%d␣of␣%d.\n", split_level, part,
          parts);
    }
    printf("Results␣will␣be␣reported␣to␣file␣%s.\n", fname);
    fflush(stdout);
  }
```

This code is used in chunk 6.

4 Recursive Enumeration

This is the core of the algorithm. We extend an x-monotone pseudoline arrangement of $n-1$ lines $1, \ldots, n-1$, by threading an additional line n through it from the bottom face to the top face. The new line gets the largest slope of all lines.

Line 0 crosses the other lines in the order $1, 2, \ldots, n$.

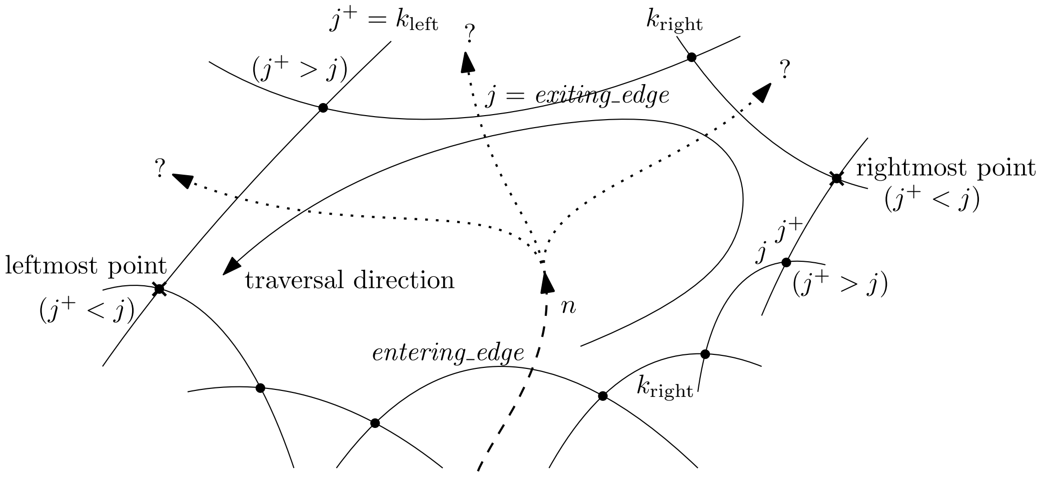


Figure 2: Threading line n through a face

16 ¶⟨ Core subroutines for recursive generation 16 ⟩ ≡

```
  void recursive_generate_PSLA_start(int n);

  void recursive_generate_PSLA(int entering_edge, int $k_{\text{right}}$, int n)
  {    /* The new line enters a face F from the bottom. The edge through which it crosses is part of line
          entering_edge, and its right endpoint is the crossing with $k_{\text{right}}$. */
    int j ← entering_edge;
    int $j^+$ ← $k_{\text{right}}$;
    while ($j^+ > j$) {    /* find right vertex of the current cell F */
      int $j^+_{\text{old}}$ ← $j^+$;
      $j^+$ ← SUCC($j^+$, j);
      j ← $j^+_{\text{old}}$;
    }    /* the right vertex is the intersection of j and $j^+$ */
    if ($j^+ \equiv 0$) {    /* F is unbounded */
      if ($j \equiv n-1$) {    /* F is the top face. */
        LINK(n, entering_edge, 0);    /* complete the insertion of line n */
```

```
            ⟨Update counters 18⟩
            ⟨Process the PSLA; return if excluded 19⟩
            if ($n < n\_max$)
              if ($n \neq split\_level \lor countPSLA[n] \,\%\, parts \equiv part$) {
#if enumAOT      /* screening one level below */
                boolean hopeful ← true;
                if ($n \equiv n\_max - 1$) {
                  ⟨Screen one level below level n_max 49⟩
                }
                if (hopeful)
#endif
                  recursive_generate_PSLA_start($n + 1$);      /* thread the next pseudoline */
              }
            return;
          }
          else {      /* jump to the upper bounding ray of F */
            $j^+ \leftarrow j + 1$;
            $j \leftarrow 0$;
          }
        }      /* Now the crossing $j \times j^+$ is the rightmost vertex of the face F. The edge $j^+$ is on the upper side.
                 If F is bounded, j is on the lower side; otherwise, $j = 0$. */
        do {      /* scan the upper edges of F from right to left and try them out. */
          $k_{\text{right}} \leftarrow j$;
          $j \leftarrow j^+$;
          int $k_{\text{left}} \leftarrow j^+ \leftarrow$ PRED($j, k_{\text{right}}$);      /* j is the exiting edge */
          LINK($j,\ k_{\text{left}}, n$);      /* insert the crossing to prepare for the recursive call */
          LINK($j,\ n, k_{\text{right}}$);
          LINK($n,\ entering\_edge, j$);
          recursive_generate_PSLA($j, k_{\text{right}}, n$);      /* enter the recursion */
          LINK($j,\ k_{\text{left}}, k_{\text{right}}$);      /* undo the changes */
        } while ($j^+ > j$);      /* terminate at left endpoint of the face F or at unbounded ray ($j^+$=0) */
        return;
      }
      void recursive_generate_PSLA_start(int n)
      {
        localCountPSLA[n] ← 0;      /* reset child counter */
        LINK($0,\ n - 1, n$);      /* insert line n on line 0 */
        LINK($0,\ n, 1$);
        recursive_generate_PSLA($0, 0, n$);      /* enter the recursion. */
           /* There is a little trick: With these parameters 0, 0, the procedure recursive_generate_PSLA will skip
             the first loop and will then correctly scan the edges of the bottom face F from right to left. */
        LINK($0,\ n - 1, 1$);      /* undo the insertion of line n */
      }
```
This code is used in chunk 6.

¶ Start with 2 pseudolines.

17 ⟨Start the generation 17⟩ ≡
```
  LINK(1, 0, 2);
  LINK(1, 2, 0);
  LINK(2, 0, 1);
  LINK(2, 1, 0);
  LINK(0, 1, 2);      /* LINK(0, 2, 3) and LINK(0, 3, 1) will be established shortly in the first recursive call. */
  recursive_generate_PSLA_start(3);
```
This code is used in chunk 6.

18 ¶⟨Update counters 18⟩ ≡
```
  countPSLA[n]++;      /* update global counter (“accession number”) */
  localCountPSLA[n]++;      /* update local counter */
```
This code is used in chunk 16.

4.1 Handling of a PSLA

```
19  ⟨Process the PSLA; return if excluded 19⟩ ≡
      ⟨Indicate Progress 20⟩;
      boolean is_excluded ← false;
      ⟨Check for exclusion and set the flag is_excluded 22⟩
      if (is_excluded) return;
      ⟨Gather statistics about the AOT, collect output 53⟩
      ⟨Further processing of the AOT 59⟩
```

This code is used in chunk 16.

¶ Indicate Progress. The user should not despair while waiting for a long run.

```
20  ⟨Indicate Progress 20⟩ ≡
      if (n ≡ n_max ∧ countPSLA[n] % 50000000000 ≡ 0) {      /* 5 × 10^10 */
        printf("..%Ld..␣", countPSLA[n]);
        P_matrix P;
        convert_to_P_matrix(&P, n);
        print_pseudolines_short(&P, n);
        fflush(stdout);
      }
```

This code is used in chunk 19.

5 Handling the exclude-file

It is assumed that the codes in the exclude-file are sorted in strictly increasing lexicographic order, and no code is a prefix of another code.

To give an example, here are a few lines from the middle of the file `exclude10.txt`:

```
1.3.7.12.9.17.45
1.3.7.12.9.18.35
1.3.7.12.9.18.37
1.3.7.12.9.19
1.3.7.12.9.20
1.3.7.12.9.21.36
1.3.7.12.9.21.37
```

NOTE: As currently implemented, the handling of the exclude-file does not work together with the parallelization through the *splitlevel* option. This is checked.

The array *excluded_code*[3... *excluded_length*] always contains the decimal code of the next PSLA that should be excluded from the enumeration. During the enumeration, the decimal code of the currently visited tree node (as stored in *localCountPSLA*) agrees with *excluded_code* up to position *matched_length*.

```
21  ⟨Global variables 5⟩ +≡
      unsigned excluded_code[MAXN + 3];
      int excluded_length ← 0;
      int matched_length ← 0;      /* These initial values will never lead to any match. */
      FILE *exclude_file;
      char exclude_file_line[100];
```

```
22  ¶⟨Check for exclusion and set the flag is_excluded 22⟩ ≡
      if (n ≡ matched_length + 1 ∧ localCountPSLA[n] ≡ excluded_code[n]) {
        matched_length ← n;      /* one more matching entry was found. */
        if (matched_length ≡ excluded_length) {      /* skip this PSLA and the whole subtree */
          is_excluded ← true;
          ⟨Get the next excluded decimal code from the exclude-file 24⟩
          ⟨Determine the matched length matched_length 25⟩
        }
      }
```

See also chunk 58.

This code is used in chunk 19.

23 ¶⟨ Open the exclude-file and read first line 23 ⟩ ≡

```
$exclude\_file \leftarrow fopen(exclude\_file\_name, "r");$
⟨ Get the next excluded decimal code from the exclude-file 24 ⟩
$matched\_length \leftarrow 2;$
```

This code is used in chunk 14.

24 ¶⟨ Get the next excluded decimal code from the exclude-file 24 ⟩ ≡

```
do {
  if (fscanf(exclude_file, "%s\n", exclude_file_line) ≠ EOF) {
    char *str1 ← exclude_file_line;
    char *token, *saveptr;
    excluded_length ← 2;
    while (true) {
      token ← strtok_r(str1, ".", &saveptr);
      if (token ≡ Λ) break;
      assert(excluded_length < MAXN + 3 − 1);
      excluded_code[++excluded_length] ← atoi(token);
      str1 ← Λ;
    }
  }
  else {
    excluded_length ← 0;    /* end of file reached. */
    fclose(exclude_file);
  }
} while (excluded_length > n_max);    /* patterns longer than n_max are filtered. */
```

This code is used in chunks 22 and 23

¶ (The following program piece could be accelerated if the exclude-file would not store every decimal code completely but eliminate the common prefix, indicating only the deviation from the previous code.)

25 ⟨ Determine the matched length *matched_length* 25 ⟩ ≡

