November 24, 2020 at 13:23

1. Intro. This program finds all of the nonisomorphic graceful labelings of the graph $K_m \square P_3$. It was inspired by the paper of B. M. Smith and J.-F. Puget in *Constraints* **15** (2010), 64–92, where Table 5 reports a unique solution for $m = 6$. I'm writing it because I want to gain experience, gracefulnesswise — and also because Smith and Puget have unfortunately lost all records of the solution!

The graph $K_m \square P_3$ is "hardwired" into the logic of this program. It has $q = 3\binom{m}{2} + 2m$ edges; that's (7, 15, 26, 40, 57, 77, ...) for $m =$ (2, 3, 4, 5, 6, 7, ...). I doubt if I'll be able to reach $m = 7$; but I see no reason to exclude that case, because the algorithm needs very little memory.

Please excuse me for writing this in a rush.

```
#define m 6 /* the size of the cliques; must be at least 2 and at most 12 */
#define q ((m * (3 * m + 1))/2) /* number of edges */
#define o mems++ /* count one mem */
#define oo mems += 2 /* count two mems */
#define ooo mems += 3 /* count three mems */
#define delta 10000000000; /* report progress every delta or so mems */
#define O "%" /* used for percent signs in format strings */
#define mod % /* used for percent signs denoting remainder in C */
#define board(i,j) brd[3 * (i) + (j)]
#define leftknown colknown[0]
#include <stdio.h>
#include <stdlib.h>
  unsigned long long mems; /* memory accesses */
  unsigned long long thresh = delta; /* time for next progress report */
  unsigned long long nodes; /* nodes in the search tree */
  unsigned long long nulls; /* nodes that need no new vertex placement */
  unsigned long long leaves; /* nodes that have no descendants */
  int count; /* number of solutions found so far */
  int brd[3 * m]; /* one-dimensional array accessed via the board macro */
  int rank; /* how many rows of the board are active? */
  int labeled[q + 1]; /* what row and column, if any, have a particular label? */
  int placed[q + 1]; /* has this edge been placed? */
  int colknown[3]; /* how many vertices of each clique are labeled? */
  int move[q][1024]; /* feasible moves at each level */
  int deg[q]; /* number of choices at each level; used in printouts only */
  int x[q]; /* indexes of moves made at each level */
  int maxl; /* maximum level reached */
  int vbose = 0; /* can set this nonzero when debugging */
```

⟨ Subroutines 3 ⟩

```
main()
{
  register int a, b, i, j, k, l, t, v, aa, bb, ii, row, col, ccol, val, mv, trouble;
  fprintf(stderr,"---␣Graceful␣labelings␣of␣K"O"d␣times␣P3␣---\n",m);
  ⟨ Initialize the data structures 2 ⟩;
  ⟨ Backtrack through all solutions 9 ⟩;
  fprintf(stderr,"Altogether␣"O"d␣solution"O"s,",count,count ≡ 1 ? "" : "s");
  fprintf(stderr,"␣"O"lld␣mems,␣"O"lld-"O"lld␣nodes,␣"O"lld␣leaves;",mems,nodes,nulls,
      leaves);
  fprintf(stderr,"␣max␣level␣"O"d.\n",maxl);
  if (sanity_checking) fprintf(stderr,"sanity_checking␣was␣on!\n");
}
```

2. The current status of the vertices labeled so far appears in the *board*, which has three columns and m rows. This is not a canonical representation: The rows can appear in any order. When a vertex is unlabeled, the *board* has -1. When the vertex in row i and column j receives label l, $labeled[l]$ records the value $(j \ll 4) + i$; but $labeled[l]$ is -1 if that label hasn't been used. If both endpoints of an edge are labeled, and if d is the difference between those labels, $placed[d] = 1$; but $placed[d] = 0$ if no edge for difference d is yet known.

The first *rank* rows of the board have been labeled, at least in part.

```
⟨ Initialize the data structures 2 ⟩ ≡
  for (i = 0; i < m; i++)
    for (j = 0; j < 3; j++) board(i, j) = −1;
  rank = 0;
  for (l = 0; l ≤ q; l++) labeled[l] = −1;
  l = 0;
```

This code is used in section 1.

