Online-Appendix

A. Run time effects

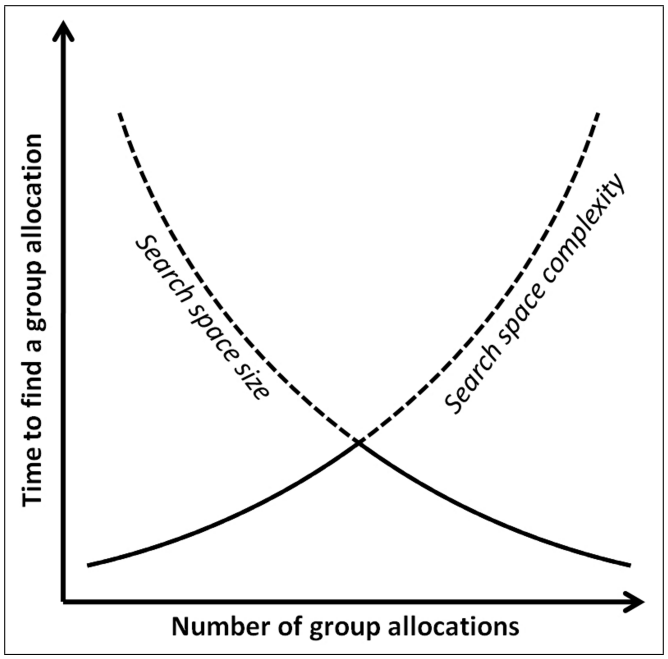

Figure A.1: The graph schematically illustrates two contrary effects which influence the run time for finding subsequent group allocations. Initially, a low run time is achieved when searching for a feasible group allocation since the search space is not constrained through matching histories. The longer the sequence of group allocations under the perfect stranger criterion, the more constraints have to be satisfied for further group allocations. While this increases the complexity of the search space, it simultaneously shrinks the size of the search space since many solutions can be excluded a-priori. Overall, this leads to an increase in run time for subsequent group allocations up to a certain point after which the run time decreases again.

B. Run time table

	Proposed Algorithm					**Brute-Force**				
P	G=2	G=3	G=4	G=5	G=6	G=2	G=3	G=4	G=5	G=6
4	~0ms	-	-	-	-	~0ms	-	-	-	-
5	-	-	-	-	-	-	-	-	-	-
6	~0ms	-	-	-	-	~0ms	-	-	-	-
7	-	-	-	-	-	-	-	-	-	-
8	~0ms	-	-	-	-	~0ms	-	-	-	-
9	-	~0ms	-	-	-	-	0.3ms	-	-	-
10	~0ms	-	-	-	-	3.1ms	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-
12	0.1ms	~0ms	-	-	-	23ms	4.3ms	-	-	-
13	-	-	-	-	-	-	-	-	-	-
14	0.1ms	-	-	-	-	30ms	-	-	-	-
15	-	0.1ms	-	-	-	-	70ms	-	-	-
16	0.2ms	-	~0ms	-	-	61ms	-	~0ms	-	-
17	-	-	-	-	-	-	-	-	-	-
18	0.2ms	0.1ms	-	-	-	6.4s	1.2s	-	-	-
19	-	-	-	-	-	-	-	-	-	-
20	0.3ms	-	0.4ms	-	-	88s	-	0.7s	-	-
21	-	0.2ms	-	-	-	-	0.6s	-	-	-
22	0.4ms	-	-	-	-	1096s	-	-	-	-
23	-	-	-	-	-	-	-	-	-	-
24	0.5ms	0.5ms	0.9ms	-	-	x	5.1s	3.8s	-	-
25	-	-	-	0.1ms	-	-	-	-	0.3s	-
26	0.7ms	-	-	-	-	x	-	-	-	-
27	-	1ms	-	-	-	-	54s	-	-	-
28	0.9ms	-	2.6ms	-	-	x	-	3.7s	-	-
29	-	-	-	-	-	-	-	-	-	-
30	1.2ms	2ms	-	1.1s	-	x	x	-	x	-
31	-	-	-	-	-	-	-	-	-	-
32	1.8ms	-	8.6ms	-	-	x	-	0.5s	-	-
33	-	5.5ms	-	-	-	-	x	-	-	-
34	2.3ms	-	-	-	-	x	-	-	-	-
35	-	-	-	1.2s	-	-	-	-	x	-
36	3ms	8.8ms	25ms	-	0.2ms	x	x	102s	-	215s
37	-	-	-	-	-	-	-	-	-	-
38	4.3ms	-	-	-	-	x	-	-	-	-
39	-	19ms	-	-	-	-	x	-	-	-
40	7.2ms	-	72ms	1.7s	-	x	-	x	x	-

Table B.1: The Table shows the average computation time needed to generate a single complete sequence for specific configurations (p,g). Run time tests were conducted on an i5 2500k without parallel computing (only a single core was used). Computation time per configuration was measured over two hours for each configuration, resulting in at least 4000 computations per configuration for our algorithm. For large problem instances, it was not possible to measure the brute-force run time due to its substantial increase in computation time. Computations were stopped after five hours. Configurations, were no complete sequence could be calculated within five hours are marked with an x.