```
matched_length ← 2;
while (excluded_code[matched_length + 1] ≡ localCountPSLA[matched_length + 1] ∧ matched_length <
      excluded_length ∧ matched_length < n)
  matched_length++;
```

This code is used in chunk 22.

6 Conversion between different representations

6.1 Convert from linked list to P-matrix

Input: PSLA with n lines $1..n$, stored in *succ*. Output: P-matrix of size $(n+1)\times n$ for pseudoline arrangement on n pseudolines.

27 ⟨ Subroutines 27 ⟩ ≡

```
void convert_to_P_matrix(P_matrix *P, int n)
{
  int j ← 1;
  for_int_from_to (i, 0, n) {
    for_int_from_to (p, 0, n − 1) {
      (*P)[i][p] ← j;
      j ← SUCC(i, j);
    }
    j ← 0;    /* j starts at 0 except for the very first line. */
  }
}
```

See also chunks 28, 30, 31, 33, 34, 35, 38, 40, 41, 43, 66, 68, 72, 74, 76, and 78

This code is used in chunk 6.

6.2 Convert from linked list to inverse P-matrix

The inverse P-matrix matrix $\bar{P}$ gives the following information: $\bar{P}_{jk} = p$ if the intersection between line j and line k is the p-th intersection on line j $(p = 0, \ldots, n-1)$. This is used to answer orientation queries about the pseudoline arrangement, and about the dual point set, see Section 7.

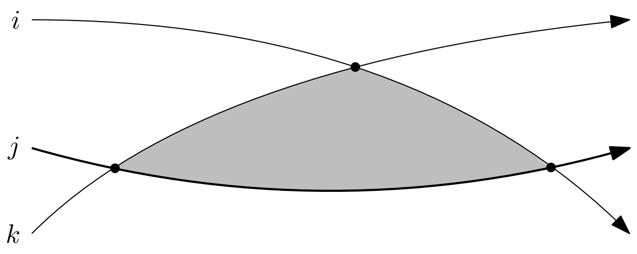
28 ⟨Subroutines 27⟩ +≡
```
void $convert\_to\_inverse\_P\_matrix$(P_matrix $*\bar{P}$, int $n$)
{
  int $j \leftarrow 1$;
  for_int_from_to $(i, 0, n)$ {
    for_int_from_to $(p, 0, n-1)$ {
      $(*\bar{P})[i][j] \leftarrow p$;
      $j \leftarrow$ SUCC$(i, j)$;
    }
    $j \leftarrow 0$;      /* j starts at 0 except for the very first line. */
  }
}
```

7 The orientation predicate

We compute the orientation predicate in constant time from the inverse permutation array $\bar{P}$. It is a **boolean** predicate that returns *true* if the points i, j, k are in counterclockwise order. It works only when the three indices are distinct.

It is computed by comparing the intersections on line j.

If $i < j < k$, this predicate is *true* if the intersection of lines i and k lies above line j. When i, j, k are permuted, the predicate must change according to the sign of the permutation. For documentation purposes, we specify an expression *getOrientation_explicit* that distinguishes all 3! possibilities in which the indices i, j, k can be ordered. *getOrientation* is a simpler, equivalent, expression.



29 **#define** *getOrientation_explicit*$(\bar{P}, i, j, k)$
$(i < j \wedge j < k\ ?\ \bar{P}[i][j] > \bar{P}[i][k] : i < k \wedge k < j\ ?\ \bar{P}[i][j] > \bar{P}[i][k] : j < i \wedge i < k\ ?\ \bar{P}[i][j] < \bar{P}[i][k] :$
$j < k \wedge k < i\ ?\ \bar{P}[i][j] > \bar{P}[i][k] : k < j \wedge j < i\ ?\ \bar{P}[i][j] > \bar{P}[i][k] : k < i \wedge i < j\ ?\ \bar{P}[i][j] < \bar{P}[i][k] : 0)$
#define *getOrientation*$(\bar{P}, i, j, k)$ $((i < j) \oplus (j < k) \oplus (\bar{P}[j][i] > \bar{P}[j][k]))$

8 Compute the convex hull points of an AOT from the PSLA

This is easy; we just scan the top face. We know that 0, 1, and n belong to the convex hull. 0 represents the line at ∞).

The input is taken from the global array *succ*. (The array *pred* is not used.) The output is stored in the array *hulledges*.

30 ⟨Subroutines 27⟩ +≡
```
int $upper\_hull\_PSLA$(int $n$, int $*hulledges$)
{
  $hulledges[0] \leftarrow 0$;
  int $hullsize \leftarrow 1$;
  int $k \leftarrow 0$, $k_{\mathrm{left}}$, $k_{\mathrm{right}} \leftarrow 1$;
  do {      /* scan the edges of the top face $F$ from left to right */
    $k_{\mathrm{left}} \leftarrow k$;
    $k \leftarrow k_{\mathrm{right}}$;
    $k_{\mathrm{right}} \leftarrow$ SUCC$(k, k_{\mathrm{left}})$;
    hulledges[hullsize++] ← k;
  } while ($k_{\text{right}} \neq 0$);
  return hullsize;      /* Result is the number of extreme points. */
}
```

9 Unique identifiers, Dewey decimal notation

The recursive enumeration algorithm imposes an implicit tree structure on PSLAs: the parents of a PSLA with n lines is the unique PSLA on $n-1$ lines from which it is generated. We number the children of each node in the order in which they are generated, starting from 1. The sequence of labels on the path from the root to a node gives a unique identifier to each node in the tree. (This is, however, specific to details of the enumeration algorithm: in which order edges are considered for crossing in the insertion, the choice of lexicographic criterion.)

The purpose of this scheme is that it allows to identify a PSLA even if we parallelize the computation, and one thread of the program only visits certain branches of the tree.

The enumeration tree has only one node on levels 1 and 2. Thus we start the fingerprint at level 3.

(In addition, the PSLAs of each size n are numbered by the global counter *countPSLA*. This can be used as an "accession number" to identify a PSLA, provided that the PSLAs of size n are enumerated in full.)

31 ⟨ Subroutines 27 ⟩ +≡

```
  unsigned localCountPSLA[MAXN + 3];      /* another global variable */
  void print_id(int n)
  {
    printf("%d", localCountPSLA[3]);
    for_int_from_to (i, 4, n) printf(".%d", localCountPSLA[i]);
  }
```

10 Output

10.1 Prettyprinting of a wiring diagram

Fill a buffer of lines columnwise from left to right.

33
```
#define TO_CHAR(i) ((char)((i < 10 ? (int) '0' : ((int) 'A' − 10)) + i))
               /* lines > 9 are codes as letters */
```

⟨ Subroutines 27 ⟩ +≡

```
  void print_wiring_diagram(int n)
  {     /* ASCII, horizontal, column-wise */
    int next_crossing[MAXN + 1];      /* current crossing on each line */
    int line_at[MAXN];      /* which line is on the i-th track, i = 0, . . . , n − 1 */
    boolean crossing[MAXN − 1];      /* Is there a crossing between track i and i + 1? */
    char buffer_line[2 * MAXN − 1][MAXN * MAXN];
        /* enough columns for 2 characters per crossing plus a little extra */
    for_int_from_to (j, 0, n − 1) {
      line_at[j] ← j + 1;
      next_crossing[j + 1] ← SUCC(j + 1, 0);      /* crossing #0 with line 0 "at ∞" is not considered. */
    }
    int num_crossings ← 0;
    int column ← 0;
    for_int_from_to (p, 0, 2 * n − 2) buffer_line[p][column] ← '␣';      /*empty start column */
    for (column ← 1; ; column++) {
      for_int_from_to (p, 0, n − 1) buffer_line[2 * p][column] ← TO_CHAR(line_at[p]);
      for_int_from_to (p, 0, n − 2) buffer_line[2 * p + 1][column] ← '␣';
      column++;
      /* find where crossings occur, set boolean array crossing[0 . . n − 2] accordingly. */
      boolean something_to_do ← false;
      for_int_from_to (p, 0, n − 2) {
        int i ← line_at[p];
        int j ← line_at[p + 1];
        $crossing[p] \leftarrow next\_crossing[i] \equiv j \land next\_crossing[j] \equiv i$;
        if $(crossing[p])$ $something\_to\_do \leftarrow true$;
      }
      if $(\neg something\_to\_do)$ break;
      for_int_from_to $(p, 0, n-1)$ $buffer\_line[2*p][column] \leftarrow$ '-';    /* continuation column */
      for_int_from_to $(p, 0, n-2)$
        if $(crossing[p])$ {
          $num\_crossings$++;
          $buffer\_line[2*p+1][column] \leftarrow$ 'X';    /* print the crossing as an 'X' */
          $buffer\_line[2*p][column] \leftarrow buffer\_line[2*p+2][column] \leftarrow$ '␣';    /* erase the adjacent lines */
          /* carry out the crossing: */
          int $i \leftarrow line\_at[p]$;
          int $j \leftarrow line\_at[p+1]$;
          $next\_crossing[i] \leftarrow$ SUCC$(i, next\_crossing[i])$;
          $next\_crossing[j] \leftarrow$ SUCC$(j, next\_crossing[j])$;
          $line\_at[p] \leftarrow j$;
          $line\_at[p+1] \leftarrow i$;
        }
        else $buffer\_line[2*p+1][column] \leftarrow$ '␣';
    }
    for_int_from_to $(p, 0, 2*n-2)$ {
      $buffer\_line[p][column] \leftarrow 0$;    /* finish the lines */
      printf("%s\n", $buffer\_line[p]$);    /* and print them */
    }
    $assert(num\_crossings \equiv n*(n-1)/2)$;
  }
```

10.2 Fingerprints

A concise description of a PSLA consists of the P-matrix entries, prefixed by the letter `P` and with the rows separated by `!` symbols. The procedure *print_pseudolines_compact* prints a more compact version that leaves out redundant parts, which are the same in all P-matrices or which can be easily inferred from the remaining information.

