```
/*
Dieses Programm ist Teil des Jugend-forscht Projekts "Lösung des n-
Damenproblems auf einem adiabatischen Quantencomputer"
 Programmiersprache: Processing (Java-abwandlung)
 IDE unter https://processing.org/download/
 FILE 1
 */

import java.util.*;

//Schachfeldgröße; frei Veränderbar
int n = 8;
boolean[][] schachfeld = new boolean[n][n];
//erste, zufällige Verteilung der Damen auf dem Schachfeld
boolean[][] ausgangsposition = new boolean[n][n];
//Matrix
int[][] hamiltonianMatrix = new int[n*n][n*n];
//Strafterm
ArrayList <Summand> hamiltonianTerm = new ArrayList<Summand>();
//Exportierte Daten des Energie-Durchlauf Verhältnis für jeden Optimierungsalgorithmus
ArrayList<String> greedyGraph = new ArrayList<String>();
ArrayList<String> simAnnGraph = new ArrayList<String>();
ArrayList<String> thresholdGraph = new ArrayList<String>();
ArrayList<String> greatDelugeGraph = new ArrayList<String>();


void setup() {
  //erste zufällige Verteilung der Damen wird ausgewürfelt
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      ausgangsposition[i][j] = int(random(0, 2)) == 1;
    }
  }
  noLoop();
}



void draw() {
  hamiltonianTermAufstellen();

  //Hamiltonianformel ausgeben in der Form: Index * Koordinate des einen Feldes *
Koordinate des anderen Feldes
  for (int i=0; i<hamiltonianTerm.size(); i++) {
    Summand s=hamiltonianTerm.get(i);
    print("+"+s.multiplikator + "*S(" +int(s.feld1.x)+"|"+int(s.feld1.y)+")"+"*S("
+int(s.feld2.x)+"|"+int(s.feld2.y)+")"+" ");
  }
  println("\n");

  //Hamiltonian-Matrix ausgeben
  erstelleHamiltonianMatrix(hamiltonianTerm);
  for (int i=0; i<n*n; i++) {
    for (int j=0; j<n*n; j++) {
      if (hamiltonianMatrix[i][j] >= 0)
        print(" ");
      print(hamiltonianMatrix[i][j]);
    }
    println();
  }
  exportiereHamiltonianMatrix();
  println("Hamiltonian-Matrix exportiert!\n\n");

  //ausgeben der Ausgangsbesetzung
  println("Ausgangslösung:");
  for (int i=0; i<n; i++) {
```

```
    for (int j=0; j<n; j++) {
      print((ausgangsposition[i][j]?1:0));
      print(" ");
    }
    println();
  }
  println("\n");

  // Ausführen der Optimierungsalgorithmen; vor jedem wird das Schachfeld auf die
Ausgangsposition zurückgesetzt
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeld[i][j]=ausgangsposition[i][j];
    }
  }
  greedy();
  exportiereGraph(greedyGraph, "greedy");
  println("\n");

  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeld[i][j]=ausgangsposition[i][j];
    }
  }
  simulatedAnnealing();
  exportiereGraph(simAnnGraph, "simAnn");
  println("\n");

  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeld[i][j]=ausgangsposition[i][j];
    }
  }
  greatDeluge();
  exportiereGraph(greatDelugeGraph, "greatDeluge");
  println("\n");

  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeld[i][j]=ausgangsposition[i][j];
    }
  }
  thresholdAccepting();
  exportiereGraph(thresholdGraph, "threshold");
}


//Strafterm aufstellen; siehe Doku S.5
void hamiltonianTermAufstellen() {

  //Diagonalen untere Hälfte (Ecke unten rechts; von rechts oben nach links unten) ohne
Gesamtdiagonale
  for (int i = 1; i <= n-1; i++) {
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int x = i, y = n, j = 1; (x >= 1 || y >= 1) && j <= i; x--, y--, j++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand>
