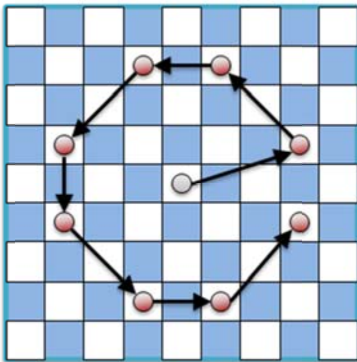
ALG-koně - komentář a řešení

Pro každého ALG-koně zřídíme jednu vrstvu ALG-šachovnice. Do každého políčka ve vrstvě zapíšeme minimální počet skoků, které tento ALG-kůň potřebuje na dosažení tohoto políčka. Políčko, na kterém ALG-kůň na začátku stojí, označíme 0. Všechna políčka, na která může skočit z políčka 0, označíme 1. Všechna dosud neoznačená políčka, na která může skočit z některého políčka označeného 1, označíme 2. Obecně, všechna dosud neoznačená políčka, na která může skočit z některého políčka označeného *k*, označíme *k*+1. Evidentně se jedná o princip procházení do šířky (BFS) a tak jej také s pomocí obyčejné fronty implementujeme.

Z dané vrstvy ALG-šachovnice se tak stane neorientovaný graf, jehož uzly jsou políčka šachovnice a jsou spojeny hranou právě tehdy, když daný ALG-kůň může jedním skokem přejít z políčka odpovídajícímu jednomu uzlu na políčko odpovídající druhému uzlu. V každém uzlu bude zaznamenána délka nejkratší cesty do počátečního uzlu, z něhož se začal budovat BFS strom

Až budeme mít vyplněny všechny vrstvy, projdeme naposled všechna políčka ALG-šachovnice a na každém políčku projdeme všechny vrstvy a tak pro dané políčko určíme, jaký je minimální čas, za který se všichni ALG-koně mohou na toto políčko dostat. Z takto vypočtených hodnot stačí vybrat minimální a určit, kolikrát se v ALG-šachovnici vyskytuje. To lze ovšem provádět zároveň s posledním procházením ALG-šachovnice.

V implementaci se obejdeme s obyčejným 3D polem, kde první rozměr představuje vrstvy a zbylé dva rozměry splývají s rozměry ALG-šachovnice.