```
34 ⟨Subroutines 27⟩ +≡
  void print_pseudolines_short(P_matrix *P, int n)
  {
    printf("P");
    for_int_from_to $(i, 0, n)$ {
      printf("!");
      for_int_from_to $(j, 0, n-1)$ printf("%c", TO_CHAR((*P)[i][j]));
    }
    printf("\n");
  }
  void print_pseudolines_compact(P_matrix *P, int n)
  {
    for_int_from_to $(i, 1, n)$ {    /* line 0 is always 1234.. */
      if $(i > 1)$ printf("!");    /* line $P_i$ starts with 0 and is a permutation that misses $i$. */
      for_int_from_to $(j, 1, n-2)$ printf("%c", TO_CHAR((*P)[i][j]));
    }
  }
```

10.3 A more compact fingerprint

A PSLA is uniquely determined by the $n \times n$ binary matrix T, which is defined in terms of the P-matrix by the rule $T_i[j] = 1$ if $P_i[j] < i$. An example is shown in Section 2.1. The fact that this is enough can be seen from the fact that this information is sufficient for drawing the wiring-diagram. It has been shown by Stefan Felsner, On the number of arrangements of pseudolines, *Discrete & Computational Geometry* **18** (1997), 257–267, doi:10.1007/PL00009318, Theorem 1. See also Stefan Felsner, *Geometric Graphs and Arrangements*, Vieweg, 2004, Chapter 6, Theorem 6.6.

(The so-called *replace matrices* from that paper would offer even more savings.)

The first column of T is fixed. The first row T_1 and the last row T_n is fixed, and they need not be coded. Also, since row T_i contains $i-1$ ones, we can omit the last entry per row, since it can be reconstructed from the remaining entries. Thus we encode the $(n-2)\times(n-2)$ array obtained removing the borders from the original $n\times n$ array.

We code 6 bits into on of 64 ASCII symbols, using the 52 small and capital letters, the 10 digits, and the 2 symbols `+` and `-`. (Care must be taken when sorting such keys with the UNIX `sort` utility, because, depending on the *locale* settings, the sorting program may conflate uppercase and lowercase letters.)

We use this encoding for the case when n is known. Therefore we need not worry about terminating the code.

35
```
#define FINGERPRINT_LENGTH 30   /* enough for 13 × 13 bits plus terminating null */
⟨ Subroutines 27 ⟩ +≡
  char fingerprint[FINGERPRINT_LENGTH];   /* global variable */
  char encode_bits(int acc)
  {
    if (acc < 26) return (char)(acc + (int) 'A');
    else if (acc < 52) return (char)(acc − 26 + (int) 'a');
    else if (acc < 62) return (char)(acc − 52 + (int) '0');
    else if (acc ≡ 62) return '+';
    else return '-';
  }
  void compute_fingerprint(P_matrix *P, int n)
  {
    int charpos ← 0;
    int bit_num ← 0;
    int acc ← 0;
    for_int_from_to (i, 1, n − 1)
      for_int_from_to (j, 1, n − 1) {
        acc <<= 1;
        if ((*P)[i][j] < i) acc |= 1;
        bit_num += 1;
        if (bit_num ≡ 6) {
          fingerprint[charpos++] ← encode_bits(acc);
          assert(charpos < FINGERPRINT_LENGTH − 1);
          bit_num ← acc ← 0;
        }
      }
    if (bit_num) fingerprint[charpos++] ← encode_bits(acc << (6 − bit_num));
    assert(charpos < FINGERPRINT_LENGTH − 1);
    fingerprint[charpos] ← '\0';
  }
```

36
```
¶⟨ Print PSLA-fingerprint 36 ⟩ ≡
  {
    P_matrix P;
    convert_to_P_matrix(&P, n);
    compute_fingerprint(&P, n);
    printf("%s:", fingerprint);   /* terminated by a colon */
  }
```
This code is used in chunk 60.

11 Enumerating abstract order types

11.1 Compute the P-matrix for a different starting edge

For reference we show how to compute the matrix from an arbitrary *starting edge*. The starting edge is specified by the line *line0* on which it lies, its right endvertex *right_vertex*, and a a *direction*. The edge lies between *left_vertex* ≡ PRED(*line0*, *right_vertex*) and *right_vertex* (which fulfills *right_vertex* ≡ SUCC(*line0*, *left_vertex*)).

If *reversed* $\equiv$ *false*, the direction follows the *succ*-pointers, and otherwise, the *pred*-pointers. The P-matrix is filled row-wise from right to left.

The stardard, unchanged, setting would be obtained with *line0* $\equiv 0$ and *right_vertex* $\equiv 1$.

The main application of this procedure is when we try out different convex hull vertices as pivot points, generating all PSLAs that can represent a given AOT, see Section 11.2. (However, we will never use the procedure *compute_new_P_matrix* directly, we use a version that computes several such P-matrices in parallel, entry by entry.)

38 ⟨Subroutines 27⟩ +≡

```
void compute_new_P_matrix(P_matrix *P, int n, int line0, int right_vertex, boolean reversed)
{
  int Sequence[MAXN + 1];
      /* Sequence[p] gives the p-th crossing (in the SUCC-direction) on line start_line. */
  int new_label[MAXN + 1];      /* new_label[j] gives the label that is use for the line with the original label
       j. */      /* Sequence and new_label are inverse permutations of each other. */
  new_label[line0] ← 0;
  int i ← right_vertex;
  for_int_from_to (p, 1, n) {
    new_label[i] ← p;
    Sequence[p] ← i;
    i ← SUCC(line0, i);
  }
  for_int_from_to (q, 0, n − 1) (*P)[0][q] ← q + 1;      /* row 0 is always the same */
  for_int_from_to (p, 1, n) {      /* compute row P_p of P-matrix */
    int pos ← reversed ? n + 1 − p : p;
    (*P)[p][0] ← 0;
    int i ← Sequence[pos];      /* Alternatively, i could be set via PRED or SUCC. */
    int j ← line0;
       /* We fill row P_p from right to left. The reason for this choice is explained in Section 11.2. */
    for_int_from_to (q, 1, n − 1) {      /* Compute P_{p,n−q} */
      j ← reversed ? SUCC(i, j) : PRED(i, j);
      (*P)[p][n − q] ← new_label[j];
    }
  }
}
```

11.2 Lexicographically smallest P-matrix representation

In order to generate every AOT only once, we check whether the the current P-matrix P the smallest among all P-matrices P' that represent the same AOT, except that the AOT is rotated or reflected.

We have to try all convex hull points as pivot points, and for each pivot point we have to choose two directions, reflected (*reversed*) and unreflected. The average number of extreme vertices is slightly less than 4. It does not pay off to shorten the loop considerably. (The average *squared* face size matters!)

In the lexicographic comparison between PSLAs, we consider the elements of the P-matrix row-wise *from right to left*, i.e., in the order $P_{1n}, P_{1,n-1}, \ldots, P_{11}; P_{2n}, P_{2,n-1}, \ldots, P_{21}; \ldots$. Here we number the entries in each row from 1 to n, unlike in the C program. In comparison with the more natural left-to-right order, this gives, experimentally, a quicker way to eliminate tentative P-matrices than the left-to-right order. (Those experiments were done with an early version of the program; there was no systematic exploration of the various possibilities.)

39 ⟨Global variables 5⟩ +≡

```
int Sequence[MAXN + 1][MAXN + 1];      /* Sequence[r][p] gives the p-th crossing on the r-th hull edge. */
int new_label[MAXN + 1][MAXN + 1];      /* When the r-th hull edge is used in the role of line 0, new_label[r][j]
      gives the index that is use for the (original) line j. */
int candidate[2 * (MAXN + 1)];      /* list of candidates, gives the index r into hulledges */
int current_crossing[2 * (MAXN + 1)];      /* indexed by candidate number */
int P_1_n_forward[MAXN + 1];
int P_1_n_reverse[MAXN + 1];
```

¶ The label arrays are not computed for those candidates that are excluded by the comparison of the *P_1_n_forward* values (unless the flag *compute_all* is set).

```
40 ⟨Subroutines 27⟩ +≡
  void prepare_label_arrays(int n, int hulledges[], int hullsize, boolean compute_all)
  {
    for_int_from_to $(r, 0, hullsize - 1)$
      if $(compute\_all \lor P\_1\_n\_reverse[r] \equiv P\_1\_n\_forward[0] \lor (r > 0 \land P\_1\_n\_forward[r] \equiv P\_1\_n\_forward[0]))$
        {    /* otherwise not needed. */
        int $line0 \leftarrow hulledges[r]$;
        $new\_label[r][line0] \leftarrow 0$;
        int $i \leftarrow (r < hullsize - 1)\ ?\ hulledges[r+1] : 0$;    /* $0 \equiv hulledges[0]$ */
        for_int_from_to $(p, 1, n)$ {
          $new\_label[r][i] \leftarrow p$;
          $Sequence[r][p] \leftarrow i$;
          $i \leftarrow$ SUCC$(line0, i)$;
        }
      }
  }
```

11.3 Compute the lex-smallest representation

The input is taken from the global *succ* and *pred* arrays. The function assumes that *hulledges* and $h = hullsize$ have been computed.

If the test returns *true*, the procedure also sets some output parameters that characterize the symmetry of the AOT: These output parameters — *rotation_period*, *has_mirror_symmetry*, and *has_fixed_vertex* — are determined on the way as a side result. The parameter *has_fixed_vertex* is only set if the PSLA is mirror-symmetric.

We scan the entries of P row-wise from right to left. We maintain the list of solutions that are still *candidates* to be lex-smallest. Initially we have $2 \times hullsize$ candidates, *hullsize* "forward" candidates and the same number of mirror-symmetric, reversed candidates.

The candidates with numbers $0 \mathinner{..} numcandidates\_forward - 1$ are forward candidates. The remaining candidates up to $numcandidates - 1$ are reverse (mirror) candidates.

If information about mirror symmetry is not necessary, then the mirror candidates can be omitted.