hamiltonianTermInternAusmult=loeseKlammernAufDiagonal(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }


  //Diagonalen obere Hälfte (Ecke oben links; von rechts oben nach links unten)
  for (int i=n; i>=1; i--) {
```

```
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int x=n, y=i, j=1; (x>=1||y>=1)&&j<=i; x--, y--, j++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand>
hamiltonianTermInternAusmult=loeseKlammernAufDiagonal(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }

  //diagonal obere hälfte links oben nach rechts unten ohne diagonale
  for (int i=1; i<=n-1; i++) {
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int x=1, y=i, j=1; (x<=n||y>=1)&&j<=i; x++, y--, j++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand> hamiltonianTermInternAusmult =
loeseKlammernAufDiagonal(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }


  //diagonal untere hälfte links oben nach rechts unten
  for (int abbruch=n, i=1; abbruch>=1||i<=n; abbruch--, i++) {
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int x=i, y=n, j=1; (x<=n||y>=1)&&j<=abbruch; x++, y--, j++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand>
hamiltonianTermInternAusmult=loeseKlammernAufDiagonal(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }

  //oben nach unten (y)
  for (int x=1; x<=n; x++) {
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int y=1; y<=n; y++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand>
hamiltonianTermInternAusmult=loeseKlammernAuf(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }

  //links nach rechts (x)
  for (int y=1; y<=n; y++) {
    ArrayList <PVector> hamiltonianTermIntern = new ArrayList<PVector>();
    for (int x=1; x<=n; x++) {
      hamiltonianTermIntern.add(new PVector(x, y));
    }
    ArrayList <Summand>
hamiltonianTermInternAusmult=loeseKlammernAuf(hamiltonianTermIntern);
    for (Summand s : hamiltonianTermInternAusmult) {
      hamiltonianTerm.add(s);
    }
  }
}


// "Umformen" des ursprünglichen Terms; siehe Doku S.6
```

```
ArrayList <Summand> loeseKlammernAuf(ArrayList <PVector> hamiltonianTerm) {
  ArrayList <Summand> result = new ArrayList<Summand>();
  for (int i=0; i<hamiltonianTerm.size(); i++) {
    for (int j=i+1; j<hamiltonianTerm.size(); j++) {
      result.add(new Summand(+2, hamiltonianTerm.get(i), hamiltonianTerm.get(j)));
    }
  }
  for (PVector p : hamiltonianTerm) {
    result.add(new Summand(-1, p, p));
  }
  return result;
}


// "Umformen" des ursprünglichen Terms; siehe Doku S.6
ArrayList <Summand> loeseKlammernAufDiagonal(ArrayList <PVector> hamiltonianTerm) {
  ArrayList <Summand> result = new ArrayList<Summand>();
  for (int i=0; i<hamiltonianTerm.size(); i++) {
    for (int j=i+1; j<hamiltonianTerm.size(); j++) {
      result.add(new Summand(+2, hamiltonianTerm.get(i), hamiltonianTerm.get(j)));
    }
  }

  return result;
}


//Die einzelnen Term-
bestandteile werden auf die Matrix übertragen; siehe Doku: letzter Absatz von "Hamilton
ian" S.7
void erstelleHamiltonianMatrix(ArrayList <Summand> hamiltonianTerm) {
  for (int i=0; i<hamiltonianTerm.size(); i++) {
    if  ( ((int(hamiltonianTerm.get(i).feld1.x)-
1)+(int(hamiltonianTerm.get(i).feld1.y)-1)*n) < ((int(hamiltonianTerm.get(i).feld2.x)-
1)+(int(hamiltonianTerm.get(i).feld2.y)-1)*n)) {
      hamiltonianMatrix[(int(hamiltonianTerm.get(i).feld1.x)-