C. Run time comparison

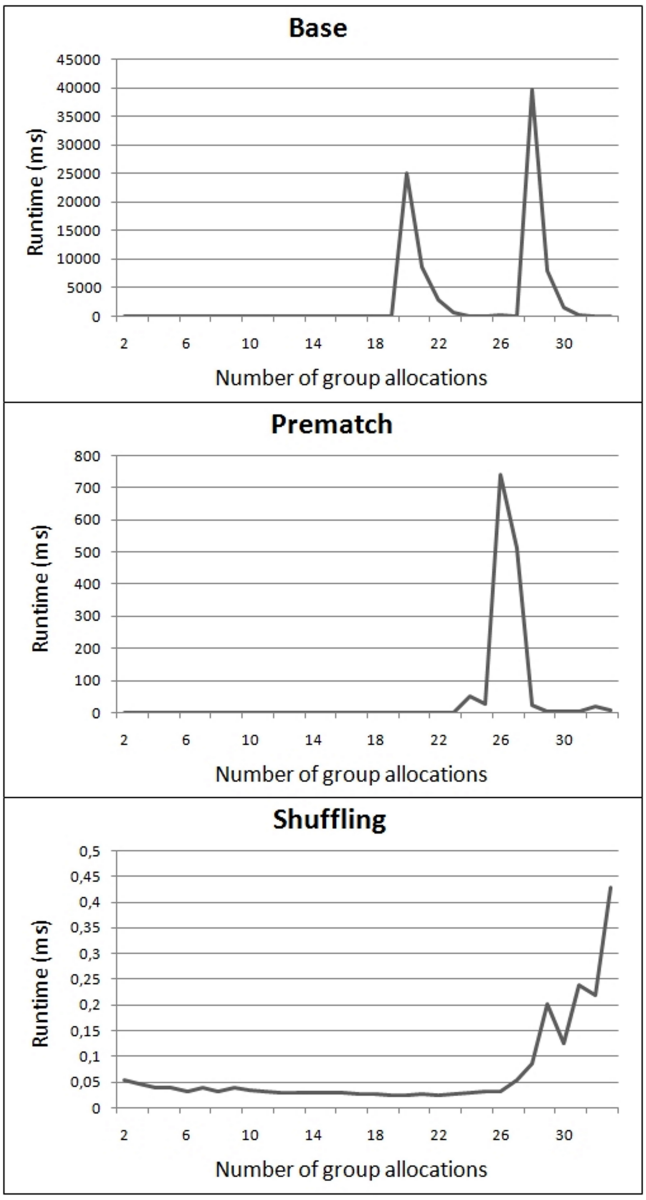

Figure C.2: For this figure, group allocations for the configuration (p,g) = (36,2) have been computed 1 million times. Thereby, the average time for finding the next group allocation within a sequence was measured. Run time measurements where conducted on an i5 2500k. The top graph displays the computation time for the proposed algorithm without random shuffling. For the graph in the middle, the first 17 group allocations have been generated with a systematic pattern before applying the proposed algorithm. The bottom graph displays average results of the proposed algorithm, combined with random shuffling.