```
41 ⟨Subroutines 27⟩ +≡
  void compute_lex_smallest_P_matrix(P_matrix *P, int n, int *hulledges, int hullsize)
  {
    for_int_from_to $(q, 0, n-1)$ $(*P)[0][q] \leftarrow q + 1$;    /* row 0 */
    prepare_label_arrays$(n, hulledges, hullsize, true)$;
    int $numcandidates \leftarrow 0$;
    for_int_from_to $(r, 0, hullsize - 1)$ $candidate[numcandidates{+}{+}] \leftarrow r$;
    int $numcandidates\_forward \leftarrow numcandidates$;
    for_int_from_to $(r, 0, hullsize - 1)$ $candidate[numcandidates{+}{+}] \leftarrow r$;
    for_int_from_to $(p, 1, n)$ {    /* compute row $P_p$ of the $P$-matrix */
      $(*P)[p][0] \leftarrow 0$;
      for_int_from_to $(c, 0, numcandidates - 1)$ {
        int $r \leftarrow candidate[c]$;
        $current\_crossing[c] \leftarrow hulledges[r]$;    /* plays the role of line 0 */
      }
      for_int_from_to $(q, 1, n-1)$ {
          /* Compute $P_{p,n-q}$ by taking the minimum over all candidate choices of line 0. */
        int $c$;
        int $new\_candidates$, $new\_candidates\_forward$;
        int $current\_min \leftarrow n + 1$;    /* essentially $\infty$ */
        boolean $reversed \leftarrow false$;
        int $pos \leftarrow p$;    /* position of line 0; the line we are currently searching in Sequence */
        for $(c \leftarrow 0;\ c < numcandidates\_forward;\ c{+}{+})$ {⟨Process candidate $c$, keep in list and advance
            $new\_candidates$ if equal; reset $new\_candidates$ if better value than $current\_min$ is found 42⟩}
        $new\_candidates\_forward \leftarrow new\_candidates$;    /* can be reset in the next loop */
```

```
            $reversed \leftarrow true$;
            $pos \leftarrow n + 1 - p$;
            for ( ; $c < numcandidates$; $c$++) {
              ⟨ Process candidate $c$, keep in list and advance $new\_candidates$ if equal; reset $new\_candidates$ if
                  better value than $current\_min$ is found 42 ⟩
            }
            $numcandidates\_forward \leftarrow new\_candidates\_forward$;
            $numcandidates \leftarrow new\_candidates$;
            $(*P)[p][n - q] \leftarrow current\_min$;      /* could enter a shortcut as soon as $numcandidates \equiv 1$ */
          }
        }
      }
```

¶ The list *candidate* of candidates is scanned and simultaneously overwritten with new values.

42 ⟨ Process candidate c, keep in list and advance *new_candidates* if equal; reset *new_candidates* if better value than *current_min* is found 42 ⟩ ≡

```
  int $r \leftarrow candidate[c]$;
  int $i \leftarrow Sequence[r][pos]$;      /* We are proceeding on line i */
  int $j \leftarrow current\_crossing[c]$;
  $j \leftarrow reversed$ ? SUCC$(i, j)$ : PRED$(i, j)$;
  int $a \leftarrow new\_label[r][j]$;
  if $(reversed \land a \neq 0)$ $a \leftarrow n + 1 - a$;
  if $(a < current\_min)$      /* new record: */
  {
    $new\_candidates \leftarrow new\_candidates\_forward \leftarrow 0$;
    $current\_min \leftarrow a$;
  }
  if $(a \equiv current\_min)$ {      /* candidate survives. */
    $candidate[new\_candidates] \leftarrow r$;
    $current\_crossing[new\_candidates] \leftarrow j$;
    $new\_candidates$++;
  }      /* Otherwise the candidate is skipped. */
```

This code is used in chunk 41.

11.4 Test if the current PSLA gives the lex-smallest P-matrix corresponding to the same AOT

This is a variation of the procedure *compute_lex_smallest_P_matrix*. The output parameters *rotation_period*, *has_mirror_symmetry*, *has_fixed_vertex*, which characterize the symmetry of the AOT, are determined on the way, as a side result.

As a speed-up, there is a fast screening procedure that tries to eliminate a few candidates in advance, see Section 12.

43 ⟨ Subroutines 27 ⟩ +≡

```
  ⟨ Screening procedures 47 ⟩
  boolean $has\_lex\_smallest\_P\_matrix$(int $n$, int $*hulledges$, int $hullsize$, int $*rotation\_period$, boolean
        $*has\_mirror\_symmetry$, boolean $*has\_fixed\_vertex$)
  {
    if $(\neg screen\_lex\_smallest(n, hulledges, hullsize))$ return $false$;
#if profile
    $numTests$++;      /* has_lex_smallest_P_matrix tests that pass the screening test */
#endif
    $prepare\_label\_arrays(n, hulledges, hullsize, false)$;
    int $numcandidates \leftarrow 0$;
    for_int_from_to $(r, 1, hullsize - 1)$
      if $(P\_1\_n\_forward[r] \equiv P\_1\_n\_forward[0])$ $candidate[numcandidates$++$] \leftarrow r$;
    int $numcandidates\_forward \leftarrow numcandidates$;
    for_int_from_to $(r, 0, hullsize - 1)$
      if $(P\_1\_n\_reverse[r] \equiv P\_1\_n\_forward[0])$ $candidate[numcandidates$++$] \leftarrow r$;
    for_int_from_to $(p, 1, n)$ {    /* explore row $P_p$ of the $P$-matrix */
      int $current\_crossing\_0 \leftarrow 0$;    /* candidate $c = 0$ is treated specially. */
      for_int_from_to $(c, 0, numcandidates - 1)$ {
        int $r \leftarrow candidate[c]$;    /* plays the role of line 1 */
        $current\_crossing[c] \leftarrow hulledges[r]$;    /* plays the role of line 0 */
      }
      for_int_from_to $(q, 1, n - 2)$ {    /* Compute $P_{p,n-q}$ for all choices of line 0. The last entry $q = n - 1$
              can be omitted, because in every matrix, row $P_p$ is a permutation of the same elements. If all
              elements except the last one agree, then the last one must also agree. */
        int $target\_value \leftarrow current\_crossing\_0 \leftarrow$ PRED$(p, current\_crossing\_0)$;
           /* special treatment of candidate 0: current line $i$ is line $p$; no relabeling necessary. */
        int $c$;
        int $new\_candidates \leftarrow 0$;
        boolean $reversed \leftarrow false$;
        int $pos \leftarrow p$;    /* position of line 0 */
        for $(c \leftarrow 0;\ c < numcandidates\_forward;\ c{+}{+})$ {
          ⟨ Process candidate $c$, keep in list and advance $new\_candidates$ if successful; return $false$ if better
              value than $target\_value$ is found 44 ⟩
        }
        $numcandidates\_forward \leftarrow new\_candidates$;
        $reversed \leftarrow true$;
        $pos \leftarrow n + 1 - p$;
        for ( ; $c < numcandidates$; $c{+}{+}$) {    /* continue the previous loop */
          ⟨ Process candidate $c$, keep in list and advance $new\_candidates$ if successful; return $false$ if better
              value than $target\_value$ is found 44 ⟩
        }
        $numcandidates \leftarrow new\_candidates$;
        if $(numcandidates \equiv 0)$ {    /* early return */
        $*rotation\_period \leftarrow hullsize$;
        $*has\_mirror\_symmetry \leftarrow false$;
        return $true$;
        }
      }
    }
    ⟨ Determine the result parameters $rotation\_period$, $has\_mirror\_symmetry$, $has\_fixed\_vertex$, by analyzing
        the set of remaining candidates 45 ⟩
    return $true$;
  }
```

¶ The current candidate is *successful* if its value $P_{p,n-q}$ agrees with the *target_value*, the value of $P_{p,n-q}$ in the matrix P^0.

44 ⟨ Process candidate c, keep in list and advance *new_candidates* if successful; return *false* if better value than *target_value* is found 44 ⟩ ≡

```
#if profile
  $numComparisons{+}{+}$;
#endif
  int $r \leftarrow candidate[c]$;
  int $i \leftarrow Sequence[r][pos]$;
  int $j \leftarrow current\_crossing[c]$;
  $j \leftarrow reversed$ ? SUCC$(i, j)$ : PRED$(i, j)$;
  int $a \leftarrow new\_label[r][j]$;
  if $(reversed \wedge a \neq 0)$ $a \leftarrow n + 1 - a$;
  if $(a < target\_value)$ return $false$;
  if $(a \equiv target\_value)$ {
    $candidate[new\_candidates] \leftarrow r$;
    $current\_crossing[new\_candidates] \leftarrow j$;
    $new\_candidates{+}{+}$;
  }
```

This code is used in chunk 43.

