Algorithms for Managing Deliberation

A THESIS PRESENTED
BY
Rose M. Hong

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE JOINT DEGREE OF
Bachelor of Arts with Honors
IN THE SUBJECT OF
Computer Science and Mathematics

Harvard University
Cambridge, Massachusetts
May 2022
Algorithms for Managing Deliberation

Abstract

Citizens’ assemblies, in which ordinary people are randomly selected to participate in the policymaking process, have become increasingly widespread in recent decades. Chosen via a method known as sortition, these assemblies have not only had a profound impact on worldwide legislation but also sparked a flurry of recent research into how they can best be organized, from recruiting and selecting participants to managing the deliberation itself. In this report, we focus on the assembly partitioning problem, in which we wish to partition participants among a set of moderated groups over multiple sessions of deliberation such that 1) each group is representative of the assembly, and 2) participants get to interact with as many new people as possible over the course of deliberation. We first provide an overview of the baseline algorithm that is being used to generate partitions. We then define a model for the problem and propose a greedy, linear programming-based approach to tackling it. Afterwards, we conduct a series of experiments to compare our algorithm with the baseline, demonstrating that our algorithm generates significantly higher-quality solutions on both synthetic and real-world assembly data. We also derive a theoretical bound on the maximum number of unique interactions that can be achieved between back-to-back partitions. Finally, we discuss the advantages of our algorithm over the previous approach, advocating that organizers adopt our algorithm for future assemblies.
Contents

1 Introduction .................................................. 1
   1.1 Sortition Overview ........................................ 1
   1.2 Motivations ............................................... 3
   1.3 Related Work ............................................. 7
   1.4 Contributions ........................................... 13

2 Preliminaries .............................................. 15
   2.1 Definitions .............................................. 15
   2.2 Model .................................................. 17

3 Algorithm .................................................. 20
   3.1 ILP Approach ............................................ 20
   3.2 Initial Algorithm ....................................... 21
   3.3 Greedy Algorithm ...................................... 21

4 Empirical Results ......................................... 30
   4.1 Datasets ................................................ 30
   4.2 Basis of Comparison .................................... 32
   4.3 Experiment Naming Conventions ....................... 33
   4.4 Experiments with Clustered Participants .......... 33
   4.5 Experiments without Clustered Participants ....... 45

5 Theoretical Results ........................................ 55
   5.1 Definitions & Assumptions ............................ 56
   5.2 Objective ............................................... 56
   5.3 Theorem (Minimum Repeated Meetings) ............ 57

6 Discussion & Conclusion ................................... 65
   6.1 Discussion ............................................. 65
   6.2 Conclusion .............................................. 69

Appendix A Initial Model & Algorithm ....................... 72
   A.1 Initial Model ........................................... 72
   A.2 Initial Algorithm ...................................... 73

References .................................................. 76
# List of Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. GroupSelect</td>
<td>10</td>
</tr>
<tr>
<td>2. Greedy</td>
<td>25</td>
</tr>
<tr>
<td>3. Corollary 5.3.1.1 – Sample Assignment Strategy</td>
<td>60</td>
</tr>
</tbody>
</table>
Acknowledgments

To my thesis advisor Professor Ariel Procaccia, thank you for welcoming me into your group and for mentoring me throughout this entire process. I am incredibly grateful that you gave me the opportunity to take CS 238 with you last spring, as your teaching and passion for the subject were what made this project possible in the first place. Your tireless devotion to helping me grow as both a researcher and an individual has made accomplishing this big milestone not only possible but also fun. Thank you for all of the time and effort you have invested in me over the past year, as well as all of the wisdom you have imparted on me along the way.

To my mentors Jake Barrett, Paul Götz, and Kobi Gal, thank you for your dedication to this project and for all of your guidance over the past year. You helped bring this project to life and made me genuinely excited to share my updates with you each month. Together, you also served as my support network and helped me make progress any time I was stuck. Thank you for all of your feedback and support.

To Philipp Verpoort at the Sortition Foundation, thank you for all of the time you have given to our team. Your work in sortition inspired us to start this project last May, and your feedback and support since then have been invaluable.

To Professor Yannai Gonczarowski, thank you for serving as my thesis reader. Although we haven’t had the chance to work together in the past, I thoroughly enjoyed learning about your Erdős-Bacon number from your website and am incredibly grateful for your time. I look forward to hearing your feedback.