D. Pseudocode

Algorithm 1 This algorithm generates a complete sequence under the perfect stranger criterion. Input parameters are the number of participants *p* and the group size g. Each of the *p* participants has a listing of participants, which they did not meet in previous group allocations, that is updated after each successful group allocation. Executing the function *AllocationSequence* returns a complete PSM sequence.

```
//possiblePartners[i,] symbolizes participants which are unknown to i
possiblePartners ← p × p matrix of ones with zeros on its diagonal

function ALLOCATIONSEQUENCE (p, g)
    sequence ← list of group allocations
    participantList ← list of p participants

    //Searching for a complete sequence
    while True do
        //Find new group allocation
        groupList ← empty list of groups
        groupList ← FINDALLOCATION (participantList, groupList,p, g)

        //allocate groups, if a match has been found
        if groupList != Null then
            add groupList to sequence
            ∀r, c ∈ [1, p]: possiblePartners[r, c] = 0 if c and r are grouped
            //optionally, shuffling of all lists can be inserted here
        else
            Break
        end if
    end while

    return sequence
end function
```

```
function FINDALLOCATION (unusedElem, groupList, p, g)
    if number of groups in groupList == (p/g) then
        return groupList
    end if
    memory ← empty group
    pivotElement ← first element of unusedElem
    posMatches ← unusedElem ∩ (participants at possiblePartners[pivotElement, ])
    if size of posMatches >= (g-1) then
        return Null
    end if

    //iterate over possible groups to add to groupList
    while True do
        //find group for given pivotElement
        curGroup ← new group with only pivotElement included
        curGroup ← FINDGROUP (posMatches, curGroup,g,memory)

        //stop if no group building was possible, else next recursion
        if curGroup == Null then
            return Null
        else
            add curGroup to groupList
            newUnusedElem ← unusedElem \ curGroup
            newGroupList ← FINDALLOCATION (newUnusedElem, groupList, p, g)
        end if

        //Store latest group in memory when group allocation failed
        if newGroupList == Null then
            memory ← curGroup
            remove curGroup from groupList
        else
            return newGroupList
        end if
    end while

    return Null
end function
```

```
function FINDGROUP (availElem, group, g, memory)
    size ← number of elements in group

    //end condition
    if memory is not empty then
        recursively build the group from memory
        clear memory and skip the rebuilt group in recursion
    end if

    if g == size then
        return group
    end if

    if number of elements in availElem < (g-size) then
        return Null
    end if

    //fetch new partner for group
    while number of elements in availElem > 0 do
        newPartner ← first element of availElem
        add newPartner to group
        posPartners ← participants at possiblePartners[newPartner, ]
        newAvailElem ← availElem ∩ posPartners
        newGroup ← FINDGROUP (newAvailElem,group,g,memory)

        if newGroup == Null then
            remove newPartner from availElem
            remove newPartner from group
        else
            return newGroup
        end if
    end while

    return Null
end function
```

E. Java source code including the classes "Element", "Group" and "MatcherMulti" with main method

```
package perfectStranger.multiCore;
import java.util.ArrayList;

//An Element with matching history (Participant during
    PSM)
public class Element {
        private ArrayList<Element> possiblePartner;
        private int id;

        public Element(int id){
                possiblePartner = new ArrayList<Element
                    >();
                this.id = id;
        }

        public void removeParnter(Element e){
                possiblePartner.remove(e);
        }

        public void setPartnerList(ArrayList<Element>
            possiblePartner){
                this.possiblePartner = possiblePartner;
        }

        public ArrayList<Element> getPartnerList(){
                return possiblePartner;
        }

        public String toString(){
                String result;
                //use uniform number format (for numbers
                    smaller than 100)
                if(id >= 10){
                        result = Integer.toString(id);
                }else{
                        result = " " + Integer.toString(
                            id);
                }


                return result;
        }
}
```

```
package perfectStranger.multiCore;
import java.util.ArrayList;

//A Group of Elements (Participants during PSM)
public class Group {

        private ArrayList<Element> elementList;

        public Group(){
                elementList = new ArrayList<Element>();
        }

        public ArrayList<Element> getElements(){
                return(elementList);
        }

        public void removeElement(Element e){
                elementList.remove(e);
        }

        public void addElement(Element e){
                elementList.add(e);
        }

        public String toString(){
                return elementList.toString();
        }

        public boolean equalElements(Group otherGroup){
                return elementList.containsAll(otherGroup
                    .getElements());
        }

}
```

```
package perfectStranger.multiCore;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.util.ArrayList;
import java.util.Collections;
import java.util.ListIterator;


public class MatcherMulti implements Runnable{
private static String destFolder = "./tables/";
private static final Object lock = new Object();

private static int groupSizeGlobal = 2;
private static int minNumSubjectsGlobal = 2;     // should
    be >= 1
private static int maxNumSubjectsGlobal = 40;
//total runtime in milliseconds
private static long totalRuntimeGlobal = 30000;

private int groupSize;
private int minNumSubjects;
private int maxNumSubjects;
private long totalRuntime;

private long repeats = 0;
private static long[] iterations;

public static void main(String[] args) throws Exception {
        //generate destination Folder if not existent
        File dir = new File(destFolder);
        dir.mkdirs();

        String log;
        //configure parameters
        readCommandLine(args);

        //dynamic core number
        int cores = Runtime.getRuntime().