3.

```
⟨ Subroutines 3 ⟩ ≡
  void print_board(int rank)
  {
    register int i, j;
    for (i = 0; i < rank; i++) {
      for (j = 0; j < 3; j++)
        if (board(i, j) ≥ 0) fprintf(stderr, ""O"3d", board(i, j));
        else fprintf(stderr, "␣␣?");
      fprintf(stderr, "\n");
    }
  }
```

See also sections 4, 5, 11, 13, 14, and 20.

This code is used in section 1.

4.

```
⟨ Subroutines 3 ⟩ +≡
  void print_placed(void)
  {
    register int k;
    for (k = 1; k ≤ q; k++) {
      if (placed[k]) {
        if (¬placed[k − 1]) fprintf(stderr, "␣"O"d", k);
        else if (k ≡ q ∨ ¬placed[k + 1]) fprintf(stderr, ".."O"d", k);
      }
    }
    fprintf(stderr, "\n");
  }
```

5. These data structures are somewhat fancy, so I'd better check that they're self-consistent.

```
#define sanity_checking 0 /* set this to 1 if you suspect a bug */
⟨ Subroutines 3 ⟩ +≡
  void sanity(void)
  {
    register int i, j, l, t, v;
    ⟨ Check the rank 6 ⟩;
    ⟨ Check the labels 7 ⟩;
    ⟨ Check the placements 8 ⟩;
  }
```

6. ⟨ Check the rank 6 ⟩ ≡

```
  for (i = rank; i < m; i++) {
    if (board(i, 0) ≥ 0) break;
    if (board(i, 1) ≥ 0) break;
    if (board(i, 2) ≥ 0) break;
  }
  if (i < m ∨ rank > m) fprintf(stderr, "rank␣shouldn't␣be␣"O"d!\n", rank);
```

This code is used in section 5.

7. ⟨ Check the labels 7 ⟩ ≡

```
  for (l = 0; l ≤ q; l++) {
    v = labeled[l];
    if (v ≥ 0 ∧ board(v & #f, v ≫ 4) ≠ l) fprintf(stderr, "labeled["O"d]␣not␣on␣the␣board!\n", l);
  }
  for (i = 0; i < rank; i++)
    for (j = 0; j < 3; j++) {
      if (board(i, j) > q) fprintf(stderr, "board("O"d,"O"d)␣out␣of␣range!\n", i, j);
      if (board(i, j) ≥ 0 ∧ labeled[board(i, j)] ≠ (j ≪ 4) + i)
        fprintf(stderr, "label␣of␣board("O"d,"O"d)␣is␣wrong!\n", i, j);
    }
```

This code is used in section 5.

8.

```
#define testedge(i, j, ii, jj)
    if (board(i, j) ≥ 0 ∧ board(ii, jj) ≥ 0)
      if (t--, ¬placed[abs(board(i, j) − board(ii, jj))])
        fprintf(stderr, "edge␣from␣("O"d,"O"d)␣to␣("O"d,"O"d)␣not␣placed!\n", i, j, ii, jj);
```

⟨ Check the placements 8 ⟩ ≡

```
  for (t = 0, l = 1; l ≤ q; l++) t += placed[l];
  for (i = 0; i < rank; i++) {
    testedge(i, 0, i, 1);
    testedge(i, 1, i, 2);
    for (j = i + 1; j < rank; j++) {
      testedge(i, 0, j, 0);
      testedge(i, 1, j, 1);
      testedge(i, 2, j, 2);
    }
  }
  if (t) fprintf(stderr, "placement␣count␣off␣by␣"O"d!\n", t);
```

This code is used in section 5.

9. At level l of the backtrack procedure I try to place the edge whose difference is $q-l$, if that edge hasn't already been placed.

Initially there are four symmetries in addition to the $m!$ permutations of the rows of the board: We can interchange the left and right cliques; that's called reflection. We can also complement each label, replacing l by $q-l$.

