Eindhoven University of Technology

2WH40: Bachelor Final Project

Department of mathematics and computer science

The Running Dinner Problem

Nienke de Jongh - 0999285
Supervisor: dr.ir. C.A.J. Hurkens

Combinatorial Optimization
August 21, 2019

EINDHOVEN UNIVERSITY OF TECHNOLOGY

Abstract

A running dinner is a dinner where (almost) none of the participants know each other and every course is eaten at a different house of one of the participants. This participant is called the host of that round. The participants are divided into equally sized groups that are different for every round. The Running Dinner Problem deals with constructing the maximal number of rounds for a given number of participants and a given table size, such that participants see each other once at most. It also deals with choosing a host for each round, such that the task of being host is distributed evenly among all participants. A perfect solution is one where every participant sees every other participant exactly once. There is a perfect solution is the following two instances: there is an even number of participants with table size 2 or the number of tables is p^e with table size p, where p is a prime number and e a positive integer. There is also a perfect solution for some pre-built cases with table size 3, 4 and 6. There are strategies for dealing with other instances in which the socialization is optimized. A multi-start randomized greedy algorithm can be used to select the hosts in each round of the running dinner, such that this role is divided fairly. The Socialization Problem is an expansion of the Running Dinner Problem that deals with choosing a predetermined number of rounds from the maximal number constructed in the Running Dinner Problem. The aim here is to select these rounds so that the socialization is maximal. This is generally a $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problem, depending on whether the input is a number of participants or a list of names of all the participants. However, this problem can be solved using a brute-force method that checks all orders of rounds and gives back the optimal order of the rounds, i.e. the order that yields the maximal socialization.

Contents

Chapter 1

Introduction

In 1850, the reverend Thomas Penyngton Kirkman proposed the following problem as Query VI in The Lady's and Gentleman's Diary (pg.48):

"*Fifteen young ladies in a school walk out three abreast for seven days in succession: it is required to arrange them daily so that no two shall walk twice abreast*" (Graham, Grötschel, & Lovász, 1995).

This problem is nowadays known as Kirkman's schoolgirls problem and, in simpler words, it asks to arrange fifteen schoolgirls in five rows of three girls for seven days such that no two girls walk in the same row more than once. The solution for the problem was published in the same year, but more questions arose from this problem. For instance, the general problem of arranging $n \equiv 3 \bmod 6$ girls into groups of three over a series of $\frac{n-1}{2}$ days and questions about tournament scheduling, such as the social golfer problem. This problem is derived from a question posted on sci.op-research in May 1998 (Triska, & Musliu, 2011). In the social golfer problem, a coordinator of a golfing club asks you to divide a number of social golfers who play golf once a week in groups of equal size, such that these golfers can play for as many weeks as possible without encountering another golfer more than once. A relatively new problem that is related to Kirkman's schoolgirls problem and the social golfer problem is the Running Dinner Problem (RDP).
The concept of a running dinner is quite simple: a dinner is organised for a group of people. These people are the participants of the running dinner. For each round, the participants are divided into smaller groups of equal size that eat together at the home of someone in this group. This person is the host for this round and will cook the dish for this group. After a round, all the participants are shuffled again and new groups are formed for the next dish. A new host is chosen in each group as well, preferably one who was not a host in the previous round. Because of this rotation system, it is possible to get to know a lot of new people in one evening, which is why running dinners are popular in, for example, neighbourhoods.
The Running Dinner Problem aims to arrange the participants of a running dinner into fixed-size groups for as many rounds as possible, such that every participant meets as many new people as possible without meeting the same people twice. It also aims to divide the task of being host as equally as possible and to minimize the number of times someone is host twice in a row.

Chapter 2

Problem Description

Formally, the RDP is described as the union of two tasks:

1. Find a seating arrangement for n participants over tables of size t for as many rounds as possible, such that no two participants meet more than once.
2. Select a host for each round, such that the role of host is distributed uniformly among all participants.

An instance of the RDP is denoted as $n - t - r_{max}$ with n the number of participants, t the table size and r_{max} the maximal number of rounds for that instance. We define a perfect solution as one where every participant sees every other participant exactly once. Of course, our aim is to find such a perfect solution, if it exists, since this solution will provide the maximal socialization. A necessary condition for an RDP having a perfect solution is that $n - 1$ should be a multiple of $t - 1$ and, if this perfect solution exists, the number of rounds is $r_{max} = \frac{n-1}{t-1}$. Furthermore, the maximal number of rounds is $r_{max} = 1$ if n is less than t^2, since a second encounter between participants is inevitable otherwise (Hurkens, n.d.). In addition, when we do not have a perfect solution, we would like to find a strategy that ensures as much socialization as possible.
Besides being interesting for running dinners, the solution to the RDP can be useful in other instances as well. For example, when we see the tables as groups and the rounds as weeks, it provides a solution to the social golfer problem. When the tables are of size three and the number of participants is $n \equiv 3 \bmod 6$, it is equal to the generalization of Kirkman's schoolgirls problem.

In this report we will mostly focus on the first task of the RDP and only shortly mention a way to solve the second task. As we will see later, the solution to the RDP usually consists of a considerable list of rounds. When we realize that every round comes with a dish, it may be desirable for the organizer and the participants to limit the amount of rounds, to prevent the running dinner from taking too long. Only a set number of rounds may be chosen from the initial list, in such a way that the socialization is still optimal for these rounds. This is the second problem that will be discussed in this report and it is defined as the Socialization Problem.

Chapter 3

The Running Dinner Problem

It is clear that there is not always a perfect solution to the running dinner problem. This already becomes apparent if the number of participants is not divisible by the table size. Now, the question is whether we can somehow construct perfect solutions and if not, how we can ensure that the solution is as close to optimal as possible. We will particularly base this on a program that was written by Siervo (2017). In the paper she wrote about it, she explains the combinatorial theory behind the problem and how this can be used to construct perfect solutions. She also explains what the program does when it cannot find a perfect solution.

Firstly, we will introduce a few definitions, theorems and propositions that are used by Siervo to construct her program. We will cover block designs, resolvable designs, Steiner triple systems, Kirkman triple systems, mutually orthogonal latin squares and mutually orthogonal latin rectangles. The definitions, theorems and propositions that are presented here are only a small selection of those introduced in Siervo's paper. The reader is encouraged to read this paper, which elaborates on these concepts in more detail and provides proofs for some of the theorems and propositions.

3.1 Definitions, Theorems and Propositions

3.1.1 Block Designs and Resolvable Designs

Definition 3.1: A **design** is a collection B of b subsets (called blocks) on a finite set V of v objects called **varieties**. So

$$B = \{B_1, B_2, ..., B_b\} \text{ with } B_i \subset V \text{ for all } i = 1, ..., b.$$

Definition 3.2: A design is **complete** of if each block consists of all of V. If at least one block is not all of V, the design is **incomplete**.

Definition 3.3: A design is called k**-uniform** if each block has the same size k.

Definition 3.4: A design is called a **balanced** design of **index** λ if each pair of distinct varieties appears together in exactly λ blocks.

Definition 3.5: A design is said to be r**-regular** if each variety appears in the same number r of blocks. The number of blocks a variety appears in is called the **replication number**.

Definition 3.6: A design is called a **block design** if it is both uniform and regular.

Definition 3.7: A k-uniform r-regular block design with v varieties arranged into b blocks that is also balanced of index λ is called a (v, b, r, k, λ) design. If $k < v$, then the (v, b, r, k, λ) design is called a **balanced incomplete block design (BIBD)**.

Proposition 3.8: Necessary conditions for a (v, b, r, k, λ) design are:

$$r(k-1) = \lambda(v-1)$$
$$vr = bk.$$

These justify why (v, b, r, k, λ) designs are often referred to as (v, k, λ) designs.

Theorem 3.9 (Fisher's inquality): A necessary condition for a (v, b, r, k, λ) BIBD is $b \geq v$.

Definition 3.10: A (v, b, r, k, λ) design is **resolvable** of its blocks can be partitioned into sets of blocks, each of which is a partition of V. Each of these sets of blocks is called a **parallel class**. The partition of blocks into classes is called a resolution.

Theorem 3.11: For each prime p and each positive integer $e \geq 1$ there exists a resolvable $(p^e, p, 1)$ design.

Theorem 3.12: If B is a resolvable (v, b, r, k, λ) design, then a resolution of B into r parallel classes is a solution for the running dinner problem instance $v - k - r$.

3.1.2 Steiner and Kirkman Triple Systems

Definition 3.14: A **Steiner triple system** STS(v) of order v is a (v, 3, 1) design.

Theorem 3.15: A STS(v) with $v \geq 3$ exists if and only if $v \equiv 1$ mod 6 or $v \equiv 3$ mod 6.

Definition 3.16: A resolvable STS(v) is called a **Kirkman triple system** KTS(v) of order v.

Theorem 3.17: A KTS(v) exists if and only if $v \equiv 3$ mod 6.

3.1.3 Mutually Orthogonal Latin Squares and Rectangles

Definition 3.18: A **latin square** of order n is an $n \times n$ array in which n distinct symbols from a symbol set S are arranged, such that each symbol occurs exactly once in each row and each column.

Definition 3.19: A $k \times n$ **latin rectangle** (where $k \leq n$) is a $k \times n$ array in which n distinct symbols from a symbol set S are arranged, such that each symbol occurs exactly once in each row and at most once in each column.

Definition 3.20: Two latin squares $A = (a_{ij})$ on a symbol set S_A and $B = (b_{ij})$ on a symbol set S_B of order n are **orthogonal** if every element in $S_A \times S_B$ occurs exactly once among the n^2 pairs (a_{ij}, b_{ij}), $1 \leq i, j \leq n$.

Definition 3.21: A set of latin squares $\{L_1, ..., L_m\}$ is a set of **mutually orthogonal latin squares** (MOLS) if L_i and L_j are orthogonal for all $i, j \in \{1, ..., m\}(i \neq j)$. The maximal number of MOLS of order n is denoted by $N(n)$.

Definition 3.22: A set of $k \times n$ latin rectangles $\{R_1, ..., R_m\}$ is a set of **mutually orthogonal latin rectangles** (MOLR) if R_i and R_j are orthogonal for all $i, j \in \{1, ..., m\}(i \neq j)$. The maximal number of MOLR(k, n) is denoted by $N(k, n)$.