            availableProcessors();
//              int cores = 8;
```

```
        //generate array to save iterations
        int numConfigurations = (maxNumSubjectsGlobal -
            minNumSubjectsGlobal)/groupSizeGlobal + 1;
        iterations = new long[numConfigurations];

        //setup threads and start perfect stranger
            searching process
        ArrayList<Thread> threads = new ArrayList<Thread
            >();
        for(int c = 0; c < cores; c++){
        Thread thread = new Thread(new MatcherMulti(
            groupSizeGlobal, minNumSubjectsGlobal,
            maxNumSubjectsGlobal, totalRuntimeGlobal));
        threads.add(thread);
        thread.start();
        }
        System.out.println("All " + cores + " threads
            have been started.");

        //wait for threads to finish
        for(int i = 0; i < threads.size(); i++){
        threads.get(i).join();
        }

        //all threads have finished, write logfile
        System.out.println("---Calculation finished for
            groupSize " + groupSizeGlobal +", iterations
            for each configuration are:---");
        log = "---Calculation finished for groupSize " +
            groupSizeGlobal +", iterations for each
            configuration are:---\r\n";
        for(int i = iterations.length-1; i >= 0; i--){
        System.out.println("subjects: " + (
            maxNumSubjectsGlobal - i*groupSizeGlobal) + ",
             iterations " +  iterations[i]);
        log += "subjects: " + (maxNumSubjectsGlobal - i*
            groupSizeGlobal) + ", iterations " +
            iterations[i] + "\r\n";
        }
        writeToLog(log, destFolder + "logfile.txt");

        System.out.println("--- Calculation finished,
            press button to close ---");
        System.in.read();
}
```

```
//reads groupSize from command line parameters
public static void readCommandLine(String[] args){
        for(int i = 0; i < args.length; i++){
        //reads groupSize from command line parameters
        if(args[i].startsWith("-g=")){
        groupSizeGlobal = Integer.parseInt(args[i].
            substring(3));
        }
        //reads runtime from command line parameters
        if(args[i].startsWith("-r=")){
        totalRuntimeGlobal = Long.parseLong(args[i].
            substring(3));
        }
        //reads minNumSubjects from command line
            parameters
        if(args[i].startsWith("-min=")){
        minNumSubjectsGlobal = Integer.parseInt(args[i].
            substring(5));
        }
        //reads minNumSubjects from command line
            parameters
        if(args[i].startsWith("-max=")){
        maxNumSubjectsGlobal = Integer.parseInt(args[i].