I've set up the levels near the root so that complementation symmetry is avoided.

Reflection symmetry will disappear as soon as *leftknown* becomes nonzero. (After that happens, the board implicitly has $(m-rank)!$ symmetries.)

```
⟨ Backtrack through all solutions 9 ⟩ ≡
enter: nodes++;
  if (mems ≥ thresh) {
    thresh += delta;
    print_progress(l);
  }
  if (sanity_checking) sanity( );
  if (l ≤ 1) ⟨ Make special moves near the root 15 ⟩;
  if (l ≥ maxl) {
    maxl = l;
    if (l ≡ q) ⟨ Report a solution and goto backup 10 ⟩;
  }
  if (o, placed[q − l]) ⟨ Record the null move and goto ready 12 ⟩;
  for (t = a = 0, b = q − l; b ≤ q; a++, b++) ⟨ Record all possible (a, b) moves in the array move[l] 18 ⟩;
ready: deg[l] = t;      /* no mems counted for diagnostics */
  if (¬t) leaves++;
tryit: if (t ≡ 0) goto backup;
advance: if (vbose) {
    fprintf(stderr, "L"O"d:␣", l);
    print_move(move[l][t − 1]);
    fprintf(stderr, "␣("O"d␣of␣"O"d)\n", deg[l] − t + 1, deg[l]);
  }
  o, x[l] = −−t;
  o, mv = move[l][t];
  ⟨ Make mv 16 ⟩;
  if (trouble) {
    if (vbose) fprintf(stderr, "␣--␣was␣bad\n");
    goto unmake;
  }
  l++;
  goto enter;
backup: if (−−l ≥ 0) {
    o, t = x[l];
  unmake: o, mv = move[l][t];
    ⟨ Unmake mv 17 ⟩;
    goto tryit;
  }
```

This code is used in section 1.

10. ⟨ Report a solution and **goto** *backup* 10 ⟩ ≡

```
{
  count++;
  printf(""O"d:\n", count);
  for (i = 0; i < m; i++) printf(""O"3d"O"3d"O"3d\n", board(i, 0), board(i, 1), board(i, 2));
  goto backup;
}
```

This code is used in section 9.

11. ⟨ Subroutines 3 ⟩ +≡

```
void print_progress(int level)
{
  register int l, k, d, c, p;
  register double f, fd;
  fprintf(stderr, "␣after␣"O"lld␣mems:␣"O"d␣sols,", mems, count);
  for (f = 0.0, fd = 1.0, l = 0; l < level; l++) {
    d = deg[l], k = d − x[l];
    fd *= d, f += (k − 1)/fd;      /* choice l is k of d */
    fprintf(stderr, "␣"O"c"O"c", k < 10 ? '0' + k : k < 36 ? 'a' + k − 10 : k < 62 ? 'A' + k − 36 : '*',
        d < 10 ? '0' + d : d < 36 ? 'a' + d − 10 : d < 62 ? 'A' + d − 36 : '*');
  }
  fprintf(stderr, "␣"O".5f\n", f + 0.5/fd);
}
```

12. A "move" consists of labeling 0, 1, or 2 vertices and updating the data structures. A 16-bit packed entry, consisting of column number (4 bits), row number (4 bits), and label value (8 bits), specifies what labeling should be done. If two 16-bit entries are present, the rightmost one is done first.

It turns out that (row, col, val) will never be simultaneously zero. Hence an all-zero move means "do nothing."

#define *pack*(row, col, val) $(((col) \ll 12) + ((row) \ll 8) + (val))$

⟨ Record the null move and **goto** *ready* 12 ⟩ ≡

```
{
  o, move[l][0] = 0, t = 1, nulls++;
  goto ready;
}
```

This code is used in section 9.