Finally, to my friends and roommates, thank you for the welcome distractions and constant encouragement. I look forward to celebrating with you once this is submitted. And to my parents, thank you for all of your love and support, and for the sacrifices you continue to make so that I can pursue my dreams 3,000 miles away from home.
1.1 Sortition Overview

In representative democracies like the United States, citizens typically elect government officials who in turn make policy decisions for them. While this system is commonplace and familiar to many of us, however, its usefulness is contingent on whether these decisions truly reflect the will of the people. Many of us are losing faith in the government because the electoral system largely excludes ordinary citizens; our elected officials are at the mercy of political parties, which are themselves at the mercy of the rich and powerful [1]. In recent decades, however, another method of selecting representatives—known as sortition—has become increasingly popular. An age-old process that was notably used in Athenian democracy and now has widespread support among politi-
cal scientists and activists alike [2, 3], the sortition method calls on ordinary citizens to participate in the policy-making process by randomly selecting representatives from the population. These representatives constitute a *sortition panel, or citizens’ assembly*, which then deliberates and votes on particular policy issues. Over the past few years, a growing number of organizations have been established to help facilitate these assemblies [4]. One of these organizations—the Sortition Foundation, a not-for-profit company that provides recruitment and selection services for citizens’ assemblies around the world—administers dozens of assemblies each year alone [4]. Moreover, the impact that these assemblies are having on legislation is evident from recent examples like the Irish Citizens’ Assembly, which led to Ireland’s legalization of abortion in 2018 [5, 6].

The process of selecting a citizens’ assembly typically works as follows: First, thousands of people are chosen uniformly at random from the population and invited to participate. Out of those invited, a subset opt in, forming a *pool of volunteers*. Then, an assembly of the desired size is chosen from this pool via some sort of selection *algorithm*, which ideally aims to produce a panel that 1) is representative of the larger population, and 2) gives everyone an equal probability of being selected [4]. Although much of the literature on these “fair” selection algorithms is relatively new, we provide a brief overview of the progress that has been made in recent years below.

The current state-of-the-art algorithm being used by the Sortition Foundation for panel selection was developed by Flanigan et al. [4]. Their algorithm—LEXIMIN1—aims to reconcile the two ideals cited above by being *maximally fair*; in other words, subject to quotas that constrain the representation of various demographic factors (e.g., race, gender, and so on) on the panel, it achieves selection probabilities that are “as equal as possible.” In their paper, Flanigan et al. demonstrate using several real-world datasets that their algorithm is significantly fairer (as computed by multiple fairness measures) than the legacy algorithm. In addition, they develop a more general framework for producing maximally fair algorithms. At the time of publication, LEXIMIN had already been used to select more than 40 panels.

Prior to Flanigan et al. [4], a few other papers had also studied the problem of panel selection; however, most

1Note that the actual implementation of the algorithm is known as StratifySelect.
of them were conducted using models of sortition that are not as applicable to real-world assemblies. Benadè et al. [7], for example, study stratified sampling—a process often used by organizers to deterministically “fix” the representation of a particular demographic factor in the panel (e.g., for gender, choosing half of the panel members among women volunteers and half among men volunteers, thereby guaranteeing an equal number of both)—and its effects on the representation of other groups. However, their model makes the assumption that all members of the population are willing to participate in the panel, which is not realistic in practice. Flanigan et al. [8], on the other hand, develop a sampling algorithm that actually does take into account how likely members of the population are to participate in the panel. In particular, they use these participation probabilities to design an algorithm that restores near-equal selection probabilities for members of the population. This approach differs most significantly from [4] in that [8] aims to equalize selection probabilities from the population to the panel, while [4] aims to equalize selection probabilities among the pool of volunteers. However, because participation probabilities are not actually observable, they must be learned from survey data using machine learning methods that may result in algorithmic bias against certain groups.

In the rest of this report, we focus on the process that follows panel selection—deliberation.

1.2 Motivations

After a citizens’ assembly has been selected using sortition, participants convene, and the actual deliberation begins. In order to facilitate this process, assembly organizers typically partition participants among a set of moderated groups (usually tables²). In an assembly of 30 people, for example, organizers might split everyone into 5 tables of 6. In practice, these partitions are typically generated for multiple sessions of deliberation that might take place over several days or weeks. This format of deliberation ultimately allows for smaller, more intimate discussions among participants where everyone has the opportunity to not only express their ideas but also gain exposure to new ones. Beyond these smaller group settings, however, it is clear that other factors—such as how

²We use the terms “group” and “table” interchangeably throughout the rest of this report.
many people each participant engages with—also contribute to the quality of deliberation, which, in the context of a deliberative democracy, should aim to promote an exchange of ideas that ultimately helps citizens arrive at political decisions through exposure to competing arguments and viewpoints [9]. Thus, in order to more concretely define what these other factors are, we consider the following example:

1.2.1 Motivating Example

Imagine that we are organizers of a climate assembly where participants will discuss how our country should reach net-zero greenhouse gas emissions by 2050\(^3\). Our assembly consists of 6 people, whose attributes are given as follows:

<table>
<thead>
<tr>
<th>Label</th>
<th>Gender</th>
<th>Geographic Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR(_1)</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WR(_2)</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WR(_3)</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WC</td>
<td>Woman</td>
<td>Coastal City</td>
</tr>
<tr>
<td>MR</td>
<td>Man</td>
<td>Rural</td>
</tr>
<tr>
<td>MC</td>
<td>Man</td>
<td>Coastal City</td>
</tr>
</tbody>
</table>

Table 1.1: Climate Assembly Participants’ Attributes

In other words, our assembly consists of 3 women and 1 man from rural communities (WR\(_1\), WR\(_2\), WR\(_3\) and MR, respectively) and 1 woman and 1 man from coastal cities (WC and MC, respectively).