```
45 ¶⟨Determine the result parameters rotation_period, has_mirror_symmetry, has_fixed_vertex, by analyzing the
        set of remaining candidates 45⟩ ≡
   {
     if (numcandidates_forward > 0) *rotation_period ← candidate[0];
     else *rotation_period ← hullsize;
     *has_mirror_symmetry ← numcandidates > numcandidates_forward;
     if (*has_mirror_symmetry) {
       int symmetric_shift ← candidate[numcandidates_forward];
           /* There is a mirror symmetry that maps vertex 0 to this hull vertex. */
       *has_fixed_vertex ← ((*rotation_period) % 2 ≡ 1) ∨ (symmetric_shift % 2 ≡ 0);
     }
   }
```

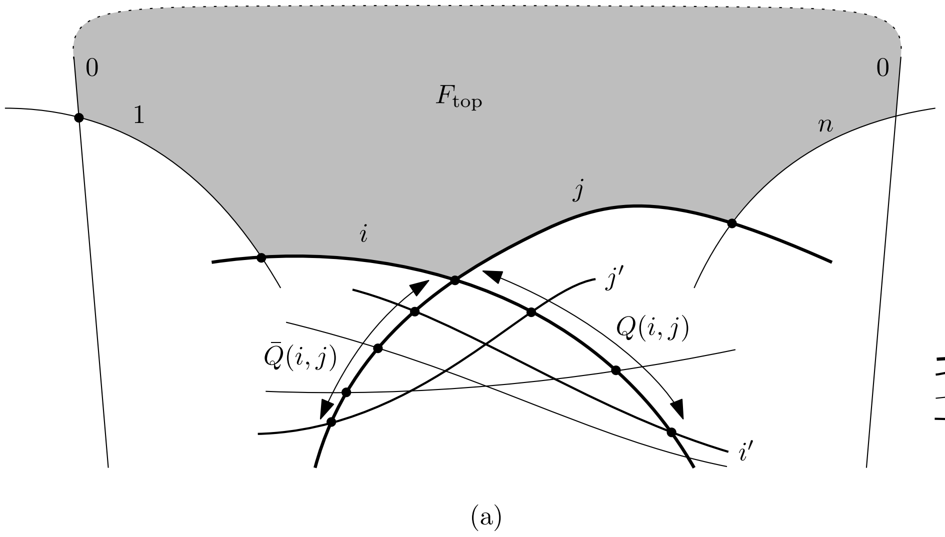
This code is used in chunk 43.

12 Screening of candidates to reduce the running time

Suppose we don't have correct labels, but we only known line 0 and line 1. We can still determine the upper right corner P_{1n} of the P-matrix, as follows (see Figure 3b).

We find $i' \leftarrow$ PRED$(0,1)$; This is line n. Then PRED$(i',0)$ would represent the last intersection on line n, which is the value of P_{1n} that we want, except that we don't have the correct label. We can recover this label by walking along line 0, using the SUCC labels, until we hit line i'.

Let i and j be two consecutive edges on the upper envelope. The quantity $Q(i,j)$ is defined as follows, see Figure 3a.



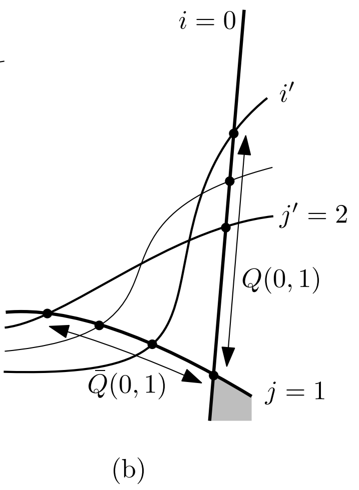


Figure 3: (a) An example with $Q(i,j) = 4$ and $\bar{Q}(i,j) = 5$; (b) an example with $Q(0,1) = \bar{Q}(0,1) = 4$

Let $i' =$ PRED(i,j). Walk on line i to the right (by SUCC) from the intersection between i and j until meeting the intersection with i'. Then $Q(i,j)$ is the number of visited points on i, including the endpoints. This convention ensures that $Q(i,j)$ is the value P_{1n} when line i is chosen to play the role of line 0, (and j will become line 1). In the walk along i, it may happen that we cross line 0 and wrap around from the right end to the left end.

The quantity $\bar{Q}(i,j)$ is defined with switched roles of i and j and with left and right exchanged, and it gives the value P_{1n} in the mirror situation (the *backward* direction) when line j is chosen to play the role of line 0: Let $j' =$ SUCC(i,j). Walk on line j to the left (by PRED) until meeting line j'.

We apply this definition two all pairs (i,j) of consecutive edges on the upper envelope, starting with $(0,1)$ and ending with $(n,0)$. (The last pair is the only pair with $i > j$.)

The numbers $Q(i,j)$ and $\bar{Q}(i,j)$ are between 2 and n, and $Q(i,j) = 2 \iff \bar{Q}(i,j) = 2$.

For $(i,j) = (0,1)$, the wedge between lines i and j appears actually at the bottom right of the wiring diagram, see Figure 3b. Here we have $Q(0,1) =$ PRED$(1,0) = P_{1n}$, since this is the original situation where line 0 is where it should be. Similarly, for $(i,j) = (n,0)$, we have to look at the bottom left corner.

Our primary criterion in the lexicographic comparison is P_{1n}. This is given by $Q(i,j)$ and $\bar{Q}(i,j)$ for the pairs (i,j) of consecutive edges on the upper envelope. This has to be compared against the current value of P_{1n}, which is $Q(0,1)$.

¶ Screen candidates by comparing the leading entry P_{1n}.

Compute the leading entry P_{1n} for all candidates directly, without first computing the *label_arrays*. The *label_arrays* are computed afterwards (if at all), and only those that are not yet eliminated.

Each of the h hull edges can be used as a starting edge in the forward direction or in the backward direction. This gives rise to h forward candidates, whose corresponding value P_{1n} is computed in the array *P_1_n_forward*[r] for $r = 0, \dots, h-1$, and h backward candidates, for which the array *P_1_n_reverse* is used. The value P_{1n} for the current solution, whose lex-minimality we are checking, is in *P_1_n_forward*[0]. If any other candidate has a smaller value *P_1_n_forward*[r] or *P_1_n_reverse*[r] than this, we can immediately abandon the current solution and **return** *false*.

This cannot occur if $P_{1n} = 2$ for line 0, but otherwise there is a high chance for finding a smaller value P_{1n} for some of the other candidates. [Observation. The relative frequence of P_{1n} over all PSLAs is about 26 % for 2 and n, about 11 % for 3 and $n-1$ and decreases towards the middle values. The symmetry can be explained as follows. A PSLA is essentially a projective oriented PSLA with a marked angle. Going to an adjacent angle and mirroring the PSLA exchanges a with $n+2-a$.]

If any candidate has a value *P_1_n_forward*[r] or *P_1_n_reverse*[r] larger than *P_1_n_forward*[0], that candidate can be excluded from further consideration.

This screening procedure saves about 20 % of the runtime for enumerating AOTs.

The following program uses the condition $Q(i,j) = 2 \iff \bar{Q}(i,j) = 2$ to shortcut the computation. (Not sure if it brings any advantage, because computing $\bar{Q}(i,j)$ would also be fast in this case.)

For example there are 18,410,581,880 PSLAs with $n = 10$ lines. Of these, only 5,910,452,118 pass the screening test. Eventually, only 2,343,203,071 PSLA are really lex-min, and this is the number of AOTs that we really want. The screening test seems to be more and more effective for larger n, but this is in line with the fact that the probability of being lex-smallest decreases, since it is close to $1/(2n+2)$.

47 ⟨Screening procedures 47⟩ ≡
```
boolean screen_lex_smallest(int n, int *hulledges, int hullsize)
{
  P_1_n_forward[0] ← PRED(1, 0);      /* because hulledges[1] ≡ 1 */
  for_int_from_to (r, 1, hullsize − 1) {
    int r_next ← (r + 1) % hullsize;
    int i ← hulledges[r];
    int j ← hulledges[r_next];      /* i or j plays the role of line 0 */
    int i' ← PRED(j, i);
    int a ← 2; int j2 ← SUCC(i, j);
    while (j2 ≠ i') {      /* compute a by running along i */
      j2 ← SUCC(i, j2);
      a++;
      if (a > P_1_n_forward[0]) break;      /* shortcut */
    }
    if (a < P_1_n_forward[0]) return false;
    P_1_n_forward[r] ← a;      /* This may not be the precise value if a > P_1_n_forward[0], but it is
          sufficient to exclude candidate r. */
  }
  for_int_from_to (r, 0, hullsize − 1) {
    int r_next ← (r + 1) % hullsize;
    if (P_1_n_forward[r] ≡ 2) {
      P_1_n_reverse[r_next] ← 2;
          /* The wedge between i and i is a triangle; Q(i, j) and Q̄(i, j) are both 2. */
      continue;
    }
    int i ← hulledges[r];
    int j ← hulledges[r_next];      /* i or j plays the role of line 0 */
    int j' ← SUCC(i, j);
    int a ← 2; int i2 ← PRED(j, i);
```