Proposition 3.23: If a set of m MOLS of order n exists, then it gives a solution for the running dinner problem instance $n^2 - n - (m + 2)$.

Proposition 3.24: If a set of m MOLR of size $k \times n$ exists, then it gives a solution for the running dinner problem instance $kn - k - r$ with at least $r = m + 1$.

Definition 3.25: A set of $n - 1$ MOLS of order n is a **complete** set of MOLS.

Theorem 3.26: If $q = p^e$ is a prime power then there exists a complete set of MOLS of order q, that is $N(q) = q - 1$.

Theorem 3.27: The running dinner problem with p^e participants and tables of size p, where p is a prime number and e is a positive integer, has a perfect solution.

3.2 The Running Dinner Problem Program

Now, we will go over how Siervo uses these theoretical constructs in her program. Especially Theorems 3.26 and 3.27, together with propositions 3.23 and 3.24, explain how to construct the seating arrangements when the number of tables is a prime or prime power. Siervo also argues that there is a perfect solution for n participants and table size 2, where n is an even positive integer. This solution is based on tournament scheduling. Unfortunately, we often do not have these perfect conditions. In these cases, it is necessary to adopt different strategies such that the solution to the RDP is as close to perfect as possible.
In fact, the program adopts four different strategies when the number of tables is not a prime power and thus does not follow the theoretical constructs for a perfect solution. These strategies all divide the problem into smaller problems that do contain primes en prime powers.

1. When $NumberOfTables = p^e \cdot TableSize$ with p^e a prime power and $p^e \geq TableSize$, then the participants are split into $TableSize$ groups of $p^e \cdot Tablesize$ people. Now, the number of tables in each group is a prime power, so we can apply the theoretical constructs obtained. This means that, if the participants were arranged in a $TableSize \times NumberOfTables$ array, in the end every participants meets the people in their row and their column at most once, while they never meet the remaining participants.

2. When the number of tables is not divisible by $TableSize$, but it can be written as $NumberOfTables = p^e \cdot m$ with $m < TableSize$ and p^e a prime power with $p^e \geq TableSize$, then we can apply a strategy similar to the first one. The difference is that we arrange the participants in an $m \times (p^e \cdot TableSize)$ array, where those in the same row are added to the same group. Again, the construction of the rounds is known for each group because the number of tables is a prime power. In the end, every participant has met every other participant in their row, but has met none of the remaining participants. Here, m is defined inversely proportional to the socialization, meaning that the smaller m is, the bigger the socialization will be in this strategy. This makes sense, since the smaller m is, the closer $NumberOfTables$ is to a prime power and, thus, to a perfect solution.

3. When $NumberOfTables = q_1 + q_2$, where q_1 and q_2 are prime powers with q_1 and q_2 close to each other and $q_1, q_2 \geq TableSize$, the participants can be divided into two groups of $q_i \cdot TableSize$ each $(i = 1, 2)$. The rounds are then constructed for each of these groups. We require the distance between q_1 and q_2 to be small, since the groups will be more equally

sized then. This gives better socialization for all participants in contrast to the larger group having much better socialization than the smaller group.

4. When none of these strategies can be applied, a number of dummy variables is added such that the number of participants eventually falls into the solvable cases. A dummy variable is a "person" that is added to the list of guests to make the number of participants better for theoretical results. After the program has implemented the solution with this new number of participants, it removes the dummy variable and classifies the place where the dummy variable is seated in a round as an empty seat. This is not the most elegant solution, but it is necessary in some cases.

The program prioritizes the first strategy and determines whether to use the second or third strategy based on which provides the better socialization. The fourth strategy is only used when no other strategy is possible. These strategies do not provide perfect solutions, since they are based on splitting the participants in groups, but they do give some good results when there is no perfect solution possible due to the number of participants and the table size.
The program also has a few pre-built cases, especially for the tables of size 3,4 or 6. It uses, among others, Kirkman triple systems to construct these pre-built cases.
Siervo also introduces a multi-start randomized greedy algorithm for the choice of hosts. This part of the program selects a host for each round and rearranges the rounds such that the distribution of the choice of host is uniform among all participants (for more details, see Siervo, 2017)

3.2.1 Examples

We will now provide some examples to illustrate how the program works. To this end, we first take Kirkman's schoolgirls problem. Recall that this problem was introduced by Kirkman in 1850 and can be translated to a running dinner problem with 15 participants and tables of size 3. The output is shown in table 3.1.

Round	**Table 1**	**Table 2**	**Table 3**	**Table 4**	**Table 5**
1	1, 5, 14	2, 4, 15	3, 8, 12	6, 9, 11	7, 10, 13
2	11, 1, 15	8, 6, 2	14, 7, 3	4, 9, 10	5, 12, 13
3	10, 8, 1	13, 11, 2	9, 5, 3	12, 4, 14	15, 7, 6
4	1, 9, 13	2, 7, 12	3, 4, 11	5, 8, 15	6, 10, 14
5	7, 4, 1	10, 5, 2	13, 6, 3	8, 11, 14	9, 12, 15
6	12, 6, 1	14, 9, 2	15, 10, 3	4, 8, 13	11, 7, 5
7	1, 2, 3	5, 4, 6	7, 8, 9	10, 11, 12	13, 14, 15

Table 3.1: RDP solution with 15 participants and tables of size 3.

Each of the numbers 1 to 15 in the table corresponds to one of the participants. The numbers are ordered in such a way that the first person at each table is the host for that round. We do not use this information in the rest of the report, but it is important to note Siervo's program has the possibility of generating a host for each round. For Kirkman's schoolgirls problem, the rounds refer to days and the tables to groups of girls.

Secondly, we will look at an example that uses dummy variables. To this end, we take 33 participants with table size 5. The output of the program is shown in table 3.2.

Round	**Table 1**	**Table 2**	**Table 3**	**Table 4**	**Table 5**	**Table 6**	**Table 7**
1	1, 8, 15, 22, 29	2, 9, 16, 23, 30	3, 10, 17, 24, 31	4, 11, 18, 25, 32	5, 12, 19, 26, 33	6, 13, 20, 27, −	7, 14, 21, 28, −
2	14, 1, 20, 26, 32	8, 2, 21, 27, 33	9, 3, 15, 28, −	10, 4, 16, 22, −	11, 5, 17, 23, 29	12, 6, 18, 24, 30	13, 7, 19, 25, 21
3	16, 12, 1, 27, 31	17, 13, 2, 28, 32	18, 14, 3, 22, 33	19, 8, 4, 23, −	20, 9, 5, 24, −	21, 10, 6, 25, 29	15, 11, 7, 26, 30
4	24, 11, 21, 1, −	25, 12, 15, 2, −	26, 13, 16, 3, 29	27, 14, 17, 4, 30	28, 8, 18, 5, 31	22, 9, 19, 6, 32	23, 10, 20, 7, 33
5	33, 9, 17, 25, 1	2, 10, 18, 26, −	3, 11, 19, 27, −	29, 12, 20, 28, 4	30, 13, 21, 22, 5	31, 14, 15, 23, 6	32, 8, 16, 24, 7
6	1, 13, 18, 23, −	14, 2, 19, 24, 29	8, 3, 20, 25, 30	4, 9, 21, 26, 31	5, 10, 15, 27, 32	6, 11, 16, 28, 33	7, 12, 17, 22, −
7	10, 1, 19, 28, 30	11, 2, 20, 22, 31	12, 3, 21, 23, 32	13, 4, 15, 24, 33	16, 14, 5, 25, −	17, 8, 6, 26, −	9, 7, 18, 27, 29

Table 3.2: RDP solution with 33 participants and tables of size 5.

Here, each of the numbers 1 to 33 in the table corresponds to one of the participants. Again, the first person at each table is the host of the round. The $'-'$ symbol refers to a dummy variable, i.e. an empty seat. We can see the program added two dummy variables so there is a total number of 35 participants, which can be divided by the table size 5. However, even with 35 participants and a table size of 5, we do not have the perfect solution, since 35 is not a prime power of 5. That is why the program chooses the appropriate strategy to split the problem into sub-problems.

Chapter 4

The Socialization Problem

We have now found the seating arrangements for all numbers of participants and table sizes. However, as illustrated by the examples mentioned in the previous chapter, the maximal number of rounds is often not feasible for a running dinner setting in real life. Unless the courses are really small and every round takes roughly less than 45 minutes, seven rounds is a bit optimistic for only one evening. This is why the organiser of the event may want to select a smaller number of rounds from this maximal number or construct some rounds himself, so the set-up is more achievable. Some sets of rounds might be better than other sets in terms of socialization of participants. This gives rise to the question how the rounds can be chosen or constructed in such a way that, in the end, the optimal socialisation is achieved. In this case, optimal socialization means that participants have as much information about other participants as possible.
We will call this problem the "Socialization Problem". We assume the acquaintance between two participants can be characterised by a value. This value will be in $\mathbb{N}$. It will be very large when two participants do not know each other at all and smaller if the participants have any kind of connection. When we talk about the acquaintance of a participant with himself, we assume that the value is 0. We also assume that a participant talks about all other people they know, either directly or indirectly (heard of from other people in previous rounds), to all other participants at their table in a new round. We define $A(r, i, j)$ as the acquaintance of participant i regarding participant j, so the extent to which i knows of j after round r. Note that this is different than $A(r, j, i)$, since i may have heard of j from other participants, but j may not have heard of i. When participant i sits at a table with participant k in round r, we get the following recursion

$$A(r,i,j) = \begin{cases} 1 & \text{if i and j sit at the same table in round r} \\ \min\{A(r-1,i,j), \min\limits_{k}\{A(r-1,k,j)+1\}\} & \text{else.} \end{cases} \tag{4.1}$$

We can define the score for a predetermined number of rounds $\hat{r}$ with this definition. After $\hat{r}$ rounds, we can make a tree for every participant j. The participants are the vertices in this tree and there is an edge from one participant to the other if they know each other directly, i.e. with acquaintance 1. If there is a path from participant j to participant i, then $A(\hat{r}, i, j)$ is the number of edges of the path. If there is no path from participant j to participant i, then i has not heard of j at all. In this case, $A(\hat{r}, i, j) = M$, where M is a large number that can be chosen depending on the number of participants and table size of the running dinner. We define the score as follows:

$$S(\hat{r}) = \sum_{i=1}^{n} \sum_{j=1}^{n} A(\hat{r}, i, j) = l \cdot M + s. \tag{4.2}$$

We define l as all $i.j$: $\nexists_{\text{path}} i \to j$. We define s as $\sum_{i,j:\exists_{\text{path}} i \to j} A(\hat{r}, i, j)$.