Representation

First, suppose that our goal is to place everyone on 3 tables over 1 deliberation session. We consider the following partitions:

\(^3\)This scenario is borrowed from an actual climate assembly organized by the UK in 2020 [10].
Table 1.2: Session 1 Partition Options

<table>
<thead>
<tr>
<th>Partition 1</th>
<th>Table 1</th>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR₁, WR₂, WR₃</td>
<td>WC, MR, MC</td>
<td></td>
</tr>
<tr>
<td>Partition 2</td>
<td>WR₁, WR₂, MC</td>
<td>WR₃, WC, MR</td>
</tr>
</tbody>
</table>

In Partition 1, all 3 women from rural communities are grouped on Table 1, while in Partition 2, WR₃ and MC have been swapped so that each table now includes 2 women and 1 man, as well as 2 individuals from rural communities and 1 individual from a coastal city. We can reason intuitively about why this property makes Partition 2 more appealing than Partition 1: In Partition 2, both the woman-to-man and rural-to-city ratios on each table (both 2 : 1) match the respective ratios in the given assembly. In practice, we know that these assemblies are selected to be representative of the larger population (e.g., of the country) so that ideally, each acts as a “microcosm of society,” and deliberation approximately simulates the nation convening [4]. In order to retain these deliberative properties, it is thus reasonable for our deliberation groups to also reflect the diversity of the assembly and larger population by representing each attribute in the appropriate ratio across all tables.

Number of New Meetings (NNM)

Now, suppose that our goal is to place everyone on 3 tables over 2 deliberation sessions. For the first session, we will partition the assembly according to Partition 2 in Table 1.2. For the second session, we consider the following options:

Table 1.3: Session 2 Partition Options

<table>
<thead>
<tr>
<th>Partition 2 (repeat)</th>
<th>Table 1</th>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR₁, WR₂, MC</td>
<td>WR₃, WC, MR</td>
<td></td>
</tr>
<tr>
<td>Partition 3</td>
<td>WR₁, WR₃, MC</td>
<td>WR₂, WC, MR</td>
</tr>
<tr>
<td>Partition 4</td>
<td>WR₂, WR₃, MC</td>
<td>WR₁, WC, MR</td>
</tr>
</tbody>
</table>

Observe that across all partitions in Table 1.3, both the woman-to-man and rural-to-city ratios on each table match the respective ratios in the given assembly. However, these partitions demonstrate that simply having appropriate representation across all tables is not enough. In particular, repeating Partition 2 for Session 2 would
mean that no one meets anyone new, while choosing either Partition 3 or 4 would allow 4 new pairs of people to exchange ideas.

<table>
<thead>
<tr>
<th>Table 1 New Mtgs.</th>
<th>Table 2 New Mtgs.</th>
<th>NNM</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>∅</td>
<td>0</td>
</tr>
<tr>
<td>Partition 3</td>
<td>(WR₁, WR₃), (WR₃, MC)</td>
<td>(WR₂, WC), (WR₂, MR)</td>
</tr>
<tr>
<td>Partition 4</td>
<td>(WR₂, WR₃), (WR₃, MC)</td>
<td>(WR₁, WC), (WR₁, MR)</td>
</tr>
</tbody>
</table>

**Table 1.4: Session 2 New Meetings**

As deliberations revolve around a central policy issue, reorganizing our groups helps participants meet new faces that bring fresh perspectives to the table. One can also imagine that vastly different discussions arise from interactions among new combinations of people (e.g., Partition 2 Table 1 from Session 1 vs. Partition 3 Table 1) such that the Partition 3 and 4 pairings in Table 1.4 ultimately help contribute to a richer discourse over the course of all sessions. Thus, both participants and we as organizers would likely prefer Partition 3 or 4 over Partition 2 for Session 2. Between Partitions 3 and 4, however, it is unlikely that we have a preference because they attain the same number of new meetings among participants. In particular, even though the actual meetings themselves differ, we do not inherently prefer one set of meetings over the other. In this way, the number of new meetings (NNM) metric captured in Table 1.4 gives us a way to quantify how effective a particular partition is at exposing participants to new and competing viewpoints, which we argued in Section 1.2 was a defining feature of high deliberation quality.