13. ⟨ Subroutines 3 ⟩ +≡

```
void print_move(int mv)
{
  if (¬mv) fprintf(stderr,"null");
  else if (mv < #10000)
    fprintf(stderr,""O"d"O"d="O"d",(mv ≫ 8) & #f,(mv ≫ 12) & #f,mv & #ff);
  else fprintf(stderr,""O"d"O"d="O"d,"O"d"O"d="O"d",(mv ≫ 8) & #f,(mv ≫ 12) & #f,
       mv & #ff,(mv ≫ 24) & #f,(mv ≫ 28) & #f,(mv ≫ 16) & #ff);
}
void print_moves(int level)
{
  register int i;
  for (i = deg[level] − 1; i ≥ 0; i−−) {      /* we try the moves in decreasing order */
    fprintf(stderr,""O"d:",deg[level] − i);
    print_move(move[level][i]);
    fprintf(stderr,"\n");
  }
}
```

14. ⟨ Subroutines 3 ⟩ +≡

```
void print_state(int levels)
{
  register int l;
  for (l = 0; l < levels; l++) {
    print_move(move[l][x[l]]);
    fprintf(stderr,"␣("O"d␣of␣"O"d)\n",deg[l] − x[l],deg[l]);
  }
}
```

15. The edge labeled q must have endpoints labeled 0 and q. This can happen in only three essentially different ways: That edge either belongs to the middle clique, the left clique, or joins the left and middle cliques. In the latter case, complement symmetry has been broken. In the former cases, complement symmetry is avoided by insisting that the edge labeled $q-1$ has endpoints labeled 1 and q.

⟨ Make special moves near the root 15 ⟩ ≡

```
if (l ≡ 0) {
  o,move[0][0] = (pack(1,1,0) ≪ 16) + pack(0,1,q);
  o,move[0][1] = (pack(1,0,0) ≪ 16) + pack(0,0,q);
  o,move[0][2] = (pack(0,1,0) ≪ 16) + pack(0,0,q);
  t = 3;
  goto ready;
} else if (o,x[0] ≠ 2) {
  t = (m ≡ 2 ? 1 : 2);
  o,move[1][0] = pack(0,x[0],1);
  if (m > 2) o,move[1][1] = pack(2,1 − x[0],1);
  goto ready;
}
```

This code is used in section 9.

16. I set *trouble* nonzero if any edge is placed more than once.

```
⟨ Make mv 16 ⟩ ≡
  for (trouble = 0; mv; mv >>= 16) {
    val = mv & #ff, row = (mv ≫ 8) & #f, col = (mv ≫ 12) & #f;
    o, labeled[val] = (mv ≫ 8) & #ff;
    o, board(row, col) = val;
    oo, colknown[col]++;
    if (col > 0) {
      o, v = board(row, col − 1);
      if (v ≥ 0) oo, trouble += placed[abs(val − v)], placed[abs(val − v)] = 1;
    }
    if (col < 2) {
      o, v = board(row, col + 1);
      if (v ≥ 0) oo, trouble += placed[abs(val − v)], placed[abs(val − v)] = 1;
    }
    for (i = 0; i < rank; i++)
      if (i ≠ row) {
        o, v = board(i, col);
        if (v ≥ 0) oo, trouble += placed[abs(val − v)], placed[abs(val − v)] = 1;
      }
    if (row ≡ rank) rank++;
  }
```

This code is used in sections 9 and 22.

17.

```
⟨ Unmake mv 17 ⟩ ≡
  if (mv ≥ #10000) mv = (mv ≫ 16) + ((mv & #ffff) ≪ 16);      /* undo in opposite order */
  for ( ; mv; mv >>= 16) {
    val = mv & #ff, row = (mv ≫ 8) & #f, col = (mv ≫ 12) & #f;
    if (row ≡ rank − 1 ∧ (o, board(row, (col + 1) mod 3) < 0) ∧ (o, board(row, (col + 2) mod 3) < 0))
      rank = row;
    o, labeled[val] = −1;
    o, board(row, col) = −1;
    oo, colknown[col]−−;
    if (col > 0) {
      o, v = board(row, col − 1);
      if (v ≥ 0) o, placed[abs(val − v)] = 0;
    }
    if (col < 2) {
      o, v = board(row, col + 1);
      if (v ≥ 0) o, placed[abs(val − v)] = 0;
    }
    for (i = 0; i < rank; i++)
      if (i ≠ row) {
        o, v = board(i, col);
        if (v ≥ 0) o, placed[abs(val − v)] = 0;
      }
  }
```

This code is used in sections 9 and 22.