We want to minimize $S(\hat{r})$ in the Socialization Problem, since this will yield the optimal socialization. To do this, we first want l to be as small as possible, since M will be a big contributor to a higher score. In particular, when we choose M to be very large. Furthermore, it is also important for s to be small, since shorter paths make the score lower. However, the emphasis lies on a small l.
When the table size is 2, the Socialization Problem has some similarities with the famous gossip problem.

4.1 The Gossip Problem

4.1.1 The Problem

In the literature, the gossip problem is usually introduced as follows:

"*There are n agents each of which knows some secret not known to anybody else. Two agents can make a telephone call and exchange all secrets they know. How many calls does it take to share all secrets, i.e., how many calls have to take place until everybody knows all secrets?*" (Herzig, & Maffre, 2017).

In our case, the secret is the information about a participant. The participants start off with only their own information and gain more information as the rounds progress. As mentioned before, the gossip problem provides insights about the Socialization Problem with table size 2, where two agents sit at the same table when they call each other. However, the gossip problem is not exactly like the Socialization Problem. First of all, in the Socialization Problem, we do not want to know how many rounds it will take before all participants know each other. Rather, we require that the socialisation is biggest. Translated to the gossip problem, this means that in a predetermined number of calls, the agents all learn as many secrets as possible. Furthermore, in the gossip problem, there are no real "values". An agent either knows a secret or not. The chain of agents from which the secret was eventually acquired does not matter. Nonetheless, we gain some insights from the solution to the gossip problem.

4.1.2 The Solution

Let $f(n)$ denote the minimal number of calls needed if there are n agents. It is immediately clear that $f(1) = 0$, since an agent does not need to call anyone if he is the only one that knows gossip, and $f(2) = 1$, since it will only require one phone call from two agents to learn each others secrets. In addition, $f(3) = 3$. We can see this as follows: let g_i denote agent i's secret. It is easy to see that $f(3) \neq 1$, since there is always one agent that does not know any secrets after one call. Furthermore, the two agents that called each other do not know the secret of the agent that did not make a phone call yet. Now suppose that $f(3) = 2$. Without loss of generality, we assume that agent 1 calls agent 2 in the first phone call and agent 1 calls agent 3 in the second phone call. We get:

1. 1 calls 2: 1 tells g_1, and 2 tells g_2.
2. 1 calls 3: 1 tells g_1 and g_2, and 3 tells g_3.

After these calls, agent 1 and agent 3 both know all secrets, but agent 2 does not know s_3 yet. Thus, a third call is needed to ensure agent 2 knows every secret. This holds for all possible combinations of the first two calls, so $f(3) = 3$. The base cases $n = 2$ and $n = 3$ are illustrated in figure 4.1, where the numbers correspond to the agents and the lines indicate a phone call between two agents. The phone calls are shown in order from left to right (so the first phone call is the most left and the last the most right).

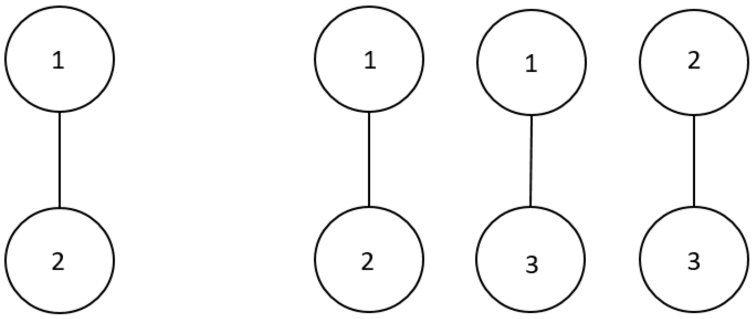


Figure 4.1: The gossip problem for $n = 2$ (l) and $n = 3$ (r).

It has been proven many times over that $f(n) = 2(n-2)$ for $n \geq 4$ (Hurkens, 2000; Baker, & Shostak, 1971; Tijdeman, 1971). Thus, shared knowledge can be achieved in $2(n-2)$ phone calls and it is impossible to do better.
We can define a round to be a set of simultaneous phone calls. It can be proved that at least $\lceil \log_2 n \rceil$ rounds are needed for an even number of agents and at least $\lceil \log_2 n \rceil + 1$ for an odd number of agents (Knödel, 1975). However, note that this is a lower bound. It is not necessarily true that all agents know each others secrets in $\lceil \log_2 n \rceil$ or $\lceil \log_2 n \rceil + 1$ rounds.
Nothing can be said yet about the values these participants have in relation to each other as well. Nonetheless, we have gained a little information about the Socialization Problem with table size 2. To find a more concrete result, we will look at an example with 27 participants and table size 3 in the next section.

4.2 The Socialization Problem with 27 Participants and Tables of Size 3

Assume we have an RDP with 27 participants and table size 3. We can subdivide these participants into 3 groups of 9 as illustrated in table 4.1.

1	2	3
4	5	6
7	8	9

10	11	12
13	14	15
16	17	18

19	20	21
22	23	24
25	26	27

Table 4.1: Subdivision of 27 participants into 3 groups of 9.

Now, suppose we want to know how many rounds it will take before every participant knows every other participant. It is easy to see that everyone knows everyone when we first choose the rows of the groups as tables, then the columns, and then we let every participant of a group sit with two participants from the other two groups. This is illustrated in table 4.2.

Round	**Table 1**	**Table 2**	**Table 3**	**Table 4**	**Table 5**	**Table 6**	**Table 7**	**Table 8**	**Table 9**
1	1,2,3	4,5,6	7,8,9	10,11,12	13,14,15	16,17,18	19,20,21	22,23,24	25,26,27
2	1,4,7	2,5,8	3,6,9	10,13,16	11,14,17	12,15,18	19,22,25	20,23,26	21,24,27
3	1,10,19	2,11,20	3,12,21	4,13,22	5,14,23	6,15,24	7,16,25	8,17,26	9,18,27

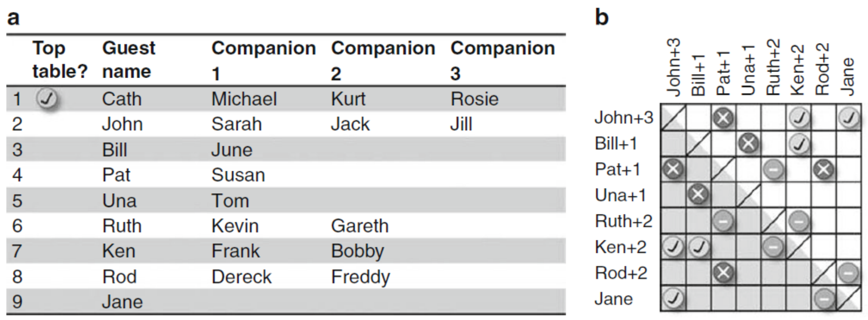
Table 4.2: Solution to Socialization Problem with 27 participants and table size 3, where three rounds are constructed.

We can see that after two rounds, everyone knows everyone in their group with maximal distance 2 and after three rounds, everyone knows everyone with maximal distance 3.
After this introduction to the problem, the question arises whether we can say something about the general problem for all numbers of participants and table sizes.

4.3 Seating Arrangements

4.3.1 The Problem

Lewis and Carroll (2016) are faced with a similar problem to the Socialization Problem. They want to organize a wedding with different groups of guests. Each group likes the other groups a certain amount. They either should either be seated 'definitely apart', 'rather apart' or 'rather together'. All these options have different values and thus all the groups have different values relative to the other groups. The option 'rather together' has the lowest value, which is -1, while the options 'rather apart' and 'definitely apart' have the values 1 and ∞ respectively. The organizers of the wedding can indicate this using a cross for 'definitely apart', a minus for 'rather apart' and a check for 'rather together'. This is illustrated in figure 4.2. The empty cells indicate that no preference was specified.



a

	Top table?	Guest name	Companion 1	Companion 2	Companion 3
1	✓	Cath	Michael	Kurt	Rosie
2		John	Sarah	Jack	Jill
3		Bill	June		
4		Pat	Susan		
5		Una	Tom		
6		Ruth	Kevin	Gareth	
7		Ken	Frank	Bobby	
8		Rod	Dereck	Freddy	
9		Jane			

b

	John+3	Bill+1	Pat+1	Una+1	Ruth+2	Ken+2	Rod+2	Jane
John+3			✗			✓		✓
Bill+1				✗		✓		
Pat+1	✗				−		✗	
Una+1		✗						
Ruth+2			−			−		
Ken+2	✓	✓			−			
Rod+2			✗					−
Jane	✓						−	

Figure 2 Specification of guest groups (left) and seating preferences (right).

Figure 4.2: Example with specification of guest groups (left) and seating preferences (right). Retrieved from Lewis and Carroll (2016).

The goal of their program is to create an optimal seating arrangement, where people are seated optimally according to their preferences.

4.3.2 Complexity of the Problem

First, the simplest form of the problem is considered. This is an $n \times n$ binary matrix $\mathbf{W}$ is constructed where

$$W_{ij} = \begin{cases} 1 & \text{if guests } i \text{ and } j \text{ are required to be sat at different tables} \\ 0 & \text{otherwise.} \end{cases}$$

However, in practical situations it can be desirable that users can place importance on their seating preferences. For instance, they could indicate that they prefer to sit with guest A over guest B while they might actually like both guests. To this end, it can be useful to adapt $\mathbf{W}$ to be an integer or real-valued matrix instead of binary. In this case, lower values of W_{ij} signify an increased preference for i and j to be sat together.
Furthermore, Lewis and Carroll assume that $W_{ij} = W_{ji}$. Given this input matrix, the task is then to partition the n guests into k subsets $\mathcal{S} = \{S_1, ..., S_k\}$ such that the objective function $f(\mathcal{S})$, given by

$$f(\mathcal{S}) = \sum_{t=1}^{k} \sum_{\forall i.j \in S_t : i<j} W_{ij}, \tag{4.3}$$

is minimised. Even for the binary matrix, the problem of confirming the existence of a zero cost solution to this problem is equivalent the $\mathcal{NP}$-complete graph k-colorability problem, in which each guest corresponds to a vertex and two vertices v_i and v_j are considered adjacent if and only if $W_{ij} = 1$. The k-colorability problem and the $\mathcal{NP}$-completeness of it are explained in more detail in the section below. For the integer or real-valued matrix, the problem of partitioning the guests over k tables becomes equivalent to the edge-weighted graph coloring problem. This is $\mathcal{NP}$-complete as well, as it generalizes the graph k-colorability problem. This will also be explained later.