**Tradeoff Between Representation & Number of Meetings**

Finally, we observe that there is a tradeoff between the strictness of representation on each table and the number of meetings that can be generated for a particular assembly. In particular, if we remain strict on representation (i.e., trying to match the attribute ratios on each table with the respective ratios in the given assembly as closely as possible), then the two men (MR and MC) will never meet each other, and the two participants from cities (WC and MC) will also never meet each other. However, if we instead relax representation along the Gender axis
(i.e., say we care less about Gender representation than Geographic Classification representation in these climate deliberations), then it becomes possible to generate the additional meeting between $MR$ and $MC$. What we as organizers prioritize in this tradeoff thus affects which meetings can be generated.

### 1.2.2 Motivating Questions

In Subsection 1.2.1, we motivated why assembly organizers should value both the representation of participants across tables and the NNM attained each session when choosing partitions. However, although these partitions were easy to generate and compare for the assembly of 6 participants, it is easy to see that the problem of partitioning scales quickly with the number of participants and attributes. Moreover, other circumstances often complicate the process of partitioning: Certain participants (e.g., ones that do not agree to be photographed), for example, may always need to be grouped together, or clustered. It is thus clear that as citizens’ assemblies become increasingly widespread, some sort of algorithm is needed to help organizers generate partitions that capture these desirable properties. In this rest of this report, we focus on developing such an algorithm. Before we proceed down such a path, however, we first frame our journey with the following questions:

- How do we model the assembly partitioning problem?
- How do we develop an algorithm that not only captures the desired properties but is also efficient, versatile, and easy to use?
- What algorithmic and theoretical guarantees can we provide?

### 1.3 Related Work

Currently, we are aware of one algorithm that is being used by assembly organizers to generate partitions. This open-source software was developed by the Sortition Foundation, and in this section we present both the algorithm and its properties.
1.3.1 GroupSelect

Other than StratifySelect, the open-source software that generates an assembly from a pool of volunteers, the Sor- tition Foundation also offers another algorithm—GroupSelect—that partitions assembly participants among a set of groups over multiple sessions of deliberation. As stated on the foundation’s website, GroupSelect’s goal is to generate diverse groups while maximizing the number of interactions participants have over these sessions. Features of the algorithm include an “unlimited number of diversification fields, prioritization of fields, clustering, and manual allocations” [11]. In the following subsections, we provide an overview of the GroupSelect algorithm and its features.

Inputs

For each assembly, GroupSelect takes as input a participants’ attributes table like the one given in Table 1.1, where each row represents a participant, and each column represents an attribute such as “Label” or “Geographic Classification.” We reproduce the table below for convenience.

<table>
<thead>
<tr>
<th>Label</th>
<th>Gender</th>
<th>Geographic Classification</th>
</tr>
</thead>
<tbody>
<tr>
<td>WR1</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WR2</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WR3</td>
<td>Woman</td>
<td>Rural</td>
</tr>
<tr>
<td>WC</td>
<td>Woman</td>
<td>Coastal City</td>
</tr>
<tr>
<td>MR</td>
<td>Man</td>
<td>Rural</td>
</tr>
<tr>
<td>MC</td>
<td>Man</td>
<td>Coastal City</td>
</tr>
</tbody>
</table>

A mode can then be chosen for each attribute, where the modes are defined as follows:

- **Ignore**: The algorithm ignores this attribute.

- **Print Label**: When outputting partitions, the algorithm uses this attribute to refer to each participant; in other words, the participant’s attribute value acts as their Python `str()` identifier. If multiple attributes are put in Print Label mode, then the algorithm concatenates each participant’s attribute values together. For example, if both Label and Geographic Classification rather than just Label are put in this mode, then participant WR1 would instead be labeled WR1 Rural. On the other hand, if no attributes are put in Print Label mode, then participants are 0-indexed by row.
• **Cluster:** For this attribute, the algorithm roughly tries to group participants with the same attribute values together. For example, if Gender is put in Cluster mode (i.e., Gender is clustered), then the algorithm tries to group all women together on the same tables, and all men together on the same tables. Furthermore, as will become important later, GroupSelect clusters a binary attribute by distributing all of the clustered participants among as few groups as possible.

• **Diversify:** For this attribute, the algorithm roughly tries to distribute its values evenly across all tables; this is akin to what we did in Partition 2 of Subsection 1.2.1 (Representation). The algorithm requires that at least one attribute be diversified.

Across all clustered and diversified attributes, GroupSelect requires an attribute ordering to be specified. This feature roughly allows organizers to rank the attributes in Cluster and Diversify mode, respectively, by importance so that higher-priority attributes are more likely to reflect the desired properties of the chosen mode. The algorithm also allows for manual allocation so that organizers have the option to assign participants to groups themselves.