1)+(int(hamiltonianTerm.get(i).feld1.y)-1)*n][(int(hamiltonianTerm.get(i).feld2.x)-
1)+(int(hamiltonianTerm.get(i).feld2.y)-1)*n]+=hamiltonianTerm.get(i).multiplikator;
    } else {
      hamiltonianMatrix[(int(hamiltonianTerm.get(i).feld2.x)-
1)+(int(hamiltonianTerm.get(i).feld2.y)-1)*n][(int(hamiltonianTerm.get(i).feld1.x)-
1)+(int(hamiltonianTerm.get(i).feld1.y)-1)*n] +=hamiltonianTerm.get(i).multiplikator;
    }
  }
}


//Berechnung der Energie auf Basis der aktuellen Verteilung der Damen; siehe Doku S.3 u
nten
int kostenfunktion(boolean[][]schachfeldLocal) {
  int ergebnis=0;
  int[] vektor=new int[n*n];
  int[] vektorerg=new int[n*n];

  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      vektor[i*n+j]=schachfeldLocal[i][j]==true? (1):(0);
    }
  }

  for (int i=0; i<n*n; i++) {
    for (int j=0; j<n*n; j++) {
      vektorerg[i]+=hamiltonianMatrix[i][j]*vektor[j];
    }
  }

  for (int i=0; i<n*n; i++) {
```

```
    ergebnis+=vektor[i]*vektorerg[i];
  }

  return ergebnis;
}


//greedy-Algorithmus; siehe Doku S.7
void greedy() {
  println("greedy:");

  for (int durchlauf=0; durchlauf<10000; durchlauf++) {
    int alteKosten = kostenfunktion(schachfeld);
    int x = int(random(n));
    int y = int(random(n));
    schachfeld[x][y]=!schachfeld[x][y]; //Änderung des Schachfelds
    int kosten=kostenfunktion(schachfeld);
    //Überprüfen: ist die Lösung schlechter? -> Änderung Rückgängig machen
    if (kosten>alteKosten) {
      schachfeld[x][y]=!schachfeld[x][y];
    }
    greedyGraph.add(durchlauf+" "+kostenfunktion(schachfeld));
    if (kosten==n*(-2) ) {
      println(durchlauf+" Durchläufe");
      break;
    }
  }

  maleSchachfeld(schachfeld);
  println("Kosten: "+kostenfunktion(schachfeld));
}


//threshold-Accepting; siehe Doku S.8
void thresholdAccepting() {
  println("threshold:");
  float threshold=randomWalkTreshold(); //Schwelle

  for (int durchlauf=0; durchlauf<10000; durchlauf++) {
    int alteKosten = kostenfunktion(schachfeld);
    int x = int(random(n));
    int y = int(random(n));
    schachfeld[x][y]=!schachfeld[x][y]; //Änderung des Schachfelds
    int kosten=kostenfunktion(schachfeld);
    //Überprüfen: ist der Unterschied zw. neuer und alter Lösung größer als die
Schwelle? -> Änderung Rückgängig machen
    if ((kosten-alteKosten) >= threshold) {
      schachfeld[x][y]=!schachfeld[x][y];
    }
    thresholdGraph.add(durchlauf+" "+kostenfunktion(schachfeld));
    if (kosten==n*(-2) ) {
      println(durchlauf+" Durchläufe");
      println("threshold Schwelle: "+threshold);
      break;
    }
    threshold*=0.99;
  }
  maleSchachfeld(schachfeld);
  println("Kosten: "+kostenfunktion(schachfeld));
}


//simulated-Annealing; siehe Doku S.9
void simulatedAnnealing() {
  println("simAnn:");
  float simAnn=randomWalkTreshold(); //Schwelle
```

```
  for (int durchlauf=0; durchlauf<10000; durchlauf++) {
    int alteKosten = kostenfunktion(schachfeld);
    int x = int(random(n));
    int y = int(random(n));
    schachfeld[x][y]=!schachfeld[x][y];  //Änderung des Schachfelds
    int kosten = kostenfunktion(schachfeld);
    int kostenUnterschied=kosten-alteKosten;
    //Überprüfen: ist die Lösung schlechter geworden und ist eine zufällige Zahl größer
als e^(-Unterschied/Schwelle)? -> Änderung Rückgängig machen
    if (kostenUnterschied>0 && (random(1)>=exp(-kostenUnterschied/simAnn))) {