In the second approach, a graph $G = (V, E)$ is constructed. Each vertex $v \in V$ represents a guest group with size s_v. This means that the total number of guests is $n = \sum_{v \in V} s_v$. Each edge $\{u, v\} \in E$ in the graph defines the relationship between the guest groups u and v according to a weigthing w_{uv} (in this case $w_{uv} = w_{vu}$), where smaller weightings indicate that u and v would like to be assigned to the same table.
A solution to the problem is then defined as a partition of the vertices into k subsets
$\mathcal{S} = \{S_1, ..., S_k\}$ such that

$$\bigcup_{i=1}^{k} S_i = V, \text{ and} \tag{4.4}$$

$$S_i \cap S_j = \emptyset \quad \forall i, j \in \{1, ..., k\}, i \neq j. \tag{4.5}$$

The number k denotes the number of tables and S_i defines the guests assigned to table i. They now construct an objective function similar to equation 4.3:

$$f_1 = \sum_{i=1}^{k} \sum_{\forall u,v \in S_i : \{u,v\} \in E} (s_v + s_u) w_{uv}. \tag{4.6}$$

This function reflects the extent to which the rules that state who sits with whom are obeyed. The weighting w_{uv} is multiplied by the total size of the two guest groups s_v and s_u involved. This is done so violations with larger numbers of people contribute more to the cost. Note that the group size s_v is equal to one for the RDP and Socialization Problem, since the participants do not travel in groups. Lewis and Carroll go on to state that when this is the case ($s_v = 1\ \forall v \in V$), then the number of guests assigned to each table i will equal the number of vertices in the group S_i. Hence, the problem will now be equivalent to the $\mathcal{NP}$-hard equitable weighted k-coloring problem, which is in itself already a generalization of the $\mathcal{NP}$-hard equitable graph k-coloring problem (Garey, & Johnson, 1979).

4.3.3 k-colorability and $\mathcal{NP}$-completeness

There are two types k-colorability (Schrijver, 2000).
Definition 4.1: a **k-(vertex) coloring** of a graph $G = (V, E)$ is a function

$$f : V \to \{1, 2, ..., k\}$$

with the property that for every edge $\{u, v\} \in E$ it holds that

$$f(u) \neq f(v).$$

In other words, all vertices that share an edge should be colored differently.

Definition 4.2: the **chromatic number** $\chi(G)$ is the smallest $k \in \mathbb{N}$ for which G has a k-coloring. If G has a k-coloring, then G is called **k-colorable**.

Definition 4.3: a **k-edge coloring** of a graph $G = (V, E)$ is a function

$$f : E \to \{1, 2, ..., k\}$$

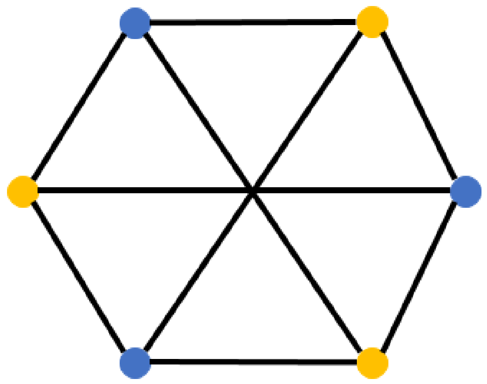
such that for every two edges e, e' with $e \neq e'$ and $e \cap e' \neq \emptyset$ (i.e., for distinct edges sharing a vertex) it holds that

$$f(e) \neq f(e').$$

In other words, all edges that share a vertex should be colored differently.

Definition 4.4: the **edge chromatic number** $\chi'(G)$ is the smallest $k \in \mathbb{N}$ for which G has a k-edge coloring. If G has a k-edge coloring, then G is called **k-edge colorable**.

To illustrate this, figure 4.3 shows a vertex coloring and an edge coloring of $G = K_{3,3}$ (the complete bipartite graph with $V = V_1 \cup V_2$, where $|V_1| = 3$ and $|V_2| = 3$).


(a) 2-vertex coloring of $K_{3,3}$.

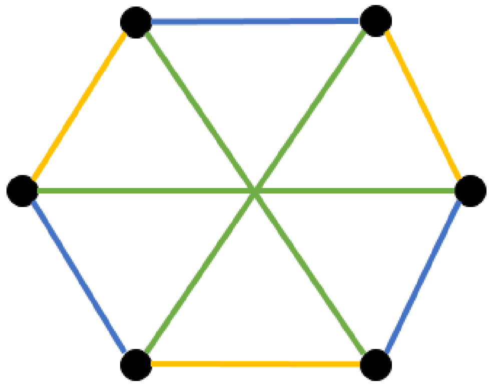

(b) 3-edge coloring of $K_{3,3}$.

Figure 4.3: Colorings of $K_{3,3}$.

It can actually be proven that $\chi(K_{3,3}) = 2$ and $\chi'(K_{3,3,}) = 3$, but this proof is left to the reader. The k-coloring problem as mentioned in the section above refers to finding a k-coloring of a graph. Finding a k-coloring for both the vertices and the edges of a graph is a $\mathcal{NP}$-complete problem (Schrijver, 2000; Holyer, 1981).

To understand what a $\mathcal{NP}$-complete problem is, it is important to understand the classes $\mathcal{P}$ and $\mathcal{NP}$ first (Cormen, Leiserson, Rivest, & Stein, 2009). The problems that are contained in the class $\mathcal{P}$ are solvable in polynomial time. This means that they can be solved in time $O(n^k)$ for some constant k, where n is the size of the input of the problem. These problems are generally regarded as tractable for three reasons.

1. Although we may argue that the problem is actually intractable if k is a very large number, this occurs in very few practical problems. In general, when the first polynomial-time algorithm for a problem is discovered, more efficient algorithms will soon follow.
2. For many reasonable problems of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another.
3. The class of problems in $\mathcal{P}$ has nice closure properties, since polynomials are closed under addition, multiplication and composition.

Most of the problems of the class $\mathcal{NP}$, on the other hand, are generally regarded as much harder. $\mathcal{NP}$ stands for non-deterministic polynomial. This means that the solutions in this class can be verified by a polynomial-time algorithm. However, most problems in this class cannot be solved in polynomial time. For example, a k-vertex coloring of a graph cannot be found with a polynomial-time algorithm, but there is a polynomial-time algorithm that can check if a given k-vertex coloring of a graph is correct. Although it is known that $\mathcal{P} \subset \mathcal{NP}$, it is unknown whether $\mathcal{P} = \mathcal{NP}$. However, most researchers believe this is not the case. A lot of them believe this due to a class that is called $\mathcal{NP}$-complete ($\mathcal{NPC}$) problems. The difference of this class with the class $\mathcal{NP}$ is the property that if any $\mathcal{NP}$-complete problem can be solved in polynomial time, then every other $\mathcal{NP}$-complete problem can be solved in polynomial time as well. In other words, if any $\mathcal{NP}$-complete problem has a polynomial-time solution, then $\mathcal{NPC} \subset \mathcal{P}$. Now, it might seem easy to find a polynomial-time solution of just one $\mathcal{NP}$-complete problem, but despite years of study, no such solution has ever been found. Figure 4.4 shows how researchers generally believe the classes $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{NP}$-complete to be related. Problems that are even harder than $\mathcal{NP}$-complete problems are generally referred to as $\mathcal{NP}$-hard problems. As mentioned before, the weighted k-coloring problem which deals with coloring weighted graphs, is an $\mathcal{NP}$-hard problem.

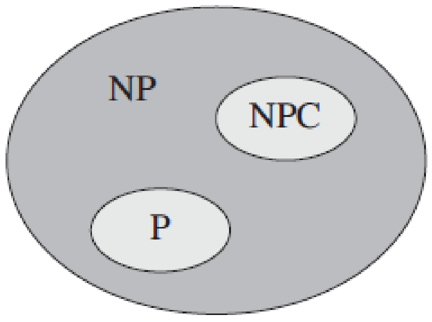


Figure 4.4: How the relation between the classes $\mathcal{P}$, $\mathcal{NP}$ and $\mathcal{NP}$-complete is generally viewed.

4.3.4 Similarities to the Socialization Problem

It is clear that finding seating arrangements is quite a complex problem. The question arises how this problem is related to the Socialization Problem. First of all, the Seating Arrangements Problem provides insights about the construction of one 'round' (the wedding) with distances between guests that indicate their preferences regarding other guests. The distances in the Socialization Problem are the closeness of participants and these change every round. Hence, in the same way as the Seating Arrangements Problem, a new round could be constructed that has maximal socialization based on a previous round. This would mean that the preferences of the Seating Arrangements Problem are actually the distances between participants from the previous round. After a new round is constructed, this becomes the new previous round and another round after that can be constructed until the number of rounds constructed is as desired. The complication was of course that this was an $\mathcal{NP}$-complete problem. Although Lewis and Carroll provide a tool to make a wedding seat planning based on preferences of guests, they admit that it is not optimal and more focused on providing users with a feasible planning in a short amount of time.
Secondly, the Socialization Problem is a bit more difficult. Recall that Lewis and Carroll assumed that two guests had the same distance to each other, so the distance from guest A to guest B is the same as distance from guest B to guest A. However, this is not the case in the Socialization Problem. Participant i can know participant j via participant k, so i has distance 2 with respect to j, but, although j may know k, j may have never even heard of i. In other words, the Seating Arrangements Problem deals with undirected graphs, where every vertex represents a guest and every weighted edge a connection between two guests. In contrast, the Socialization Problem deals with directed graphs, where every vertex represents a participant, every weighted arc from vertex u to v represents the closeness of u with respect to v and every weighted arc from vertex v to u represents the closeness of v with respect to u. This aspect makes the problem even more complicated.
We conclude that the problem is actually already an $\mathcal{NP}$-complete problem when the input of the program is a list of names (each name is then linked to a vertex) of all the participants, a table size and a desired number of rounds. If the input only contains the number of participants, the table size and a desired number of rounds, it becomes even more complicated and thus an $\mathcal{NP}$-hard problem.
This is why we choose to use a method called brute-forcing in favor of any heuristic algorithm. To this end, we use the rounds generated by the program of Siervo.