Finally, GroupSelect takes as input the number of groups desired; the number of seats available per group (which must be constant over all groups); the number of partitions, or allocations\(^4\), desired; and a random seed.

**Output**

For each allocation, GroupSelect outputs the participants that have been assigned to each group. The output for 2 allocations, for example, might look something like Table 1.2, which we reproduce below.

<table>
<thead>
<tr>
<th></th>
<th>Table 1</th>
<th>Table 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition 1</td>
<td>WR₁, WR₂, WR₃</td>
<td>WC, MR, MC</td>
</tr>
<tr>
<td>Partition 2</td>
<td>WR₁, WR₂, MC</td>
<td>WR₃, WC, MR</td>
</tr>
</tbody>
</table>

**Algorithm**

We now present the GroupSelect algorithm (p. 10).

\(^4\) We use the terms “partition” and “allocation” interchangeably throughout the rest of this report.
Algorithm 1 GroupSelect

1: function GroupSelect(participants, num_allocs)
2:     Handle manual allocation ▷ See details.
3:
4:     allocs ← []
5:     for i in range(100) do
6:         Randomly shuffle participants
7:         alloc ← GetAllocation(participants)
8:         Add alloc to allocs
9:     end for
10:
11:    samples ← []
12:    for i in range(100) do
13:        sample ← (random sample of num_allocs allocations from allocs)
14:        Add sample to samples
15:    end for
16:
17:    return (sample in samples that generates the most meetings)
18: end function
19:

20: function GetAllocation(participants)
21:     Order participants acc. to user-specified attribute ordering ▷ See details.
22:     alloc ← {}
23:     for p in participants do
24:         alloc[p] ← (group where p’s attribute values are most underrepresented) ▷ See details.
25:     end for
26:    return alloc
27: end function
Below are a few additional details, labeled by line number:

(2) GroupSelect gives organizers the option to assign participants to groups themselves. It handles manual allocation by first validating that the total number of manually-allocated participants assigned to each group does not exceed the number of seats available in that group, and then assigning the participants to their designated groups across all allocations.

(21) For convenience, suppose that each participant is associated with an attribute vector that captures their attribute values across all attributes. In Table 1.1, for example, participant $WR_1$’s attribute vector can be represented as

$$[WR_1, \text{Woman, Rural}].$$

From the user-specified attribute ordering, GroupSelect recursively generates a deterministic ordering of all possible attribute vectors. The subset of participants with each attribute vector is then slotted into its appropriate place in the ordering, with the ordering of the participants input list determining the ordering of participants within each subset (i.e., participants with identical attribute vectors). For example, suppose for the Motivating Example (1.2.1) that the user-specified attribute ordering is

$$[\text{Gender, Geographic Classification}],$$

and that the ordering of the participants input list is the one specified in Table 1.1. The ordering of all possible attribute vectors would be

$$[[\text{Woman, Rural}], [\text{Woman, Coastal City}], [\text{Man, Rural}], [\text{Man, Coastal City}]],$$

and the final ordering of participants would be

$$[WR_1, WR_2, WR_3, WC, MR, MC].$$

Note that this method of ordering participants means that the value of the random seed, which is used to shuffle the participants input list, only affects the ordering of participants with identical attribute vectors; the ordering of participants with distinct attribute vectors is deterministic given the attribute ordering.

(24) For a participant $p$, GroupSelect successively filters out groups as follows: For an attribute $a$, let $p[a]$ denote $p$’s attribute value for $a$. For each attribute $a$ in the user-specified attribute ordering, GroupSelect determines which groups $p[a]$ is currently most poorly represented in. It then filters out the rest of the groups and continues this process with the next attribute in the ordering. At the end of this process, $p$ is assigned to the first of the remaining groups.

Note that this method of assigning participants to groups, combined with the method of ordering participants in line 21 of the algorithm, results in allocations where all variation in group assignments arises from the treatment of participants with identical attribute vectors as interchangeable. In the Room for Improvement subsection below, we discuss the consequences of this behavior in more detail.
Room for Improvement

In practice, the GroupSelect algorithm is rather efficient and generates partitions quickly. It also packages its many features, including attribute ordering, clustering, and manual allocation, into an intuitive UI for organizers to use. However, thinking back to our Motivating Questions (1.2.2), we discover that there is still significant room for improvement.

First, GroupSelect fails to provide any algorithmic guarantees for attribute representation across tables. In the Motivating Example (1.2.1), for example, we were able to match the woman-to-man and rural-to-city ratios on each table with those of the actual assembly. While mirroring these ratios may not be realistic in practice (i.e., with messier real-world data, additional attributes, clustering, and so on), it is reasonable to think that organizers might still want the guarantee that their partitions’ attribute ratios fall within some deviation of the assembly’s. GroupSelect, however, only allows organizers to specify an attribute ordering, which provides no guarantees with respect to whether each table still reflects the diversity of the assembly.