      schachfeld[x][y]=!schachfeld[x][y];
    }
    simAnnGraph.add(durchlauf+" "+kostenfunktion(schachfeld));
    if (kosten==n*(-2) ) {
      println(durchlauf+" Durchläufe");
      println("simAnn Schwelle: "+simAnn);
      break;
    }
    simAnn*=0.85;
  }
  maleSchachfeld(schachfeld);
  println("Kosten: "+kostenfunktion(schachfeld));
}


//great Deluge Algorithmus; siehe Doku S.9 unten
void greatDeluge() {
  println("greatDeluge");
  boolean[][] schachfeldWorstCase = new boolean[n][n];
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeldWorstCase[i][j]=true;
    }
  }
  float greatDeluge=kostenfunktion(schachfeldWorstCase); //Schwelle

  for (int durchlauf=0; durchlauf<1000000; durchlauf++) {
    int x = int(random(n));
    int y = int(random(n));
    schachfeld[x][y]=!schachfeld[x][y]; //Änderung des Schachfelds
    int kosten=kostenfunktion(schachfeld);
    //Überprüfen: Ist die Energie über der Schwelle? -> Änderung Rückgängig machen
    if (kosten>greatDeluge+(n*-2)) {
      schachfeld[x][y]=!schachfeld[x][y];
    }
    greatDelugeGraph.add(durchlauf+" "+kostenfunktion(schachfeld));
    if (kosten==n*(-2) ) {
      println(durchlauf+" Durchläufe");
      println("greatDeluge Schwelle: "+greatDeluge);
      break;
    }
    greatDeluge*=0.9999;
  }
  maleSchachfeld(schachfeld);
  println("Kosten: "+kostenfunktion(schachfeld));
}


//ausgeben des Schachfelds in der Konsolenleiste
void maleSchachfeld(boolean[][] schachfeldLocal) {
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      print(" ");
      print(schachfeldLocal[i][j]?(1):(0));
    }
    println();
  }
```

```
}



int randomWalkTreshold() {
  ArrayList<Integer> kostenDifferenzen=new ArrayList<Integer>();
  boolean[][] schachfeldRandomWalk = new boolean[n][n];
  for (int i=0; i<n; i++) {
    for (int j=0; j<n; j++) {
      schachfeldRandomWalk[i][j]=true;
    }
  }
  for (int i=0; i<10000; i++) {
    int kostenOld=kostenfunktion(schachfeldRandomWalk);
    int x = int(random(n));
    int y = int(random(n));
    schachfeldRandomWalk[x][y]=!schachfeldRandomWalk[x][y];
    kostenDifferenzen.add(kostenOld-kostenfunktion(schachfeldRandomWalk));
  }
  Collections.sort(kostenDifferenzen);
  return kostenDifferenzen.get(9900);
}


//exportieren der Matrix
void exportiereHamiltonianMatrix() {
  String [] export=new String[n*n];
  for (int x=0; x<n*n; x++) {
    export[x]="";
    for (int y=0; y<n*n; y++) {
      export[x]+=hamiltonianMatrix[x][y]+" ";
    }
  }
  saveStrings("qubomatrix_" + n + ".txt", export);
}


//exportieren der Graphen (Energie-
Durchlauf Diagramm der unterschiedlichen Optimierungsverfahren)
void exportiereGraph(ArrayList <String> listToWrite, String algorithmus) {
  String[]graphWrite= new String[listToWrite.size()];
  for (int i=0; i<listToWrite.size(); i++) {
    graphWrite[i]=listToWrite.get(i);
  }
  saveStrings("Graph_"+n+"_"+algorithmus+"ausgangs.txt", graphWrite);
}




//Hilfsklasse zum Speichern des Terms; siehe Doku S.3 / S.6
class Summand {
  int multiplikator;
  PVector feld1;
  PVector feld2;

  Summand(int multiplikator, PVector feld1, PVector feld2) {
    this.multiplikator = multiplikator;
    this.feld2 = feld2;
    this.feld1 = feld1;
  }

  Summand(int multiplikator, PVector feld1) {
    this.multiplikator = multiplikator;
    this.feld1 = feld1;
  }
}
```