4.4 The Socialization Problem Program

4.4.1 Brute-forcing

The brute-force method is often used for optimization programs that require a solution of minimal cost among all valid solutions of the problem. The cost is actually the score $S(\hat{r})$ for the Socialization problem. The brute-force method is also often used in $\mathcal{NP}$-complete problems. In this case, the algorithm simply considers every possible sequence of rounds and selects the order of the rounds that is optimal. This works well, but it can be quite time consuming.
Let r_{max} be the number of rounds generated by Siervo's program and $\hat{r}$ be the number of desired rounds. We have to choose $\hat{r}$ rounds from r_{max} rounds, where the order in which the rounds are chosen matters. To check which combination of rounds is optimal, we have to check $\frac{r_{max}!}{(r_{max}-\hat{r})!}$ options. Hence, we have to assume that this number is not too large, to ensure the program gives the desired output fast enough.
The program works by selecting the first round, then the second and so on until the number of rounds desired is reached. Then it considers all possible rounds for the last round that are not any of the rounds chosen before. When it is finished doing this, the program continues to the first to

last round and chooses a new round for this. Then all possible rounds are checked for last round again and so on. To illustrate this, figure 4.5 shows how the program does this when we want to choose three rounds out of five.

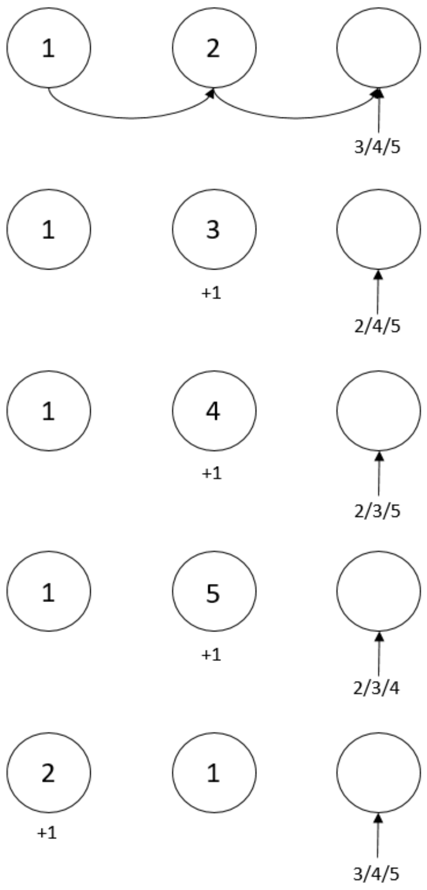


Figure 4.5: The first few orders of rounds chosen by the program when it selects three rounds out of five.

After each combination, the program determines the score of this combination of rounds and compares it to the previous minimal score. If it is less, it replaces the previous minimal score with the score for this combination and the previous combination with this new combination.
The program will now be explained in more detail and some examples will be given as well.

4.4.2 Functions of the Program

The program makes use of a number of different functions. Below is an overview and short explanation presented to better understand how the program works. The input of the program are the rounds generated by Siervo's program and the output is a display of the optimal order of rounds to ensure maximal socialization and the minimal score that belongs to this order. It also makes use of a distance matrix that keeps track of the distance between participant i and participant j for all participants.

- *makeMatrix*: this function makes the initial matrix with the distance $A(0,i,j)$ between participant i and participant j for all participants, where

$$A(0,i,j) = \begin{cases} 0 & \text{if } i = j \\ M & \text{if } i \neq j, \end{cases}$$

  and M is the large number that ensures two people not knowing each other in any way is not beneficial to finding the minimal score solution.

- *createAdjacencyMatrix*: this function creates a binary matrix for all rounds that has a zero entry if two participants do not sit at the same table in that round and a one entry if two participants do sit at the same table.

- *computeNewDistanceMatrix*: this function computes the next iteration of $A(r,i,j)$ as seen in equation 4.1 using the adjacency matrix created in the function *createAdjacencyMatrix*.

- *computeScore*: this function computes the score $S(\hat{r})$ of the distance matrix. We want to minimize the score and give back the optimal rounds that were used to create this distance matrix. If there are dummy variables present in the matrix, then the distances of these dummy variables to the real participants do not count for the score.

- *checkRounds*: this is the main function of the program. It uses recursion to check the score for all combinations of rounds and returns the order of the rounds that have the minimal score and the minimal score itself. If there are multiple orders of rounds that produce the same score, the program picks the first order of rounds with this score it encounters.

The complete code for the RDP program can be found in Siervo's paper. The complete code for the Socialization Program can be found in appendix A.

4.4.3 Examples

Before we give some examples, it is important to note that Siervo used a method to find the optimal rounds for a given number of people and table size and a method that rearranged these rounds such that hosts were distributed uniformly. However, the second method can yield a different order of the rounds each time it is applied. This would mean that the optimal order of the rounds is different for some instances. To prevent this, we use the first method in the program. This method gives the same order of the rounds for every instance and this implies that we will always get the same optimal order of the rounds.

We will first look at Kirkman's schoolgirls problem again. The examples given in chapter 3 are generated with the method that distributes the hosts uniformly, so in table 4.3 we will provide the rounds without the distribution of hosts taken into account.

Round	**Table 1**	**Table 2**	**Table 3**	**Table 4**	**Table 5**
1	1, 2, 3	4, 5, 6	7, 8, 9	10, 11, 12	13, 14, 15
2	1, 4, 7	2, 5, 10	3, 6, 13	8, 11, 14	9, 12, 15
3	1, 5, 14	2, 4, 15	3, 8, 12	6, 9, 11	7, 10, 13
4	1, 9, 13	2, 7, 12	3, 4, 11	5, 8, 15	6, 10, 14
5	1, 8, 10	2, 11, 13	3, 5, 9	4, 12, 14	6, 7, 15
6	1, 6, 12	2, 9, 14	3, 10, 15	4, 8, 13	5, 7, 11
7	1, 11, 15	2, 6, 8	3, 7, 14	4, 9, 10	5, 12, 13

Table 4.3: RDP solution with 15 participants and tables of size 3.

It is immediately clear that the minimal score we can obtain is $14 \cdot 15 = 210$, since all participants know each other directly then, i.e. have distance 1 to each other, and every participant has distance 0 to themselves. This result will be obtained when we choose all seven rounds. The maximal score we can obtain is $14 \cdot M$, where M is a large number. In this example we have chosen $M = 50$. This maximal score will never occur in practice, since it is obtained from choosing zero rounds. It is also clear that, when one round is chosen, it does not matter which round this will be. It will always yield the same score, since every participant knows only two other participants after each round. Perhaps a little harder to see is that choosing two rounds will always yield the same score as well. We can see this as follows: after one round every participant knows two other participants, so two participants have information about them. In the second round, these two participants are both at different tables and they tell the two participants at these tables about the original participant. Thus, every participant knows four participants directly (two from the first round and two from the second) and knows of four other participants indirectly (heard of in the second round). After adding a third round, the scores begin to differ and the result becomes interesting.

When we use the rounds from table 4.3 as input for the Socialization Problem program and request that the program chooses three rounds that ensure optimal socialization we obtain:

The optimal order of the rounds is $[1, 4, 7]$ with score 338.

Figure 4.6 shows the histogram of all scores for all combinations of rounds to gain some insight in how the scores are distributed.

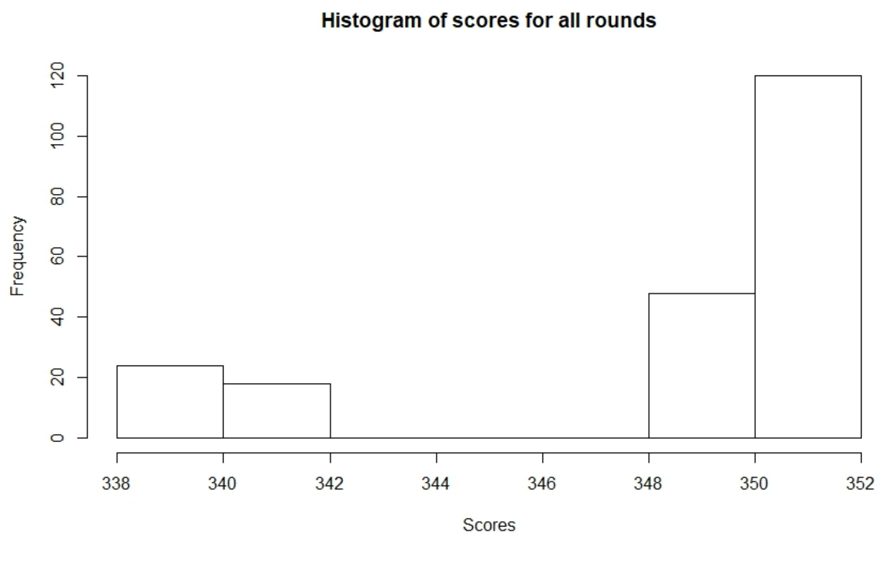
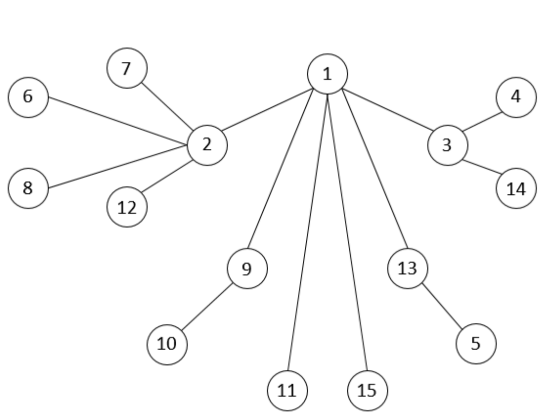
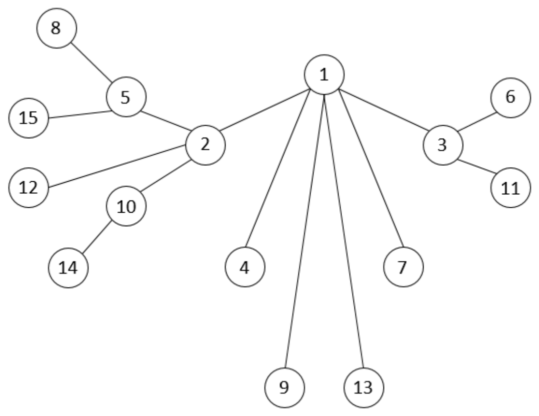


Figure 4.6: Histogram of scores for all combinations of three rounds in an RDP instance with 15 participants and table size 3.