On a related note, we also explained in Subsection 1.3.1 that for a fixed set of inputs (i.e., for a fixed assembly, attribute modes and orderings, desired number of groups, and number of seats available per group), the distribution of attributes across all allocations generated by GroupSelect is constant; in other words, regardless of the desired number of allocations and the value of the random seed, any allocation generated for the given set of inputs will have, e.g., 1 woman and 1 man on Table 1, 2 women on Table 2, and so on. One result of this constant distribution property is that between different allocations, all variation in group assignments arises from the treatment of participants with identical attribute vectors as interchangeable. When diversifying Gender and Geographic Classification in the Motivating Example (1.2.1), for example, $WR_1$, $WR_2$, and $WR_3$ are viewed as interchangeable, while $WR_1$ and $WC$ are not. In practice, this heuristic is convenient in that it allows GroupSelect to generate many allocations quickly. However, because the distribution of attributes is constant across all allocations, organizers again have little flexibility with respect to how the various attributes are represented across tables. As the
number of diversified attributes increases, participants are also less likely to share identical attribute vectors. This means that fewer unique allocations are generated, and the number of meetings that can be attained over all sessions decreases. Moreover, in the extreme case that everyone’s attribute vectors are unique (i.e., no participants can be treated as interchangeable), GroupSelect is only able to generate one unique allocation. This scenario is particularly problematic for organizers because all groups are fixed, and additional sessions do not generate any new meetings.

Finally, GroupSelect also fails to provide any algorithmic guarantees for the number of meetings generated over all sessions. Instead, it generates 100 allocations by randomly shuffling participants with identical attribute vectors and assigning them to groups where their attribute values are most underrepresented. It then randomly samples the desired number of sessions (say, 5) from these allocations 100 times and returns the sample that generates the most meetings over all 5 sessions. Thus, while this heuristic for generating meetings is rather efficient in practice, it is important to note that GroupSelect is not explicitly maximizing the number of interactions participants have over these sessions. As such, the algorithm is not able to provide organizers any guarantees with respect to how maximal the number of meetings generated for their particular inputs really is.

Code Availability

The implementation of GroupSelect is available at github.com/sortitionfoundation/groupselect-app. Note that the algorithm is not associated with a publication, so the information presented in this section was gathered from my own experiences with using it, as well as conversations with its creator Philipp Verpoort.

1.4 Contributions

In this report, we define a model for the assembly partitioning problem and propose a greedy, linear programming-based approach to generating allocations. We then conduct a series of experiments to compare our algorithm with GroupSelect, demonstrating in settings both with and without clustered participants that our algo-
algorithm achieves more desirable results. Finally, we derive a theoretical bound on the maximum possible number of links that can be achieved between back-to-back sessions of allocation in assemblies without clustered participants.

We briefly outline the remaining chapters as follows: In Chapter 2, we define key terms and formally model the assembly partitioning problem. In Chapter 3, we describe our linear-programming approach to designing an algorithm and formally introduce “the greedy algorithm,” as well as the various metrics and visualizations used in later chapters to compare its performance with GroupSelect’s. In Chapter 4, we introduce both synthetic and real-world assembly data and conduct experiments both with and without clustered participants to compare the performances of the two algorithms. In Chapter 5, we study a theoretical problem involving the minimum number of repeated meetings that can be achieved between back-to-back sessions of allocation in assemblies without clustered participants and thereby derive a bound on the maximum possible number of links that can be achieved between such sessions. Finally, in Chapter 6, we discuss the advantages and disadvantages of both the greedy algorithm and GroupSelect and suggest next steps for future research.
2 Preliminaries

2.1 Definitions

Before we formally model the assembly partitioning problem, we first define a few terms that will be helpful to us moving forward. For convenience, let \( n \) denote the number of participants, \( P \) denote the set of participants, and \( G \) denote the set of groups.

- **Attribute**: An attribute refers to a column of the participants’ attribute table. In Table 1.1 from the Motivating Example (1.2.1), for example, the attributes are Label, Gender, and Geographic Classification. In practice, attributes are typically labels (such as an ID associated with each participant) or demographic factors like race, gender, age group, income group, and so on.

- **Attribute value**: For each attribute, its attribute values are the values it takes on in the population (which, in our case, is the assembly). In Table 1.1, for example, the Gender attribute’s values are Woman and Man,
while the Geographic Classification attribute’s values are Rural and Coastal City. As most attributes are demographic factors, we may also refer to attribute values as demographics.

- **Attribute vector**: Each participant is associated with an attribute vector that captures their attribute values across all attributes. In Table 1.1, for example, participant WR1’s attribute vector can be represented as

  \[[WR1, Woman, Rural],\]

- **Meeting**: We say that two participants \(i\) and \(j\) (where \(i, j \in P\) and \(i < j\)) met in a session \(s\) if \(i\) and \(j\) were assigned to the same group \(g \in G\) for \(s\). Note by definition that participants may meet each other multiple times over the course of deliberation (i.e., over multiple sessions).