            substring(5));
        }
        }
}

/**
 *
 * @param groupSize defines groupSize for perfect
    stranger matching
 * @param minNumSubjects defines minimum subject number
    for perfect stranger matching (for given groupsize)
 * @param maxNumSubjects defines maximum subject number
    for perfect stranger matching (for given groupsize)
 * @param totalRuntime defines the total runtime for all
    subject numbers
 */
public MatcherMulti(int groupSize,int  minNumSubjects,int
     maxNumSubjects,long totalRuntime){
        this.groupSize = groupSize;
        this.minNumSubjects = minNumSubjects;
        this.maxNumSubjects = maxNumSubjects;
        this.totalRuntime = totalRuntime;
```

```
}

@Override
public void run() {
        try {
        generateAllBestMatches(groupSize, minNumSubjects,
            maxNumSubjects, totalRuntime);
        } catch (IOException e) {
        e.printStackTrace();
        }
}

//fills a group given a list of possible partners
//first (fixed) element already in group
//possiblePartners should be a cloned version since
    operations are performed on it
public Group fillGroup(ArrayList<Element>
    possiblePartners, Group group, int groupSize,ArrayList
    <Group> lastGroup){

        //variable initialization
        int size = group.getElements().size();
        Group newGroup;
        Element newPartner;
        Element currentElement;
        ArrayList<Element> availableElements;

        //group not filled yet
        if(size == groupSize){
        //reconstruct recursion if last group is given
        if(lastGroup.size() == 0){
        return group;
        }else{
        if(group.equalElements(lastGroup.get(0))){
        lastGroup.clear();
        return null;
        }
        }
        }

        //accelerate group reconstruction
        if(lastGroup.size() != 0){
        //get next element
        newPartner = lastGroup.get(0).getElements().get(
            size);
        while(possiblePartners.size() > 0){
```

```
currentElement = possiblePartners.get(0);

//first non constant element
if( !newPartner.equals(currentElement) ){
//remove first element
possiblePartners.remove(0);
}else{
//do not remove next element!
break;
}
}
}

//check if enough elements are available to fill
    group
if(possiblePartners.size() < groupSize-size ){
return null;
}

//fetch new partner
while(possiblePartners.size() > 0){
//get first available element
newPartner = possiblePartners.get(0);
availableElements = (ArrayList<Element>)
    possiblePartners.clone();

//actualize possiblePartner list (excludes itself
    )
availableElements.retainAll(newPartner.
    getPartnerList());

//add Partner to group
group.addElement(newPartner);

newGroup = fillGroup(availableElements,group,
    groupSize,lastGroup);

if(newGroup == null){
//remove last group partner
possiblePartners.remove(newPartner);
group.removeElement(newPartner);
}else{
return newGroup;
}
}
```

```
        return null;
}


//perfectStranger rematching
public ArrayList<Group> rematch(ArrayList<Element>
    unusedElements, ArrayList<Group> groupList, int
    groupSize){
        //variable initialization
        ArrayList<Element> possiblePartners;
        ArrayList<Element> possiblePartnersClone;
        ArrayList<Element> unusedElementsClone;
        ArrayList<Group> newGroupList;
        Element pivotElement;
        Group currentGroup;
        ArrayList<Group> lastGroup = new ArrayList<Group
            >();
        int groupNumber = (int) (unusedElements.size()/((
            double) groupSize));

        //check if recursion finished
        if(groupNumber == 0){
        return groupList;
        }

        //get first element (constant element)
        pivotElement = unusedElements.get(0);

        //get possible Partners (cloning is important)
        possiblePartners = (ArrayList<Element>)
            pivotElement.getPartnerList().clone();
        possiblePartners.retainAll(unusedElements);
                    //excludes itself

        while(possiblePartners.size() >= groupSize -1){
        newGroupList = null;
        possiblePartnersClone = (ArrayList<Element>)
            possiblePartners.clone();
        //add constant element to new group
        currentGroup = new Group();      //counting
            backwards
        currentGroup.addElement(pivotElement);

        //fill group
        currentGroup = fillGroup(possiblePartnersClone,
            currentGroup, groupSize,lastGroup);
```

```
            //check for break condition
            if(currentGroup == null){
            return null;
            }else{
            //add group to groupList
            groupList.add(currentGroup);

            //actualize list of unused elements
            unusedElementsClone = (ArrayList<Element>)
                unusedElements.clone();
            unusedElementsClone.removeAll(currentGroup.