18. The nitty gritty. OK, I've put all the infrastructure into place. It remains to figure out all legal ways to place a new edge whose endpoints are labeled a and b. (This is where the graph $K_m \,\square\, P_3$ is really "hardwired.")

I do this by brute force, while trying to be careful. Sometimes I just barely avoided a bug, but I hope that I've exterminated them all.

```
⟨ Record all possible (a, b) moves in the array move[l] 18 ⟩ ≡
  {
    oo, aa = labeled[a], bb = labeled[b];
    if (aa ≥ 0) {
      if (bb ≥ 0) continue;      /* a and b are already on the board */
      row = aa & #f, col = aa ≫ 4;
      ⟨ Record all legal placements of b adjacent to a 19 ⟩;
    } else if (bb ≥ 0) {
      row = bb & #f, col = bb ≫ 4;
      ⟨ Record all legal placements of a adjacent to b 21 ⟩;
    }
    else ⟨ Record all adjacent placements of a and b 22 ⟩;
  }
```

This code is used in section 9.

19.

```
⟨ Record all legal placements of b adjacent to a 19 ⟩ ≡
  switch (col) {
  case 0: if ((o, board(row, 1) < 0) ∧ legal_in_col(b, 1) ∧ ((o, board(row, 2) < 0) ∨ (o,
          ¬placed[abs(b − board(row, 2))]))) o, move[l][t++] = pack(row, 1, b);
    break;
  case 1: if ((o, board(row, 0) < 0) ∧ legal_in_col(b, 0)) o, move[l][t++] = pack(row, 0, b);
    if ((o, leftknown) ∧ (o, board(row, 2) < 0) ∧ legal_in_col(b, 2)) o, move[l][t++] = pack(row, 2, b);
    break;
  case 2: if ((o, board(row, 1) < 0) ∧ legal_in_col(b, 1) ∧ ((o, board(row, 0) < 0) ∨ (o,
          ¬placed[abs(b − board(row, 0))]))) o, move[l][t++] = pack(row, 1, b);
    break;
  }
  if (legal_in_col(b, col)) {
    for (i = 0; i < rank; i++)
      if (o, board(i, col) < 0) {
        if (col > 0 ∧ (o, board(i, col − 1) ≥ 0) ∧ (o, placed[abs(b − board(i, col − 1))])) continue;
        if (col < 2 ∧ (o, board(i, col + 1) ≥ 0) ∧ (o, placed[abs(b − board(i, col + 1))])) continue;
        o, move[l][t++] = pack(i, col, b);
      }
    if (rank < m) o, move[l][t++] = pack(rank, col, b);
  }
```

This code is used in section 18.

20. ⟨ Subroutines 3 ⟩ +≡

```
int legal_in_col(val, col)
{
  register int i, v;
  if (o, colknown[col] ≡ m) return 0;
  for (i = 0; i < rank; i++) {
    o, v = board(i, col);
    if (v ≥ 0 ∧ (o, placed[abs(v − val)])) return 0;
  }
  return 1;
}
```

21. ⟨ Record all legal placements of a adjacent to b 21 ⟩ ≡

```
switch (col) {
case 0: if ((o, board(row, 1) < 0) ∧ legal_in_col(a, 1) ∧ ((o, board(row, 2) < 0) ∨ (o,
        ¬placed[abs(a − board(row, 2))]))) o, move[l][t++] = pack(row, 1, a);
  break;
case 1: if ((o, board(row, 0) < 0) ∧ legal_in_col(a, 0)) o, move[l][t++] = pack(row, 0, a);
  if ((o, leftknown) ∧ (o, board(row, 2) < 0) ∧ legal_in_col(a, 2)) o, move[l][t++] = pack(row, 2, a);
  break;
case 2: if ((o, board(row, 1) < 0) ∧ legal_in_col(a, 1) ∧ ((o, board(row, 0) < 0) ∨ (o,
        ¬placed[abs(a − board(row, 0))]))) o, move[l][t++] = pack(row, 1, a);
  break;
}
if (legal_in_col(a, col)) {
  for (i = 0; i < rank; i++)
    if (o, board(i, col) < 0) {
      if (col > 0 ∧ (o, board(i, col − 1) ≥ 0) ∧ (o, placed[abs(a − board(i, col − 1))])) continue;
      if (col < 2 ∧ (o, board(i, col + 1) ≥ 0) ∧ (o, placed[abs(a − board(i, col + 1))])) continue;
      o, move[l][t++] = pack(i, col, a);
    }
  if (rank < m) o, move[l][t++] = pack(rank, col, a);
}
```