- **Link**: By contrast, links simply track whether two participants met in any session over the course of deliberation. In particular, we say that two participants \(i\) and \(j\) (where \(i, j \in P\) and \(i < j\)) form a link if \(i\) and \(j\) have been assigned to the same group \(g \in G\) at least once over the course of deliberation.

- **Discrepancy**: We formally define how to compute the discrepancy of each demographic as follows:

  First, for each demographic \(v\), let \(n_v\) denote the number of participants in the assembly that belong to \(v\). We define the proportion of participants in the assembly that belong to \(v\) as

  \[p_v = \frac{n_v}{n}.\]

  Similarly, for each group \(g \in G\) with size \(z_g\), let \(n_{v,g}\) denote the number of participants in \(g\) that belong to \(v\). The proportion of participants in \(g\) that belong to \(v\) is

  \[p_{v,g} = \frac{n_{v,g}}{z_g}.\]

  The discrepancy of the demographic \(v\) is then computed as the maximum absolute difference between \(p_v\) and \(p_{v,g}\) over all groups \(g\); in other words, the discrepancy of \(v\) is

  \[\max_{g \in [n_G]} \{|p_v - p_{v,g}|\}.\]

  Concretely, discrepancies give assembly organizers an idea of how closely the demographics of their allocation groups resemble those of the assembly. In particular, the closer the discrepancy of a particular demographic is to 0, the more the demographic’s representation in each group mirrors its representation in the actual assembly.

- **Diversify**: When we diversify an attribute (henceforth referred to as a diversified attribute), we aim to distribute its demographics across groups such that the demographics of each group resemble those of the assembly; this is akin to what we did in Partition 2 of Subsection 1.2.1 (Representation).

- **Cluster**: When we cluster a group of participants (henceforth referred to as clustered participants), we assign all of them to a select group of tables (clustered tables) across all allocations. Note that rather than
allowing organizers to cluster multiple attributes as in the GroupSelect algorithm (1.3.1), we simply model whether each participant needs to be clustered as a binary attribute. In practice, this is reasonable as clustered attributes are typically privacy-related and thus correlated\(^5\) (e.g., whether participants agree to be photographed or recorded), so participants who opt out of one are also likely to opt out of others. Moreover, for special cases of clustering in which participants must be grouped on the same table (rather than on any of the clustered tables), e.g., those who require interpreters, we can simply resort to manual allocation\(^6\).

2.2 Model

We now formally model the assembly partitioning problem.

2.2.1 Inputs

First, define an assembly as a set of \(n_P\) participants \([n_P] = \{0, \ldots, n_P - 1\}\) and a set of \(n_A\) attributes \([n_A] = \{0, \ldots, n_A - 1\}\). Each of the \(n_P\) participants is associated with a length-\(n_A\) attribute vector. The participants’ attribute table \(T\) can thus be formally defined as an \(n_P \times n_A\) matrix where each row \(i \in [n_P]\) is participant \(i\)'s attribute vector.

Next, define the set of \(n_G\) groups as \([n_G] = \{0, \ldots, n_G - 1\}\); the set of \(n_S\) sessions, or allocations, as \([n_S] = \{1, \ldots, n_S\}\);\(^7\) the set of \(n_D\) diversified attributes as \(D = \{d_1, \ldots, d_{n_D}\} \subseteq [n_A]\); the set of \(n_C\) clustered participants as \(C = \{c_1, \ldots, c_{n_C}\} \subseteq [n_P]\); and the set of \(n'_C\) clustered tables as \([n'_C] = \{0, \ldots, n'_C - 1\} \subseteq [n_G]\). Note that as in the GroupSelect algorithm, we require that \(n_D \geq 1\), while \(n_C\) may be \(0\); in other words, we require that at least one attribute be diversified, while choosing to cluster participants is optional. In order to constrain diversified attribute representation in groups, we also define a function \(R\) that maps each diversified attribute value \(v\) and group size \(z\) to a pair of bounds \((l_{v,z}, u_{v,z})\); for a full description, see Subsection 2.2.3.

\(^5\)This insight was gathered from conversations with Oliver Escobar, a Senior Lecturer in Public Policy at the University of Edinburgh, as well as GroupSelect’s creator Philipp Verpoort.

\(^6\)Note that we have yet to implement manual allocation as part of our algorithm defined in Section 3.3, but we plan to do so in the future (see our Future Work in Subsection 6.2.1).

\(^7\)Note that the set of sessions is 1-indexed rather than 0-indexed.
Putting everything together, the assembly partitioning problem takes as input an $n_P \times n_A$ participants’ attribute table $T$, the number of groups $n_G$, the number of sessions $n_S$, the set of diversified attributes $D$, the set of clustered participants $C$, the number of clustered tables $n'_C$, and the representation function $R$.