```
      do {      /* compute $a$ by running along $j$ */
        $i2 \leftarrow$ PRED($j, i2$);
        $a$++;
        if ($a > P\_1\_n\_forward[0]$) break;
      } while ($i2 \neq j'$);
      if ($a < P\_1\_n\_forward[0]$) return false;
      $P\_1\_n\_reverse[r\_next] \leftarrow a$;
    }
    return true;
  }
```
This code is used in chunk 43.

12.1 More aggressive screening at the next-to-last level $n-1$

We apply a test at level $n-1$, before the n-th pseudoline is inserted. If we find out that none of the PSLAs obtained by adding line n has a change of surviving the screening test, we can save a lot of time by not generating these PSLAs at all (rather than generating many PSLAs with n lines and eliminating them by screening).

When adding a new line n, the quantities $Q(i,j)$ can change in a few ways.

1. We cut off some hull vertices. In particular, $(n-1,0)$ will always disappear.
2. We generate two new hull vertices: (i,n) with $1 \le i \le n-1$, and $(n,0)$.
3. In the definition of $Q(i,j)$, line n could take the role of i'. (or j' in the case of $\bar{Q}(i,j)$).
4. In the definition of $Q(i,j)$, line n could intervene between the intersections with j and i' on line i, thus increasing $Q(i,j)$ by 1. (or a similar situation for $\bar{Q}(i,j)$).

A very rudimentary pre-screening test has been implemented, namely for the comparison between $Q(0,1)$ and $\bar{Q}(1,0)$:

If $\bar{Q}(0,1) < Q(1,0)-1$ in the arrangement with $n-1$ lines, then there is no chance to augment this to a lex-min PSLA.

Proof: See Figure 3b. There are two cases. If line n does not intersect the segment between 1×0 and $1 \times \texttt{PRED}(1,0)$, then $Q(0,1) = P_{1n}$ is unchanged. $\bar{Q}(1,0)$ can increase by at most 1. Thus $\bar{Q}(1,0)$ will beat $Q(1,0)$.

If line n intersects line 1 between 1×0 and $1 \times \texttt{PRED}(1,0)$, then n becomes the new $i' = \texttt{PRED}(1,0) = Q(0,1) = P_{1n}$, and thus P_{1n} has the maximum possible value, n, and is certainly larger than before. $\bar{Q}(1,0)$ can still increase by at most 1. Thus $\bar{Q}(1,0)$ will beat $Q(1,0)$.

For example, with $n=9$ lines there are 112,018,190 PSLAs, and they generate as children 18,410,581,880 PSLAs with $n=10$ lines, as mentioned above. The screening test at level $n=9$ eliminates 22,023,041 out of the 112,018,190 PSLAs (19.66%) because they are not able to produce a lex-min AOT in the next generation. The remaining 89,995,149 PSLAs produce 15,409,623,219 offspring PSLAs with $n=10$ lines, as opposed to 18,410,581,880 without this pruning procedure. These remaining PSLAs are subject to the screening as before.

This test takes only $O(n)$ time. It would make sense to run further pre-screening tests, even if they take much longer. We are running this test on level $n-1$. If a test can exclude the generation of hundreds of successor configurations, it is worth while. I have formulated some more powerful other tests, but have not implemented them, because they are getting more and more delicate.

```
49 ¶⟨Screen one level below level $n\_max$ 49⟩ ≡
  int $P\_1\_n \leftarrow$ PRED(1, 0);      /* insertion of last line $n$ can only make this larger. */
  if ($P\_1\_n > 3$) {
    int $a \leftarrow 2$;
    int $i2 \leftarrow P\_1\_n$;      /* $\equiv i'$ */
    while ($i2 \neq 2$) {      /* compute $a$ by running along $j \equiv 1$ */
      $i2 \leftarrow$ PRED($1, i2$);
      $a$++;
    }      /* Now $P\_1\_n\_reverse \equiv a$ but insertion of line $n$ could increase this by 1. */
    if ($a + 1 < P\_1\_n$) hopeful $\leftarrow$ false;
  }
  if (hopeful) cpass++; else csaved++;
```
This code is used in chunk 16.

¶ We maintain statistics about the effectiveness of this test:

```
50 ⟨ Global variables 5 ⟩ +≡
     long long unsigned cpass, csaved;
```

13 Statistics

Characteristics:

- number h of hull points.
- period p of rotational symmetry on the hull. (The order of the rotation group is h/p.)
- mirror symmetry, with or without fixed vertex on the hull (3 possibilities).

U_PSLAcount counts OAOT of point sets with a marked point on the convex hull, but no specified traversal direction. (U stands for unoriented.) Equivalently, it counts the x-monotone PSLAs when a PSLA is identified with its left-right mirror.

PSLAcount counts OAOT of point sets with a marked point on the convex hull and a specified traversal direction. Equivalently, it counts the x-monotone PSLAs, see http://oeis.org/A006245. This gives always the correct number, even if the program does not visit all these PSLAs due to the pre-screening.

```
51 #define NO_MIRROR 0
   #define MIRROR_WITH_FIXED_VERTEX 1
   #define MIRROR_WITHOUT_FIXED_VERTEX 2
   ⟨ Global variables 5 ⟩ +≡
     long long unsigned countPSLA[MAXN + 2], countO[MAXN + 2], countU[MAXN + 2];
     long long unsigned PSLAcount[MAXN + 2];      /* A006245, Number of primitive sorting networks on n
         elements; also number of rhombic tilings of 2n-gon. */
        /* 1, 1, 2, 8, 62, 908, 24698, 1232944, 112018190, 18410581880, 5449192389984 ... until n = 16. */
     long long unsigned U_PSLAcount[MAXN + 2];
     long long unsigned classcount[MAXN + 2][MAXN + 2][MAXN + 2][3];
     long long unsigned numComparisons ← 0, numTests ← 0;      /* profiling */
```

```
52 ¶⟨ Initialize statistics and open reporting file 52 ⟩ ≡
     countPSLA[1] ← countPSLA[2] ← 1;
     countO[3] ← countU[3] ← PSLAcount[2] ← U_PSLAcount[2] ← 1;
        /* All other counters are automatically initialized to 0. */
     if (strlen(fname)) {
       reportfile ← fopen(fname, "w");
     }
```

This code is used in chunk 6.

```
53 ¶ ⟨ Gather statistics about the AOT, collect output 53 ⟩ ≡
     int hulledges[MAXN + 1];
     int hullsize ← upper_hull_PSLA(n, hulledges);      /* Determine the extreme points: */
     int rotation_period;
     boolean has_fixed_vertex;
     boolean has_mirror_symmetry;
     int n_points ← n + 1;      /* number of points of the AOT */
     boolean lex_smallest ← has_lex_smallest_P_matrix(n, hulledges, hullsize, &rotation_period,
         &has_mirror_symmetry, &has_fixed_vertex);
     if (lex_smallest) {
       countU[n_points]++;      /* We count to contribution from this AOT to the various counters countO,
           PSLAcount, U_PSLAcount according to the symmetry information. */
       if (has_mirror_symmetry) {
         countO[n_points]++;
         PSLAcount[n] += rotation_period;
         if (has_fixed_vertex) U_PSLAcount[n] += rotation_period/2 + 1;
              /* works for even and odd rotation_period */
         else U_PSLAcount[n] += rotation_period/2;
```

```
    }
    else {
      countO[n_points] += 2;
      PSLAcount[n] += 2 * rotation_period;
      U_PSLAcount[n] += rotation_period;
    }
    classcount[n_points][hullsize][rotation_period][¬has_mirror_symmetry ? NO_MIRROR : has_fixed_vertex ?
        MIRROR_WITH_FIXED_VERTEX : MIRROR_WITHOUT_FIXED_VERTEX]++;
  }
#if 0    /* debugging */
  printf("found␣n=%d.␣%Ld␣", n_points, countO[n_points]);
  print_small(S, n_points);
#endif
```

This code is used in chunk 19.

¶ First some basic statistics are written in tabular form to the terminal:

54 ⟨Report statistics 54⟩ ≡

```
  printf("%34s%69s\n", "#PSLA␣visited␣by␣the␣program", "#PSLA␣computed␣from␣AOT");
  for_int_from_to (n, 3, n_max + 1) {
    long long symmetric ← 2 * countU[n] − countO[n];
    printf("n=%2d", n);
    if (split_level ≠ 0 ∧ n > split_level) printf("*,"); else printf(",␣");
    printf("#PSLA=%11Ld", countPSLA[n]);
#if enumAOT
    printf(",␣#AOT=%10Ld,␣#OAOT=%10Ld,␣#symm.␣AOT=%7Ld,␣", countU[n], countO[n], symmetric);
    printf("#PSLA=%11Ld,␣#uPSLA=%10Ld", PSLAcount[n], U_PSLAcount[n]);
#endif
    printf("\n");
  }
  if (split_level ≠ 0) printf("*␣Lines␣with␣\"*\"␣give␣results␣from␣partial␣enumeration.\n");
#if profile
  printf("Total␣tests␣is_lex_min␣(after␣screening)␣=␣%Ld,␣total␣comparisons␣=␣%Ld,␣averag\
      e␣=%6.3f\n", numTests, numComparisons, numComparisons/(double) numTests);
#endif
#if enumAOT
  printf("Prescreening:␣passed␣%Ld,␣saved␣%Ld␣out␣of␣%Ld␣=␣%.2f%%\n", cpass, csaved,
      cpass + csaved, 100 * csaved/(double)(cpass + csaved));
#endif
```

See also chunk 55.

This code is used in chunk 6.

¶ The statistics gathered in the *classcount* array are written to a *reportfile* so that a subsequent program can conveniently read and process them.

55 ⟨Report statistics 54⟩ +≡