This code is used in section 18.

22. Finally, the hard case is when a double move is needed. First I tentatively try all placements of a, actually changing the board. Then I record the double moves for b adjacent to every such placement. Of course the board has to be restored again.

```
⟨ Record all adjacent placements of a and b 22 ⟩ ≡
  for (o, ccol = (leftknown ? 2 : 1); ccol ≥ 0; ccol−−)
    if (legal_in_col(a, ccol)) {
      for (ii = 0; ii < rank; ii++)
        if (o, board(ii, ccol) < 0) {
          if (ccol > 0 ∧ (o, board(ii, ccol − 1) ≥ 0) ∧ (o, placed[abs(a − board(ii, ccol − 1))])) continue;
          if (ccol < 2 ∧ (o, board(ii, ccol + 1) ≥ 0) ∧ (o, placed[abs(a − board(ii, ccol + 1))])) continue;
          aa = mv = pack(ii, ccol, a); ⟨ Make mv 16 ⟩; mv = aa;
          if (¬trouble) ⟨ Record all double placements of b adjacent to a 23 ⟩;
          ⟨ Unmake mv 17 ⟩;
        }
      if (rank < m) {
        aa = mv = pack(rank, ccol, a); ⟨ Make mv 16 ⟩; mv = aa;
        if (¬trouble) ⟨ Record all double placements of b adjacent to a 23 ⟩;
        ⟨ Unmake mv 17 ⟩;
      }
    }
```

This code is used in section 18.

23.

```
⟨ Record all double placements of b adjacent to a 23 ⟩ ≡
  {
    switch (col) {
    case 0: if ((o, board(row, 1) < 0) ∧ legal_in_col(b, 1) ∧ ((o, board(row, 2) < 0) ∨ (o,
              ¬placed[abs(b − board(row, 2))]))) o, move[l][t++] = (pack(row, 1, b) ≪ 16) + mv;
      break;
    case 1: if ((o, board(row, 0) < 0) ∧ legal_in_col(b, 0)) o, move[l][t++] = (pack(row, 0, b) ≪ 16) + mv;
      if ((o, leftknown) ∧ (o, board(row, 2) < 0) ∧ legal_in_col(b, 2))
        o, move[l][t++] = (pack(row, 2, b) ≪ 16) + mv;
      break;
    case 2: if ((o, board(row, 1) < 0) ∧ legal_in_col(b, 1) ∧ ((o, board(row, 0) < 0) ∨ (o,
              ¬placed[abs(b − board(row, 0))]))) o, move[l][t++] = (pack(row, 1, b) ≪ 16) + mv;
      break;
    }
    if (legal_in_col(b, col)) {
      for (i = 0; i < rank; i++)
        if (o, board(i, col) < 0) {
          if (col > 0 ∧ (o, board(i, col − 1) ≥ 0) ∧ (o, placed[abs(b − board(i, col − 1))])) continue;
          if (col < 2 ∧ (o, board(i, col + 1) ≥ 0) ∧ (o, placed[abs(b − board(i, col + 1))])) continue;
          o, move[l][t++] = (pack(i, col, b) ≪ 16) + mv;
        }
      if (rank < m) o, move[l][t++] = (pack(rank, col, b) ≪ 16) + mv;
    }
  }
```

This code is used in section 22.

24. Index.

BACK-GRACEFUL-KMP3