                getElements());

            //next recursion
            newGroupList = rematch(unusedElementsClone,
                groupList, groupSize);
            }

            //check if rematch is successful under current
                group assignment
            if(newGroupList == null){
            //lastGroup.clear() not necessary
            //set lastGroup as starting point
            lastGroup.add(currentGroup);
            groupList.remove(currentGroup);
            }else{
            return newGroupList;
            }
            }

            return null;
    }


    //assign new group membership
    public void actualizeGroups(ArrayList<Group> groupList){
            ArrayList<Element> elementList;

            for(Group gr : groupList){
            elementList = gr.getElements();

            //actualize known partners for all elements
                inside
            for(Element el : elementList){
            el.getPartnerList().removeAll(elementList);
```

```
        }
        }
}

//pass empty ArrayLists for allElements and allGroups
    which will be filled by this method
public void initializeMatcher(ArrayList<Element>
    allElements, ArrayList<Group> allGroups, int
    numSubjects, int groupSize){
        Element currentElement = null;
        ArrayList<Element> possiblePartner;

        //initialize groups
        for(int g = 0; g < numSubjects/groupSize*1.0; g
            ++){
        allGroups.add(new Group());
        }

        //initialize subjects
        for(int e = 0; e < numSubjects; e++){
        allElements.add(new Element(e));
        }

        //initialize possiblePartner for all elements (
            cloning important!)
        for(Element e2 : allElements){
        possiblePartner = (ArrayList<Element>)
            allElements.clone();
        e2.setPartnerList(possiblePartner);
        }

        //first group allocation
        ListIterator<Element> iter = allElements.
            listIterator();
        for(Group gr : allGroups){
        for(int s = 0; s < groupSize; s++){
        currentElement = iter.next();
        gr.addElement( currentElement );
        }
        }

        //possible partner matches in future
        actualizeGroups(allGroups);
}

/*
```

```
 * generates best matching, calculates for x milli
    seconds (specified by runtime)
 * @input: number of subjects has to be a multiple of
    groupSize
 */
public ArrayList<ArrayList<Group>> generateBestMatching(
    int groupSize, int numSubjects, long runtime) throws
    IOException{
        //variable definition
        ArrayList<ArrayList<Group>> allMatches = new
            ArrayList<ArrayList<Group>>();
        long startTime = System.currentTimeMillis();
        this.repeats = 0;

        //calculate for a specified time
        while(System.currentTimeMillis() - startTime <
            runtime){
        //apply weak boundary
        if(allMatches.size() >= (numSubjects-1)/(
            groupSize-1)){
        break;
        }
        //weak condition, no more matches possible
        if(numSubjects/groupSize < groupSize &&
            allMatches.size() >= 1){
        break;
        }

        //variable initialization
        ArrayList<ArrayList<Group>> tempMatches = new
            ArrayList<ArrayList<Group>>();
        ArrayList<Element> allElements = new ArrayList<
            Element>();
        ArrayList<Group> groupList = new ArrayList<Group
            >();              //is not consistently stored

        //initialize first allocation
        initializeMatcher(allElements,groupList,
            numSubjects,groupSize);
        //already shuffle before the algorithm starts
        allElements = reorganizeElements(allElements);
        tempMatches.add(groupList);

        /**REMATCHING**/
        //rematch until no more matches are found
        while(groupList != null){
```

```
            //generate rematch
            groupList = rematch((ArrayList<Element>)
                allElements.clone(), new ArrayList<Group>(),
                groupSize);
            //change ordering of allElements and partnerLists
            allElements = reorganizeElements(allElements);

            if(groupList != null){
            //perform change
            actualizeGroups(groupList);
            tempMatches.add(groupList);
            }
            }

            //save best matching sequence
            if(allMatches.size() < tempMatches.size()){
            allMatches = tempMatches;
            }
            this.repeats++;
            }

            return allMatches;
}

/*
 * generates all best matching sequences for a specific
     groupSize, up to the specified max amount of subjects
 */
public void generateAllBestMatches(int groupSize, int
    minNumSubjects, int maxNumSubjects, long totalRuntime)
     throws IOException{
            long timeIncrement = (long) (totalRuntime / Math.