Při průchodu do šířky, když ALG-kůň stojí na políčku p, musíme dokázat určit, do kterých z maximálně možných 8 směrů může skočit dalším skokem. V řeči grafů, musíme znát všechny hrany, které vedou z uzlu odpovídajícího aktuálnímu políčku do dalších nejvýše 8 uzlů. Tuto informaci můžeme před začátkem prohledávání předpočítat a vytvořit tak vlastně kompletní reprezentaci grafu (někdy se to hodí). Během BFS ale každou hranu grafu použijeme maximálně dvakrát, a tak se zdá, že se předpočítávání v této úloze nevyplatí, protože hrany již dále potřebovat nijak nebudeme.V předložené implementaci je předpočítán pouze relativní posun ALG-koně při prozkoumávání svého okolí. Nepamatujeme si absolutní souřadnice políček, jež může ALG-kůň příštím skokem navštívit, protože ta jsou pro každé políčko jiná. Stačí si pamatovat vektor přesunu na další možné cílové políčko skoku z jednoho uzlu, jak je to naznačeno na obrázku pro ALG-koně s parametry K(1) = 3 a K(2) = 1. Vektory budou v tomto případě postupně (3,1), (-2, 2), (-2, 0), ..., (2, 2) a použijeme je identicky při návštěvě každého políčka. Otázku, zda skok vede mimo šachovnici, vyřešíme na místě v kódu. Jednim z běžných postupů je, že se kolem dokola šachovnice (nejen ALG-šachovnice) vytvoří dostatečný počet dalších obalových řad a soupců políček, která se však označí jako nedosažitelná, takže není třeba zvlášť kontrolovat, zda skok vede mimo předepsaný prostor, korektnost skoku je vždy zapsána přímo políčku. V našem případě ale toto řešení nepoužijeme, protože při možné maximální délce skoku ALG-koně rovné délce strany ALG-šachovnice, bychom museli plochu ALG-šachovnice prakticky zdevateronásobit, což patrně není efektivní.

```
public static void main(String[] args) throws IOException {

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
  StringTokenizer st = new StringTokenizer(br.readLine());
  int N = Integer.valueOf(st.nextToken());
  int k11 = Integer.valueOf(st.nextToken());
  int k12 = Integer.valueOf(st.nextToken());
  int k21 = Integer.valueOf(st.nextToken());
  int k22 = Integer.valueOf(st.nextToken());
  int k31 = Integer.valueOf(st.nextToken());
  int k32 = Integer.valueOf(st.nextToken());
  int k41 = Integer.valueOf(st.nextToken());
  int k42 = Integer.valueOf(st.nextToken());
```

```
  int [][][] board = new int [4][N][N];          // 4 ALG-knights
  BFSfill(0,    0,  k11, k12, board[0]);         // fill board layers one by one
  BFSfill(N-1,  0,  k21, k22, board[1]);
  BFSfill(N-1, N-1, k31, k32, board[2]);
  BFSfill(0   , N-1, k41, k42, board[3]);
  evaluate (board);                              // collect the results
}

static int [][] neighVectors = new int [8][2];

static void makeNeighVectors(int k1, int k2) {
  if (k1 > k2)
     { int tmp = k1; k1 = k2; k2 = tmp; }       // swap to have k1 <= k2
  int d = k2-k1;
  neighVectors[0][0] =  k2;   neighVectors[0][1] = k1;
  neighVectors[1][0] =  -d;   neighVectors[1][1] = d;
  neighVectors[2][0] = -2*k1; neighVectors[2][1] = 0;
  neighVectors[3][0] = -d;    neighVectors[3][1] = -d;
  neighVectors[4][0] =  0;    neighVectors[4][1] = -2*k1;
  neighVectors[5][0] =  d;    neighVectors[5][1] = -d;
  neighVectors[6][0] =  2*k1; neighVectors[6][1] = 0;
  neighVectors[7][0] =  d;    neighVectors[7][1] =  d;
}

static int UNREACHED = -1;

static void BFSfill ( int X, int Y, int k1, int k2, int [][] board) {
  makeNeighVectors(k1, k2) ;
  for(int i = 0; i < board.length; i++)
    for(int j = 0; j < board[i].length;j++)
      board[i][j] = UNREACHED;        // init board
  int N = board.length;
  int [][] q = new int [N*N][3];      // simple queue
  int front = 0;
  int tail = 0;
  int distance = 0;

  board[Y][X] = distance;             // register zero distance form the start
  q[front][0] = distance;             // init the queue
  q[front][1] = X;                    // [X, Y] is also a start field
  q[front][2] = Y;                    // enqueue it
  while(front <= tail) {
    distance = q[front][0];           // pop the queue
    X = q[front][1];
    Y = q[front++][2];
    for(int i = 0; i < 8; i++) {
      X += neighVectors[i][0];        // explore all 8 neighbouring cells
      Y += neighVectors[i][1];
      if ((X < N) && (X >= 0) && (Y < N) && (Y >= 0)
              && (board[Y][X] == UNREACHED) ) {
        board[Y][X] = distance+1;
        q[++tail][0] = distance+1;    // enqueue neighbouring field
        q[tail][1] = X;
        q[tail][2] = Y;
      }
    } // for
 } // while
} // BFS
```

```
static void evaluate (int [][][] allBoards) {
  int n = allBoards[0].length;                        // 4 in our problem
  int min = Integer.MAX_VALUE;
  int maxDistOnOneField;
  boolean allCanReachTheField;

  int totalBestDist = Integer.MAX_VALUE;              // infinity first
  int totalBestDistCounter = 0;

  for(int i = 0; i < n; i++)
    for(int j = 0; j < n; j++) {
      // analyze the particular field
      maxDistOnOneField = Integer.MIN_VALUE;          // minus infinity first
      allCanReachTheField = true;
      for(int k = 0; k < allBoards.length; k++) {
        if (allBoards[k][i][j] == UNREACHED)
          { allCanReachTheField = false; break; }
        if (allBoards[k][i][j] > maxDistOnOneField)
          maxDistOnOneField = allBoards[k][i][j];
      }
      if (!allCanReachTheField) continue;

      // check if this field is the best so far
      if (maxDistOnOneField > 0) {
        if (totalBestDist > maxDistOnOneField ) { // yes, it is
          totalBestDist = maxDistOnOneField;
          totalBestDistCounter = 1;
        }
        else
          if (totalBestDist == maxDistOnOneField )
            totalBestDistCounter++;                    // another field equally good
      }
    } // both for loops

  // print results
   if (totalBestDist == Integer.MAX_VALUE)
     System.out.printf("-1\n");
   else
     System.out.printf("%d %d\n",totalBestDist,totalBestDistCounter);
 }
```