From the summary of the data we get that the minimal score is 338 and the maximal score is 351. By selecting two orders of rounds that correspond to these scores, we will dive deeper into the cause of this difference. We take [1,4,7], the first order of rounds that has score 338, and we take [1,2,4], the first order of rounds that has score 351. For simplicity, we consider one participant and examine how the information of this participant is spread throughout the chosen rounds. We choose participant 1. Figure 4.7 shows the trees corresponding to the information flow of participant 1 for both orders of rounds selected.



(a) Order of rounds [1,4,7].



(b) Order of rounds [1,2,4].

Figure 4.7: The trees that correspond to the information spread of participant 1.

Figure 4.7 provides us with some insights about both order of rounds. In figure 4.7a we can see that the information is more evenly spread out and at the end of the three rounds, every other participant either knows participant 1 personally or knows of participant 1 only via one other participant. In figure 4.7b, on the other hand, the information is not evenly spread out and at the end of the three rounds, there are participants that have distance 3 regarding participant 1. These trees are just examples, but this is likely the case for some other participants as well in both instances and this explains the score difference between the two orders.
When we add another round, so we choose four rounds instead of three, the output of the program is as follows:

The optimal order of the rounds is $[1, 2, 3, 4]$ with score 300.

As expected, one round extra makes for better socialization and so the scores are closer together as can be seen in figure 4.8.

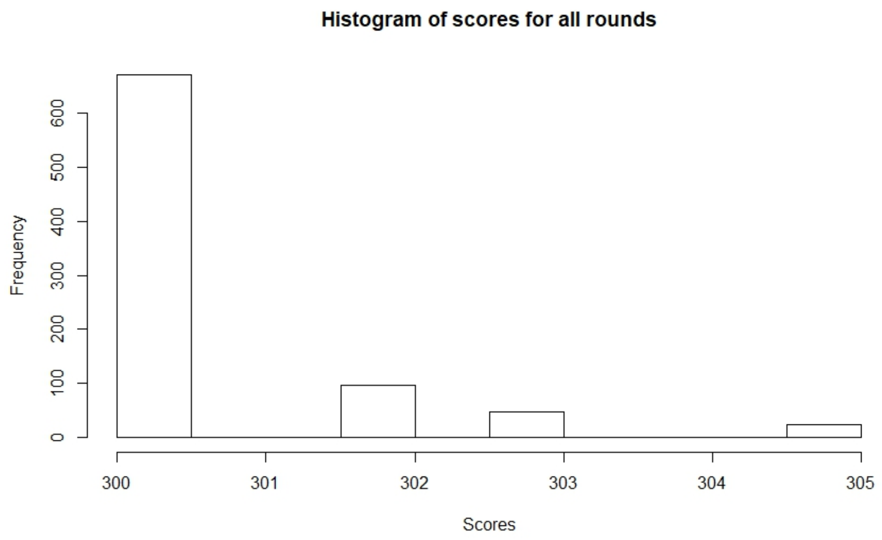


Figure 4.8: Histogram of scores for all combinations of four rounds in an RDP instance with 15 participants and table size 3.

We also see that the number of orders that have an optimal score becomes larger. Furthermore, it is interesting to mention that the three rounds from the optimal order of rounds of the previous example (1, 4 and 7) combined with any other round (2, 3, 5 or 6) also yields an order of rounds with minimal score.
When we choose five rounds we get the result:

The optimal order of the rounds is $[1, 2, 3, 4, 5]$ with score 270.

However, upon further inspection, we see that all orders have the same score. This is shown in figure 4.9.

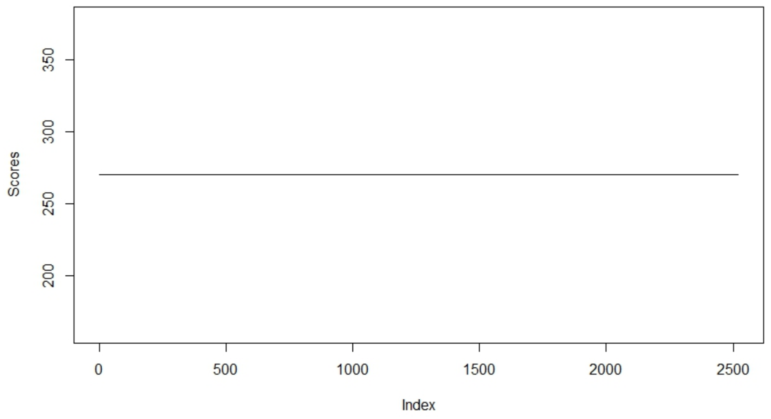


Figure 4.9: Plot of scores for all combinations of five rounds in an RDP instance with 15 participants and table size 3.

The index in figure 4.9 refers to the order in which the combination of rounds are chosen by the program. Thus, index one stands for $[1, 2, 3, 4, 5]$. All scores being equal means that apparently it does not matter anymore which rounds are chosen if the desired number of rounds is five or more. Considering it also does not matter which rounds are chosen if the desired number of rounds is two or less, there is only an optimal order of rounds if three or four rounds are chosen.

The second example is a much smaller example. We have 5 participants and table size 2, meaning we have one dummy variable. We choose three rounds out of five. We will use the example to dive deeper into the characteristics of a 'good' round, meaning a round that is a good fit given the previous rounds. Here, this means a good third round given the first two rounds. Table 4.5 shows the rounds generated by Siervo's program.

Round	**Table 1**	**Table 2**	**Table 3**
1	1, –	2, 5	3, 4
2	2, –	1, 3	4, 5
3	3, –	2, 4	1, 5
4	4, –	3, 5	1, 2
5	5, –	1, 4	2, 3

Table 4.4: RDP solution with 5 participants and tables of size 2.

Recall that the '$-$' notation refers to an empty seat. The minimal score is $4 \cdot 5 = 20$, when all rounds are chosen, and the maximal score is $4 \cdot 50 = 200$, when no rounds are chosen. Note that we have chosen $M = 50$ again. Furthermore, choosing one or two rounds yields the same score for all orders of rounds, as we have seen before. When we let the Socialization Problem program pick three rounds, we obtain:

The optimal order of the rounds is $[1, 2, 4]$ with score 77.

Upon further inspection, it becomes apparent that the score is either 77 or 220 for all orders of three rounds. This is shown in figure 4.10.

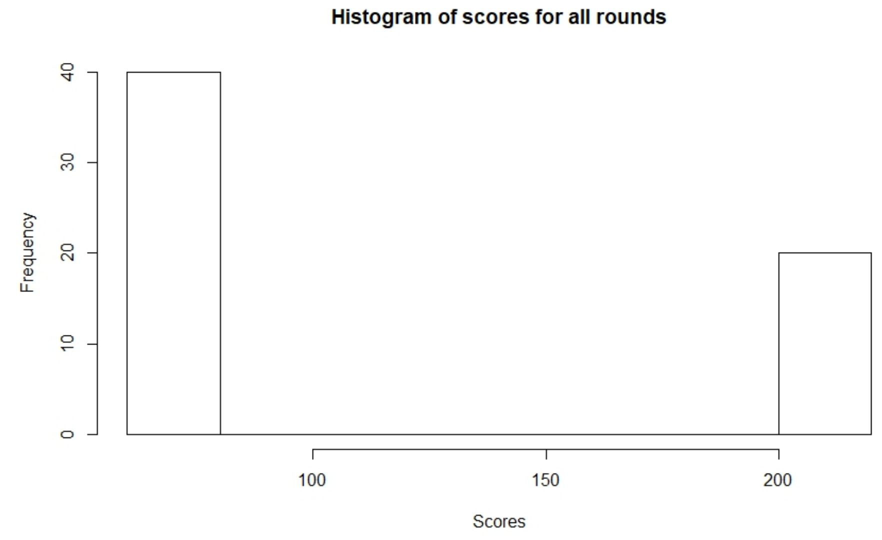


Figure 4.10: Histogram of scores for all combinations of three rounds in an RDP instance with 5 participants and table size 2.

We investigate the order [1,2,3], with score 220, and order [1,2,4], with score 77, to see what makes round 4 a better fit following round 1 and 2. In table 4.5 we can see the distances between all participants after the three rounds. The table consists of the values $A(r, i, j)$ for $i, j = 1, ...5,$.

$i \downarrow, j \rightarrow$	**1**	**2**	**3**	**4**	**5**
1	0	2	1	2	1
2	50	0	2	1	1
3	1	50	0	1	50
4	50	1	1	0	1
5	1	1	2	1	0

$i \downarrow, j \rightarrow$	**1**	**2**	**3**	**4**	**5**
1	0	1	1	2	2
2	1	0	2	3	1
3	1	2	0	1	1
4	50	2	1	0	1
5	2	1	1	1	0

Table 4.5: Distance tables for all participants after rounds [1,2,3] (l) and [1,2,4] (r).

In table 4.5 we see a clear difference between the number of people that do not each other after rounds [1,2,3] and after rounds [1,2,4]. This difference is caused by the dummy variable. In the same situation with 6 participants, the score would have been the same for all orders of rounds as can be seen in figure 4.11.