                ceil((maxNumSubjects-minNumSubjects+1)*1.0/
                groupSize));
            int remainingIter = (maxNumSubjects-
                minNumSubjects)/groupSize + 1;

            //best matching strategy
            for(int i = minNumSubjects/groupSize; i <=
                maxNumSubjects/groupSize; i++){
            int numOfSubjects = i*groupSize;

            ArrayList<ArrayList<Group>> strangerMatches =
                generateBestMatching(groupSize,numOfSubjects,
                timeIncrement);
            System.out.println("Best Strategy for " +
```

```
            numOfSubjects + " subjects and groupSize " +
            groupSize + " found " + strangerMatches.size()
             +" matches");

        //check data for best known strategy
        String pathname = destFolder + groupSize + "to" +
             numOfSubjects + ".txt";
        try{
        //file input
        ArrayList<ArrayList<Group>> bestMatches =
            readMatchesTable(pathname);

        //only output to file, if new solution is better
        if(strangerMatches.size() > bestMatches.size()){
        System.out.println("Better matching found!");
        saveMatchesTable(strangerMatches,pathname);
        }

        }catch(IOException e){
        System.out.println("No table named " + pathname +
             " found. Creating new table!");

        //file output
        saveMatchesTable(strangerMatches,pathname);
        }

        //actualize time increment (has an impact if
            calculation finished earlier)
        remainingIter--;
        iterations[remainingIter] += repeats;
        }
}


/*
 * changes ordering for allElements, as well as the lists
     of known partner for all elements
 */
public ArrayList<Element> reorganizeElements(ArrayList<
    Element> allElements){
        ArrayList<Element> allElementsNew = (ArrayList<
            Element>) allElements.clone();
        Collections.shuffle(allElementsNew);

        //iterate over allElement List
        for(Element el : allElementsNew){
```

```
            Collections.shuffle(el.getPartnerList());
            }

            return allElementsNew;
    }

    public void saveMatchesTable(ArrayList<ArrayList<Group>>
        allMatches, String pathName) throws IOException{
            PrintWriter out;
            int numRematches = allMatches.size();

            synchronized (lock) {
            //generate outputfile
            out = new PrintWriter(new FileWriter(pathName));
            out.println("# " + numRematches + " different
                group configurations");
        out.close();

        for(ArrayList<Group> groups : allMatches){

        //print specific match
            out = new PrintWriter(new FileWriter(pathName,
                true));
            out.println(groups);
            out.close();
        }
            lock.notifyAll();
            }
    }

    public ArrayList<ArrayList<Group>> readMatchesTable(
        String pathName) throws IOException{
            ArrayList<ArrayList<Group>> allMatches = new
                ArrayList<ArrayList<Group>>();
            ArrayList<Group> currentGroupList;
            Group currentGroup = null;
            Element currentElement;

            synchronized(lock){
            //read table as String
        String line = null;
            InputStreamReader isr = new InputStreamReader(new
                FileInputStream(pathName));
            BufferedReader in = new BufferedReader(isr);

        //iterate over all lines in .txt
```

```
    while ((line = in.readLine()) != null) {
        if( line.startsWith("[")){
        line = line.substring(1);

        currentGroupList = new ArrayList<Group>();
        allMatches.add(currentGroupList);

        //construct all groups for specific line
        while(line.length() > 0){
        if( line.startsWith("[")){
                currentGroup = new Group();
                currentGroupList.add(currentGroup);
        }

        //number detected (side-effect: all unicode
            digits are detected)
        if(Character.isDigit(line.charAt(0))){
        //check if a "two digit number" is present
        if(Character.isDigit(line.charAt(1))){
        currentElement = new Element( Character.
            getNumericValue(line.charAt(0))*10 + Character
            .getNumericValue(line.charAt(1)) );

        //substract one more char since two have been
            processed
        line = line.substring(1);
        }else{
        currentElement = new Element( Character.
            getNumericValue(line.charAt(0)) );
        }
        //add Element to group and link group
        currentGroup.addElement(currentElement);
        }

        //iterate to next char
        line = line.substring(1);
        }

        }
    }
    in.close();
    lock.notifyAll();
        }

    return allMatches;
}
```

```
public static void writeToLog(String text, String
    pathName) throws IOException{
        PrintWriter out;

        //generate outputfile
        out = new PrintWriter(new FileWriter(pathName,
            true));
        out.println(text);
     out.close();
}
}
```