```
  if (strlen(fname)) {
    fprintf(reportfile, "#␣N_max=%d/%d", n_max, n_max + 1);
    if (parts ≠ 1) fprintf(reportfile, ",␣split-level=%d,␣part␣%d␣of␣%d", split_level, part, parts);
    fprintf(reportfile, "\n#x␣N␣hull␣period␣mirror-type␣␣NUM\n");
    for_int_from_to (n, 0, n_max + 1) {
      char c ← 'T';    /* total count */
      if (parts ≠ 1 ∧ n > split_level + 1) c ← 'P';    /* partial count */
      for_int_from_to (k, 0, n)
        for_int_from_to (p, 0, n)
          for_int_from_to (t, 0, 2)
            if (classcount[n][k][p][t])
              fprintf(reportfile, "%c␣%d␣%d␣%d␣%d␣␣%Ld\n", c, n, k, p, t, classcount[n][k][p][t]);
    }
    if (parts ≡ 1) fprintf(reportfile, "EOF\n");
    else fprintf(reportfile, "EOF␣%d,␣part␣%d␣of␣%d\n", split_level, part, parts);
    fclose(reportfile);
    printf("Results␣have␣been␣written␣to␣file␣%s.\n", fname);
  }
```

13.1 Mirror symmetries of PSLAs

For $n = 6$, there are 908 PSLAs (as accumulated in *PSLAcount*[6]), but only 461 unoriented PSLAs ("uPSLAs"), as accumulated in *U_PSLAcount*[6].

From this we conclude that among the 908 PSLAs, there must be 14 PSLAs that have a vertical symmetry axis, and the remaining 894 come in 447 mirror-symmetric pairs, because $2 \times 447 + 14 = 908$ and $447 + 14 = 461$.

We also know that there must be the same number, 14, of PSLAs with a horizontal symmetry axis, because n is even, and an appropriate rotation swaps the directions. A PSLA cannot have both a vertical and a horizontal symmetry axis, because then it would allow a 180° rotation, and this is impossible: Consider the cross formed by lines 1 and n. The crossing of lines 2 and $n - 1$ is in one of the four sectors of this cross, and after rotation, it is in the opposite sector.

Similarly, for $n = 7$, the 24698 PSLAs split into 12270 pairs without mirror symmetry and 158 symmetric ones, since $2 \times 12270 + 158 = 24698$ and $12270 + 158 = 12428$, which is the number of uPSLAs. In this case, there are no PSLAs with a horizontal symmetry axis, because n is odd: The "middle" line must pass either above or below the crossing $1 \times n$, and this property is inverted by a reflection at a horizontal axis.

A consequence of these considerations is that the number of PSLAs is always even, because they can be grouped into pairs that are either vertically symmetric or horizontally symmetric.

14 Special problem-specific extensions

Program extensions for special purposes can be added here: The following data are available:

lex_smallest ...

The AOT has $n + 1$ points; its convex hull has *hullsize* vertices and is stored in the array *hulledges*. ....

lex_smallest

in the *succ* and *pred* arrays

P-matrix is / is not available.

After computing the inverse P-matrix, one can perform a few tests on the order type, using orientation queries, see for example Section 15.3.

14.1 Further exclusion criteria

If some PSLAs or AOTs and their subtrees should not be considered, they can be filtered here, by setting *is_excluded* to *false*.

58 ⟨Check for exclusion and set the flag *is_excluded* 22⟩ +≡ /* Currently no further exclusion tests. */

14.2 Further processing of AOTs

Problem-specific processing can be added here.

59 ⟨Further processing of the AOT 59⟩ ≡ /* Currently no further processing of the AOT. */

See also chunks 60, 62, 63, and 69

This code is used in chunk 19.

14.2.1 Listing all PSLAs

List all PSLAs plus their IDs, as preparation for generating exclude-files of nonrealizable AOTs.

60 ⟨Further processing of the AOT 59⟩ +≡

```
#if generatelist
  if (n ≡ n_max ∧ lex_smallest) {
    ⟨Print PSLA-fingerprint 36⟩
    print_id(n);
    printf("\n");
  }
#endif
```

14.2.2 Various further test programs

¶ Print “some” example.

```
62 ⟨Further processing of the AOT 59⟩ +≡
#if 0
  if ($n ≡ n\_max ∧ countPSLA[n] ≡ 50$) {       /∗ print some arbitrary example ∗/
    P_matrix $P$, $\bar{P}$;
    convert_to_P_matrix(&$P$, $n$);
    convert_to_inverse_P_matrix(&$\bar{P}$, $n$);
    print_pseudolines_short(&$P$, $n$);
    printf("inverse␣");
    print_pseudolines_short(&$\bar{P}$, $n + 1$);
    print_wiring_diagram($n$);
  }
#endif
```

¶ Estimate the size of possible subproblems for a divide-&conquer Ansatz.

```
63 ⟨Further processing of the AOT 59⟩ +≡
#if 0     /∗ estimate size of possible subproblems for divide-&conquer Ansatz ∗/
#define MID 5
  if ($n ≡ 2 ∗$ MID $− 2$) {
    P_matrix $P$;
    convert_to_P_matrix(&$P$, $n$);
    for_int_from_to ($i$, 2, MID $− 1$) {
      boolean show ← true;
      for_int_from_to ($j$, 1, $n − 1$) {
        int $x ← P[i][j]$;
        if ($x ≡$ MID $∨ x ≡ 1$) break;
        printf("%c", TO_CHAR($x$));
      }
      printf("!");
    }
    for_int_from_to ($i$, MID $+ 1$, $n$) {
      boolean show ← false;
      for_int_from_to ($j$, 1, $n − 1$) {
        int $x ← P[i][j]$;
        if (show) printf("%c", TO_CHAR($x$));
        if ($x ≡$ MID) show ← true;
        if ($x ≡ 1$) break;
      }
      printf($i < n$ ? "!" : "␣");
    }
    for_int_from_to ($j$, 1, $n − 1$) {
      int $x ← P[1][j]$;
      if ($x ≡$ MID) break;
      printf("%c", TO_CHAR($x$));
    }
    printf("!");
    for_int_from_to ($j$, 1, $n − 1$) {
      int $x ← P$[MID][$j$];
      if ($x ≡ 1$) break;
      printf("%c", TO_CHAR($x$));
    }
    printf("\n");
  }
#endif
```

15 Other representations of abstract order types: λ-matrices and and Λ-matrices

¶ More type definitions.

```
65 ⟨ Types and data structures 4 ⟩ +≡
  typedef boolean large_matrix_entry;
  typedef unsigned small_matrix_entry;
  typedef small_matrix_entry small_lambda_matrix[MAXN + 1][MAXN + 1];
  typedef large_matrix_entry large_Lambda_matrix[MAXN + 1][MAXN + 1][MAXN + 1];
```

15.1 (“Small”) λ-matrices

Input: PSLA with n lines $1 \mathinner{..} n$ plus line 0 “at ∞”. Output: “small” λ-matrix B for AOT on $n+1$ points. Line at ∞ corresponds to point 0 on the convex hull.

```
66 #define entry_small(A, i, j) (A)[i][j]
⟨ Subroutines 27 ⟩ +≡
  void convert_to_small_lambda_matrix(small_lambda_matrix *B, int n)
  {
    for_int_from_to (i, 0, n) {
      (*B)[i][i] ← 0;
    }
    for_int_from_to (i, 1, n) {
      int level ← i − 1;    /* number of lines above the crossing */
      (*B)[0][i] ← level;
      (*B)[i][0] ← n − 1 − level;
      int j ← SUCC(i, 0);
      while (j ≠ 0) {
        if (i < j) {
          (*B)[i][j] ← level;
          level++;
        }
        else {
          level−−;
          (*B)[i][j] ← n − 1 − level;
        }
        j ← SUCC(i, j);
      }
    }
  }
```

15.2 (“Large”) Λ-matrices

The three-dimensional Λ-matrix stores the orientation of all triples.

It would be possible to save space by a more elaborate indexing function into a one-dimensional array, storing entries Λ_{ijk} only for $i < j < k$. More general access could then be provided by a macro *get_entry_Lambda*.

We have the natural labeling around the *pivot* point, which is assumed to lie on the convex hull.

¶ Generate the Λ-matrix. Only for testing purposes. Assumes natural ordering. Assumes general position. Works by plucking points from the convex hull one by one. The input is a λ-matrix A. The result is stored in B. The entries Λ_{ijk} are not set if the indices i, j, k are not distinct.

```
68 ⟨ Subroutines 27 ⟩ +≡
  void copy_small(small_lambda_matrix *A, small_lambda_matrix *B, int n)
  {
    for_int_from_to (i, 0, n − 1)
      for_int_from_to (j, 0, n − 1) entry_small(*B, i, j) ← entry_small(*A, i, j);
  }
```