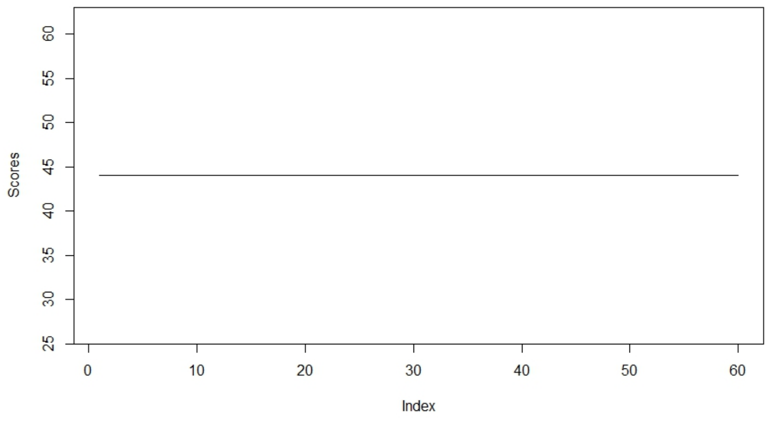


Figure 4.11: Plot of scores for all combinations of three rounds in an RDP instance with 6 participants and table size 2.

Table 4.6 shows the distances between de participants after round 1 and round 2.

$i \downarrow$, $j \rightarrow$	**1**	**2**	**3**	**4**	**5**
1	0	50	1	2	50
2	50	0	50	50	1
3	1	50	0	1	50
4	50	2	1	0	1
5	50	1	2	1	0

Table 4.6: Distance matrix after round 1 and 2.

Table 4.6 shows that, after two rounds, participant 4 already has a lot of information about the other participants, while participant 3 has less information. This explains why it is better to choose round 4, where participant 4 sits alone instead of round 3, where participant 3 sits alone. Furthermore, participant 2 has very little information after round 1 and 2. Round 4 yields more information for participant 2 than round 3. In particular about participant 1.
In this example, the dummy variable influences the outcome and the difference between a 'good' and a 'bad' round. The outcome is also influenced by how evenly the information is distributed. The distribution of information is often the difference between a 'good' and a 'bad' round as we saw before in figure 4.7. A 'good' round is one where the information is more evenly distributed.

In the third example, we explore how the solution changes if the initial matrix changes. There may be running dinners in real life where some of the participants already know each other beforehand or have heard of each other from family or friends. In this case, the organizer may want to include this information in the input of the program. We illustrate this using the previous example. We say participant 1 and participant 3 already know each other, so $A(0,1,3) = A(0,3,1) = 1$. In addition, participant 1 and participant 4 have a common friend and we assume this friend only told participant 4 about participant 1 , so $A(0,4,1) = 2$ and $A(0,1,4) = M$. Table 4.7 shows the initial matrix with $M = 50$.

$i\downarrow,\ j\rightarrow$	**1**	**2**	**3**	**4**	**5**
1	0	50	1	50	50
2	50	0	50	50	50
3	1	50	0	50	50
4	2	50	50	0	50
5	50	50	50	50	0

Table 4.7: Initial matrix with $M = 50$.

We can already predict that it is not beneficial to choose round 2 anymore, since participant 1 and 3 sit at a table together in this round. This means that these participants do not gain any new information.
When we choose three rounds, we get:

The optimal order of the rounds is $[1, 3, 4]$ with score 26.

Indeed, we see that round 2 is skipped in favor of round 3 with this new initial matrix. Furthermore, round 5 is not chosen as well. This was already the case in the previous example, but now it may also have something to do with the fact that participant 1 and 4 sit together in this round and participant 4 already has information about participant 1.

The last example is to illustrate that the program can also handle larger numbers of participants. We have chosen $M = 50$ again. Let the number of participants be 100 and the table size 5. This results in a maximum of nine rounds. Suppose we want to choose five of these. We have to check $\frac{9!}{(9-5)!} = 15120$ options. The program takes 42 seconds to obtain:

The optimal order of the rounds is $[1, 5, 7, 8, 9]$ with score 350688.

For choosing five rounds out of sixteen with 200 participants and table size 5, we have to check $\frac{16!}{(16-5)!} = 524160$ options. The program takes 1 hour and 21 minutes to obtain:

The optimal order of the rounds is $[1, 2, 3, 9, 13]$ with score 87416.

Thus, the program can also handle larger cases in limited time.

Chapter 5

Summary

In chapter 3, we went over the basic constructs from combinatorial design that are used to find a solution to the Running Dinner Problem. We saw that these constructs ensured a perfect solution for two instances: when we have an even number of participants and table size 2 and when the number of tables is p^e and the table size is p with p a prime number and e a positive integer. The program also used some pre-built cases for perfect solutions. In particular, when the table size was 3,4 or 6. We also saw four strategies for cases where these instances do not apply. These strategies do not result in a perfect solution, but they ensure optimal socialization for these cases. We also briefly went over how the RDP program distributes the task of being host. In the end, a suitable seating arrangement was found for all instances with n participants and table size t.

In chapter 4, the Socialization Problem was defined. In this problem, we had to choose or construct a predetermined number of rounds that would lead to optimal socialization. It became clear that this was an $\mathcal{NP}$-complete or even $\mathcal{NP}$-hard problem, depending on the input of the program. Hence, a brute-forcing approach was used to solve this problem. The program takes the maximal number of rounds from the RDP and it checks every combination of rounds to see which combination is optimal. We looked at some examples to see the underlying logic of the choice of rounds. In the end, instances of the Socialization Problem where $\frac{r_{max}!}{(r_{max}-\hat{r})!}$ is not too large can be solved by this program in a limited amount of time. This is usually the case for situations that we could encounter in real life. We will shortly analyze the other cases in the discussion.

In conclusion, the organizer of a running dinner can use the program to make an optimal seating arrangement for his particular number of participants and table size. He can also choose the number of rounds that he wants. The program can also be used for some other purposes mentioned throughout this report, like the social golfer problem and spreading gossip, although it may need some adaptations for the last one.

Chapter 6

Discussion and Future Work

Although the program provides a solution to almost all cases of the Socialization Problem, it is not perfect. If $\frac{r_{max}!}{(r_{max}-\hat{r})!}$ becomes very large, it will take too much time for the program to generate a solution. Furthermore, even for smaller instances, the program sometimes takes a long time to find a solution. We saw this in the last example where we had to wait 1 hour and 21 minutes for a solution. An organizer of a running dinner may not want to wait that long for a reasonable planning of his running dinner. It could be interesting to look into ways to make the running time of the program shorter, therefore providing a more user-friendly waiting time.

Moreover, the program is now made in Java. This is an environment that most people are not familiar with. It may be desirable to transfer the program to an environment that is better understandable for people that do not have a background in programming. This becomes most apparent in the input of a different initial matrix, i.e. when the organizer has to indicate whether participants already know each other before coming to the running dinner. A good way to do this may be the wedding seat planner as provided by Lewis and Carroll (see www.weddingseatplanner.com).

In addition, Lewis and Carroll take different group sizes into account in their wedding seat planner. Of course, different group sizes may also occur in running dinners, e.g. if couples want to be seated together throughout the evening. This makes the problem more difficult, since we cannot simply regard the couple as one person and thus, we have to find a new mechanism to make the arrangements. Perhaps less plausible, but still possible, are different table sizes. Maybe some people can host eight people in their houses, while others can only host three. These concepts may be interesting for future research, though Lewis and Carroll admit adding these features makes the RDP $\mathcal{NP}$-hard.

Lastly, we have not focused on the role of host in this report. Worse still, this role is completely omitted in the Socialization Problem program, since the Socialization Problem only examines the relations between participants. For future work, a mechanism could be added to the program that takes into account how many times a participant has already been host and makes rounds that feature this participant as host again less desirable to choose. This is a difficult task, because the output of Siervo's program differs each time due to the algorithm she uses to distribute the task of being host fairly. This influences the output of the Socialization Problem program as well, such that it differs each time, and this may be confusing for organizers. More research about algorithms for the choice of host could be conducted to gain more insights.

Appendix A

Socialization Problem Program (Java)

```
package nl.tue.rdinner.calculator;

public class Result { // Puts the order and score together

    public final int[] order;
    public final int score;

    public Result(int[] order, int score){
        this.order = order.clone();
        this.score = score;
    }

    public boolean isBetterThan(Result x){
        return this.score < x.score;
    }
}


package nl.tue.rdinner.calculator;

import nl.tue.rdinner.model.RDinnerModel;

import java.util.*;
public class Calculator {
    static Scanner sc = new Scanner(System.in);
    private static int nrGuests;
    private static int tableSize;
    private static int nrRounds;
    private static boolean alreadyKnows;

    public static void main(String[] args){
        Calculator calc = new Calculator();
        RDinnerModel r = new RDinnerModel(); // Calls Siervo's program

        // Let's the organizer choose the values for their particular instance:
```

```
        System.out.println("How many people are participating?");
        nrGuests = sc.nextInt();
        System.out.println("What is the table size?");
        tableSize = sc.nextInt();
        System.out.println("How many rounds would you like?");
        nrRounds = sc.nextInt();
        System.out.println("Do any participants know each other? Type true for
        yes and false for no");
        alreadyKnows = sc.nextBoolean();


        int[][][] rounds = r.findArrangements(tableSize, nrGuests);
        // Constructs rounds based on Siervo's program


        int[][][] arrangements = r.arrangeHeaders(rounds, nrGuests);
        // Rearranges the rounds such that the distribution of the hosts is uniform

        // Prints the rounds generated by Siervo's program:
        for (int k = 0; k < rounds.length; k++){
            System.out.print("============ round ");
            System.out.print(k+1);
            System.out.print(" ============");
            System.out.println("");
            for (int i = 0; i < rounds[k].length; i++){
                System.out.print("Table ");
                System.out.print(i+1);
                System.out.print(": ");
                for (int j = 0; j < rounds[k][i].length; j++){
                    System.out.print(rounds[k][i][j]+1);
                    System.out.print(", ");
                }
                System.out.println("");
            }
        }
        int[] order = new int[nrRounds];
        if (alreadyKnows == false){ // Case 1: no one knows each other at
        the beginning of the running dinner
            Result bestResult = checkRounds(rounds, createAdjacencyMatrix(rounds),
                makeMatrix(rounds, 50), rounds.length, nrGuests, order, 0);
        int[] optimalOrder = bestResult.order;
        int optimalScore = bestResult.score;
        System.out.println("");
        System.out.println("The optimal order of the rounds is "
        +Arrays.toString(optimalOrder)+ " with score " +optimalScore);
        } else { // Case 2: some participants already know each other
            int nrPeople = arrangements[0].length * arrangements[0][0].length;
            // Accounts for dummies
            int [][] matrix = new int [nrPeople][nrPeople];
            // Let's the organizer decide the initial distance between participants:
            System.out.println("The value between two participants is 1 if
            they know each other, "
                        + "2 if they know the same person, etc. Typ 50 if they do
                    not know each other.");
            for(int i = 0; i < matrix.length; i++){
                for(int j = 0; j < matrix[i].length; j++){
                    if ((i >= nrGuests || j >= nrGuests) && i != j){
                        matrix[i][j] = 50;
                    } else if(i == j){
                        matrix[i][j] = 0;
                    } else {
                    System.out.print("Type␣the␣value␣between␣participant␣");
                    System.out.print(i+1);
                    System.out.print("␣and␣");
                    System.out.print(j+1);
                    System.out.println("");
                    matrix[i][j] = sc.nextInt();
                    }
                }
            }
            Result bestResult = checkRounds(rounds, createAdjacencyMatrix(rounds),
                matrix, rounds.length, nrGuests, order, 0);
        System.out.println("The␣given␣distance␣matrix␣is:");
        printMatrix(matrix);
        int[] optimalOrder = bestResult.order;
        int optimalScore = bestResult.score;
        System.out.println("");
        System.out.println("The␣optimal␣order␣of␣the␣rounds␣is␣"
        +Arrays.toString(optimalOrder)+ "␣with␣score␣" +optimalScore);
        }
    }

    static Result checkRounds(int[][][] arrangements, int[][][] adjacencyMatrix,
            int[][] matrix, int nrRounds, int nrPeople, int[] order, int orderIndex){
            // Uses recursion to check all rounds
        int max = nrPeople;
        Result bestResult = new Result(order, Integer.MAX_VALUE);
        if (orderIndex >= order.length){ // Case 1: the desired number of rounds
        is reached, the score is computed and compared to previous scores
            int S = computeScore(matrix, max);
            System.out.println("The␣score␣for␣" +Arrays.toString(order)+ "␣is␣" +S);
            if( S <= bestResult.score){
                bestResult = new Result(order, S);
            }
        } else { //Case 2: a new round is added to the list after checking
        if this round is not already present in the list
          for (int round = 0; round < nrRounds; round++){
              boolean alreadyPresent = false;
              for (int j = 0; j < orderIndex; j++){
                 if (order[j] == round){
                    alreadyPresent = true;
                    break;
                 }
              }
              if(!alreadyPresent){
                 order[orderIndex] = round;
                 // The new distance between participants is computed after
                 the new round is added:
```

```
                int[][] x = computeNewDistanceMatrix(arrangements, matrix,
                adjacencyMatrix[round]);
                // The method calls itself until the list is filled:
                Result newResult = checkRounds(arrangements, adjacencyMatrix, x,
                nrRounds, nrPeople, order, orderIndex + 1);
                // If a new score is better than the previous score,
                the previous order and score are replaced:
                if (newResult.isBetterThan(bestResult)){
                    bestResult = newResult;
                }
            }
          }
        }
        return bestResult;
    }

static int[][] makeMatrix(int[][][] arrangements, int largeNumber){
// Makes the initial matrix
int nrPeople = arrangements[0].length * arrangements[0][0].length;
// Accounts for dummies
int[][] matrix = new int[nrPeople][nrPeople];
for(int i = 0; i < matrix.length; i++){
    for(int j = 0; j < matrix[i].length; j++){
        if(i == j){
            matrix[i][j] = 0;
        }
        else{
            matrix[i][j] = largeNumber;
        }
    }
}
return matrix;
}

static int[][][] createAdjacencyMatrix(int[][][] rounds){
// Creates an adjacency matrix for all rounds
    int nrPeople = rounds[0].length * rounds[0][0].length; // Accounts for dummies
   int[][][] matrix = new int[rounds.length][nrPeople][nrPeople];
   for(int i = 0; i < rounds.length; i++){
       for(int tableindex = 0; tableindex < rounds[i].length; tableindex++){
           for(int indexOne = 0; indexOne < rounds[i][tableindex].length; indexOne++){
               for(int indexTwo = 0; indexTwo < rounds[i][tableindex].length;
               indexTwo++){
                   int personOne = rounds[i][tableindex][indexOne];
                   int personTwo = rounds[i][tableindex][indexTwo];
                   matrix[i][personOne][personTwo] = 1;
               }
           }
       }
   }
   return matrix;
}

   public static void printMatrix(int[][] matrix){ // Prints a matrix
```

```
        System.out.println("==========␣BEGINS␣HERE␣===============");
        for(int i = 0; i < matrix.length; i++){
            for(int j = 0; j < matrix[i].length; j++){
                System.out.print(matrix[i][j] + "␣");
            }
            System.out.println("");
        }
        System.out.println("==========␣ENDS␣HERE␣===============");
    }

  static int[][] computeNewDistanceMatrix(int[][][] arrangements,
  int[][] oldDistanceMatrix, int[][] adjacencyMatrix, int nrGuests){
  // Computes the new distance after a round
        int nrPeople = arrangements[0].length * arrangements[0][0].length;
        // Accounts for dummies
        int[][] newDistanceMatrix = new int[nrPeople][nrPeople];
        for (int i = 0; i < nrGuests;i++){
            for (int j = 0; j < oldDistanceMatrix[i].length; j++){
                newDistanceMatrix[i][j] = oldDistanceMatrix[i][j];
                for (int k = 0; k < nrGuests; k++){
                    if (adjacencyMatrix[i][k] == 1){
                        newDistanceMatrix[i][j] = Math.min(newDistanceMatrix[i][j],
                        oldDistanceMatrix[k][j] + 1); // Iteration of S(r,i,j)
                    }
                }
            }
        }
        for (int i = nrGuests; i < oldDistanceMatrix.length; i++){
            for (int j = 0; j < oldDistanceMatrix[i].length; j++){
                newDistanceMatrix[i][j] = oldDistanceMatrix[i][j];
            }
        }
        return newDistanceMatrix;
    }

    static int computeScore(int[][] distanceMatrix, int max){ // Adds all
    entries of the last distance matrix, ignores the entries of dummy variables
        int score = 0;
        for (int i = 0; i < max; i++){
            for(int j = 0; j < max; j++){
                score = score + distanceMatrix[i][j];
            }
        }
        return score;
    }
}
```

Appendix B

Data Analysis Code (R)

```
#Histogram and summary of scores with 15 participants,
table size 3 and three rounds chosen
Data.RD.15p <- read.csv("~/Jaar 3/Q3/BEP/Data RD 15p.txt", header=FALSE)
summary(Data.RD.15p$V1)
hist(Data.RD.15p$V1, main = "Histogram of scores for all rounds", xlab = "Scores")

#Histogram of scores with 15 participants, table size 3 and four rounds chosen
Data.RD.15p4r <- read.table("~/Jaar 3/Q3/BEP/Data RD 15p4r.txt",
quote="\"", comment.char="")
hist(Data.RD.15p4r$V1,main = "Histogram of scores for all rounds", xlab = "Scores")

#Plot of scores with 15 participants, table size 3 and five rounds chosen
Data.RD.15p5r <- read.table("~/Jaar 3/Q3/BEP/Data RD 15p5r.txt",
quote="\"", comment.char="")
plot(Data.RD.15p5r$V1, ylab = "Scores", type = "l")

#Histogram of scores with 5 participants, table size 2 and three rounds chosen
Data.RD.5p3r <- read.table("~/Jaar 3/Q3/BEP/Data RD 5p3r.txt",
quote="\"", comment.char="")
hist(Data.RD.5p3r$V1,main = "Histogram of scores for all rounds", xlab = "Scores")

#Plot of scores with 6 participants, table size 2 and three rounds chosen
Data.RD.6p3r <- read.table("~/Jaar 3/Q3/BEP/Data RD 6p3r.txt",
quote="\"", comment.char="")
plot(Data.RD.6p3r$V1, ylab = "Scores", type = "l")
```

Bibliography

[1] Graham, R. L., Grötschel, M., & Lovász, L. (1995). *Handbook of combinatorics, Volume 2.* Amsterdam: Elsevier.

[2] Triska, M., & Musliu, N. (2011). *An effective greedy heuristic for the Social Golfer Problem.* Annals of Operations Research, 194(1), 413-425. doi:10.1007/s10479-011-0866-7.

[3] Hurkens, C.A.J. (n.d.). *Vind een tafelschikking voor uw Running Dinner.* http://www.win.tue.nl/ wscor/RunningDinner.html.

[4] Siervo, R. M. (2017). *The Running Dinner Problem* (Rep.).

[5] Herzig, A., & Maffre, F. (2017). *How to share knowledge by gossiping.* AI Communications, 30(1), 1-17. doi:10.3233/aic-170723.

[6] Hurkens, C.A.J. (2000). *Spreading gossip efficiently.* Faculteit Wiskunde en Informatica, Technische Universiteit Eindhoven.

[7] Baker, B., & Shostak, R. (1972). *Gossips and telephones.* Discrete Mathematics, 2(3), 191-193. doi:10.1016/0012-365x(72)90001-5

[8] Tijdeman, R. (1971). *On a telephone problem.* Nieuw Archief voor Wiskunde 19(3), 188–192.

[9] Knödel, W. (1975). *New gossips and telephones.* Discrete Mathematics, 13(1), 95. doi:10.1016/0012-365x(75)90090-4

[10] Lewis, R., & Carroll, F. (2016). *Creating seating plans: A practical application.* Journal of the Operational Research Society, 67(11), 1353-1362. doi:10.1057/jors.2016.34

[11] Garey, M. R., & Johnson, D. S. (2009). *Computers and intractability: A guide to the theory of NP-completeness.* New York: Freeman.

[12] Schrijver, A. (2000). *A course in combinatorial optimization.* Delft: TU Delft.

[13] Holyer, I. (1981). *The NP-Completeness of Edge-Coloring.* SIAM Journal on Computing, 10(4), 718-720. doi:10.1137/0210055

[14] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms (3rd ed.).* Massachusetts: MIT.