```
void convert_small_to_large(small_lambda_matrix *A, large_Lambda_matrix *B, int n)
{
  small_lambda_matrix Temp;
  copy_small(A, &Temp, n);      /* the small matrix Temp will be destroyed */
  for_int_from_to (k, 0, n − 1)
    for_int_from_to (i, k + 1, n − 1)
      for_int_from_to (j, i + 1, n − 1)      /* $k < i < j$ */
      {
        boolean plus ← entry_small(Temp, i, k) < entry_small(Temp, j, k);
        (*B)[k][i][j] ← (*B)[i][j][k] ← (*B)[j][k][i] ← plus;
        (*B)[k][j][i] ← (*B)[i][k][j] ← (*B)[j][i][k] ← ¬plus;
        if (plus) entry_small(Temp, i, j)−−;
        else entry_small(Temp, j, i)−−;
      }
}
```

15.3 Checking correctness of the orientation test

The following test program computes the inverse P-matrix and then exhaustively compares the orientation queries against an explicitly computed three-dimensional Λ-matrix (see Section 15.2).

69 ⟨Further processing of the AOT 59⟩ +≡
```
#if 0
  P_matrix $\bar{P}$;      /* the orientation test is computed from this array. */
  convert_to_inverse_P_matrix(&$\bar{P}$, n);
  small_lambda_matrix S;
  convert_to_small_lambda_matrix(&S, n_points);
  large_Lambda_matrix L;
  convert_small_to_large(&S, &L, n_points);
  ⟨Compare orientation tests 70⟩
#endif
```

70 ¶⟨Compare orientation tests 70⟩ ≡
```
  for_int_from_to (i, 0, n_points − 1)
    for_int_from_to (j, 0, n_points − 1)
      for_int_from_to (k, 0, n_points − 1)
        if ($i \neq j \wedge k \neq j \wedge k \neq i$)
          if (getOrientation($\bar{P}$, i, j, k) ≠ L[i][j][k]) {
            printf("[%d,%d,%d]=%d!=%d\n", i, j, k, getOrientation($\bar{P}$, i, j, k), L[i][j][k]);
            exit(1);
          }
```
This code is used in chunk 69.

16 Reading from the Order-Type Database

For simplicity, we work only with numbers in the 16-bit format. Inputs in 8-bit formats are converted.

71 ⟨Global variables 5⟩ +≡
```
  struct {      /* 16-bit unsigned coordinates: */
    uint16_t x, y;
  } points[MAXN + 1];
  struct {      /* 8-bit unsigned coordinates: */
    uint8_t x, y;
  } pointsmall[MAXN + 1];
```

16.1 Orientation test for points

The return value of *orientation_test* is positive for counterclockwise orientation of the points i, j, k.

72 ⟨ Subroutines 27 ⟩ +≡
```
large_int orientation_test(int $i$, int $j$, int $k$)
{
  large_int $a \leftarrow points[j].x - $ (large_int) $points[i].x$;      /* range $-65535 \mathinner{\ldotp\ldotp} 65535$ */
  large_int $b \leftarrow points[j].y - $ (large_int) $points[i].y$;
  large_int $c \leftarrow points[k].x - $ (large_int) $points[i].x$;
  large_int $d \leftarrow points[k].y - $ (large_int) $points[i].y$;
  return $a * d - b * c$;
}
```

¶ Intermediate results can be almost 2^{32} in absolute value, and they have signs. The final value is the signed area of the parallelogram spanned by 3 points. Thus it can also be almost 2^{32} in absolute value. 32 bits are not enough to be safe. We use 64 bits.

73 ⟨ Types and data structures 4 ⟩ +≡
```
typedef int_least64_t large_int;      /* for intermediate calculations */
```

16.2 Turn point set with coordinates into PSLA

We insert the lines one by one into the arrangement. This is similar to the insertion of line n in the recursive enumeration procedure *recursive_generate_PSLA* of Section 4. The difference is that we don't try all possibilities for the edge through which line n exits, but we choose the correct edge the by orientation test. By the zone theorem, the insertion of line n takes $O(n)$ time.

We have n points. The first point (point 0) is on the convex hull and the other points are sorted around this point. We get a PSLA with $n-1$ pseudolines, which correspond to points $1, \ldots, n-1$ in the order in which they are given.

74 ⟨ Subroutines 27 ⟩ +≡
```
void insert_line(int $n$);
void PSLA_from_points(int $n$)
{
  LINK(1, 0, 2);
  LINK(1, 2, 0);
  LINK(2, 0, 1);
  LINK(2, 1, 0);
  LINK(0, 1, 2);
     /* LINK(0, 2, 3) and LINK(0, 3, 1) will be established shortly in the first recursive call. */
  for_int_from_to ($i, 3, n-1$) insert_line($i$);
}
void insert_line(int $n$)
{
  LINK(0, $n-1, n$);
  LINK(0, $n$, 1);
  int $entering\_edge \leftarrow 0,\ j \leftarrow 0,\ j^+ \leftarrow 0$;
  int $k_{\text{left}},\ k_{\text{right}}$;
  while (1) {
    while ($j^+ > j$) {      /* find right vertex of the cell */
      int $j^+_{\text{old}} \leftarrow j^+$;
      $j^+ \leftarrow$ SUCC($j^+, j$);
      $j \leftarrow j^+_{\text{old}}$;
    }
    if ($j^+ \equiv 0$) {      /* $F$ is unbounded */
      if ($j \equiv n-1$) {      /* $F$ is the top face. */
        LINK($n$, $entering\_edge$, 0);      /* complete the insertion of line $n$ */
        return;
      }
      $j^+ \leftarrow j+1$;      /* jump to the upper ray of $F$ */
      $j \leftarrow 0$;
    }      /* Now the crossing $j \times j^+$ is the rightmost vertex of the face $F$. $j^+$ is on the upper side, and if $F$
             is bounded, $j$ is on the lower side, */
    do {     /* scan the upper edges of $F$ from right to left and find the correct one to cross. */
      $k_{\text{right}} \leftarrow j$;
      $j \leftarrow j^+$;
      $k_{\text{left}} \leftarrow j^+ \leftarrow$ PRED($j, k_{\text{right}}$);
    } while ($j^+ > j \wedge orientation\_test(j, k_{\text{left}}, n) > 0$);
    LINK($j$, $k_{\text{left}}, n$);     /* insert crossing with $n$ on line $j$ */
    LINK($j$, $n, k_{\text{right}}$);
    LINK($n$, $entering\_edge, j$);
    $entering\_edge \leftarrow j$;
    $j^+ \leftarrow k_{\text{right}}$;
  }
}
```

16.3 Select the order-type files to be read

We have to figure out the filenames and the format of the stored numbers. We assume that the order types with up to 10 points are stored in the current directory in with the original file names `otypes10.b16`, `otypes09.b16`, `otypes08.b08`, etc., and the order types with 11 points are stored in a subdirectory `Ordertypes` with names `Ordertypes/ord11_00.b16` ... `Ordertypes/ord11_93.b16`.

75 ⟨Include standard libaries 10⟩ +≡

```
#include <fcntl.h>
#include <unistd.h>
```

76 ¶⟨Subroutines 27⟩ +≡

```
void swap_all_bytes(int n)
{     /* convert numbers from little-endian to big-endian format. */
  for_int_from_to (i, 0, n − 1) {
    points[i].x ← (points[i].x ≫ 8) | (points[i].x ≪ 8);
    points[i].y ← (points[i].y ≫ 8) | (points[i].y ≪ 8);
      /* Assumes 16 bits. It is important that coordinates are unsigned. */
  }
}
```

77 ¶⟨Read all point sets of size $n\_max + 1$ from the database and process them 77⟩ ≡

```
int n_points ← n_max + 1;
int bits ← n_points ≥ 9 ? 16 : 8;
char inputfile[60];
int record_size ← (bits/8) ∗ 2 ∗ n_points;
printf("Reading␣order␣types␣of␣%d␣points\n", n_points);
printf(".\n");
printf("One␣record␣is␣%d␣bytes␣long.\n", record_size);
boolean is_big_endian ← ( ∗ ( uint16_t ∗ ) "\0\xff" < #100 ) ;
if (bits > 8) {
  if (is_big_endian) printf("This␣computer␣is␣big-endian.\n");
  else printf("This␣computer␣is␣little-endian.␣No␣byte␣swaps␣are␣necessary.\n");
}
if (n_points < 11) {
  snprintf(inputfile, 60, "otypes%02d.b%02d", n_points, bits);
  read_database_file(inputfile, bits, record_size, n_points, is_big_endian);
}
else
  for_int_from_to (num_db, 0, 93) {
    snprintf(inputfile, 60, "Ordertypes/ord%02d_%02d.b16", n_points, num_db);
    read_database_file(inputfile, bits, record_size, n_points, is_big_endian);
  }
printf("%Ld␣point␣sets␣were␣read␣from␣the␣file(s).\n", read_count);
```

This code is used in chunk 6.

16.4 Do the actual reading

Open and read database file and process the input points.

```
78 ⟨ Subroutines 27 ⟩ +≡
  long long unsigned read_count ← 0;
  void read_database_file(char *inputfile, int bits, int record_size, int n_points, boolean is_big_endian)
  {
    printf("Reading␣from␣file␣%s\n", inputfile);
    int databasefile ← open(inputfile, O_RDONLY);
    if (databasefile ≡ −1) {
      printf("File␣could␣not␣be␣opened.\n");
      exit(1);
    }
    while (1) {
      ssize_t bytes_read;
      if (bits ≡ 16) bytes_read ← read(databasefile, &points, record_size);
      else bytes_read ← read(databasefile, &pointsmall, record_size);
      if (bytes_read ≡ 0) break;
      if (bytes_read ≠ record_size) {
        printf("Incomplete␣file.\n");
        exit(1);
      }
      read_count++;
      if (bits ≡ 16 ∧ is_big_endian) swap_all_bytes(n_points);
      if (bits ≡ 8)
        for_int_from_to (i, 0, n_points − 1) {
          points[i].x ← pointsmall[i].x;
          points[i].y ← pointsmall[i].y;
        }
      int n ← n_points − 1;
      PSLA_from_points(n_points);
      int hulledges[MAXN + 1];
      int hullsize ← upper_hull_PSLA(n, hulledges);
      P_matrix P;
      compute_lex_smallest_P_matrix(&P, n, hulledges, hullsize);
      compute_fingerprint(&P, n);
      printf("%s:\n", fingerprint);
    }
    close(databasefile);
  }
```

17 Timings

On my five-year-old Laptop (as of 2023), `NumPSLA 9` took 8 seconds to determine, among other quantities, the number $B_9 = 112018190$ of pseudoline arrangements with 9 lines.

There is a program *REFLECT* by Donald Knuth, to accompany the enumerations in his book *Axioms and Hulls* [?], see https://www-cs-faculty.stanford.edu/~knuth/programs.html. This program computes the quantities B_n, C_n, D_n, E_n that are defined in the book. The stripped-down version which computes only the quantities B_n took 37 seconds. (When Knuth ran it in 1991, without compiler optimizations and with the `-g` compiler option, the same program took nearly 20 hours.)

18 Things to consider

1. The `-exclude` option does not currently work with the parallelization through *splitlevel*. (This combination of input parameters is checked.)
2. Does the enumeration of PSLAs work in constant amortized time (CAT)? Test this experimentally by a loop counter.

3. Enumerate PSLAs for which the corresponding AOT has a given symmetry. In connection with the PSLAs without regard to symmetries, which are known up to 16 lines (17 points), this would lead to counts of AOTs with up to 17 points without much computational effort. (The current record is 13 points).
4. Projective types of PSLAs, projective AOTs.
5. Better drawings of PSLAs.
6. Selective exploration of subtrees. Goal-directed search for particular examples. Can be implemented by definining further exclusion criteria, see Section 14.1.
7. *Entropy encoding* of PSLAs?
8. Using inverse-PSLA makes *screening* slower! It might however be good in the context of screening one level before the last! Computing *inverse_PSLA* one level before *max_n* costs almost nothing.
9. The *succ* and *pred* arrays could be implemented as one-dimensional arrays, accessing them as $\texttt{SUCC}(i,j) \equiv \mathit{succ}[(i) \ll 4 \mid (j)]$. On some computers, this 1d option was clearly slower, by about 10 %. On others, there was only a small variation, less than the variation between runs of the same program.

18.1 Potential speed-ups

1. In *has_lex_smallest_P_matrix*, special treatment of the first row of P to avoid treating P_{1n} again (which has already been checked during screening). (Minor)
2. More elaborate pre-screening.

¶

Contents

Index

List of Refinements