2.2.2 Output

The assembly partitioning problem outputs a function that takes as input a participant $i \in [n_P]$ and a session $s \in [n_S]$ and outputs the group $g \in [n_G]$ that $i$ has been assigned to for $s$.

2.2.3 Constraints

We now define the following constraints:

First, for each session, each participant must be assigned to exactly one group, and clustered participants must be assigned to one of their designated groups. We also constrain group sizes (i.e., the number of seats available in each group) with a function $Z$ that maps each group $g \in [n_G]$ to its size as follows:

$$Z(g) = \begin{cases} 
\lfloor n_P/n_G \rfloor + 1 & \text{if } g < (n_P \mod n_G) \\
\lfloor n_P/n_G \rfloor & \text{otherwise}
\end{cases}$$

In other words, we split participants among the groups as evenly as possible and distribute the remaining $r = (n_P \mod n_G)$ participants among the first $r$ groups, which each have an extra seat. We reason that modeling groups in such a way is both fair and practical, as participant responsibilities are roughly even across groups, and participants are not assigned to particularly small tables that are not only less likely to be representative of the assembly but also make it more difficult for them to meet new people that session. Moreover, because we define the set of $n'_C$ clustered tables as $[n'_C]$, any extra seats available are first allocated to clustered tables. As clustered participants have restricted seating assignments, we reason that doing so gives them a slightly higher probability of meeting new people (and, in particular, increases their exposure to non-clustered participants).
Finally, we also constrain diversified attribute representation in groups. For each diversified attribute value \( v \), the assembly partitioning problem takes as input a function \( R \) that maps each diversified attribute value \( v \) and group size \( z \) to a pair of bounds \((l_{v,z}, u_{v,z})\). These bounds constrain how many participants with attribute value \( v \) may be assigned to groups of size \( z \); in particular, for each group \( g \) of size \( z \), the number of participants with attribute value \( v \) that are assigned to group \( g \) must lie in the interval \([l_{v,z}, u_{v,z}]\). Note that because our group sizes are defined by the function \( Z \), only two distinct sizes are possible given \( n_P \) and \( n_G \). Thus, moving forward, we informally refer to groups with size \( \lfloor n_P/n_G \rfloor + 1 \) as the “larger” groups or tables.

2.2.4 Objective

As motivated in Chapter 1, organizers should aim to maximize the number of links generated over all sessions. As participants continue to meet each other over multiple sessions, however, it also makes sense that we would prefer to generate another meeting between a pair that has only met once versus one that has already met thrice; in other words, we reason that pairs that have met more often over previous sessions should be less likely than those who have met less often to meet in the current session. This not only matches our intuition of preferring pairs that have only met once over those that have already met thrice but also captures our original notion that pairs with no previous meetings should have the highest priority when it comes to meeting in the current session. We can thus formulate the objective of the assembly partitioning problem as a cost function as follows: Let \( s \) denote the current session. For each pair of participants \((i, j)\) where \( i, j \in [n_P] \) and \( i < j \), let \( m_{i,j} \) denote the total number of times \( i \) and \( j \) met over the previous sessions \( 1, \ldots, s - 1 \), and let \( I_{i,j} \) indicate whether \( i \) and \( j \) meet this current session. The objective, which formalizes the properties motivated above, can be written as

\[
\min \left\{ \sum_{i,j \in [n_P] \text{ s.t. } i < j} I_{i,j}m_{i,j} \right\}.
\]
In this chapter, we introduce our algorithm for the assembly partitioning problem.

3.1 ILP Approach

We first describe our approach to designing the algorithm.

As modeled in Section 2.2, the assembly partitioning problem lends itself quite naturally to an integer linear programming (ILP) approach. In particular, such an approach allows us to structure our algorithm as the optimization of a linear objective subject to various linear constraints, which is convenient as we can simply attempt to translate our model (2.2) into a program that captures the desired properties. As we will discuss next, many optimization solvers (both commercial and open-source) are also readily available online and offer helpful interfaces.
for modeling ILPs and other programs, including quadratic programs (QPs) and various types of mixed-integer programs (MIPs). Before we continue with such an approach, however, it is important to note that integer programming is NP-complete, and it may be the case that our ILPs cannot be solved to provable optimality within a reasonable time limit.

In order to implement the ILPs for our algorithm, we take advantage of Gurobi, a powerful commercial optimization solver. Its Python API (gurobipy) is not only extensively documented but also offers all of the features needed to model the assembly partitioning problem. Additional information on the API, including its documentation, is available at gurobi.com/documentation/9.5/refman/py_python_api_overview.html.

3.2 Initial Algorithm

Our initial algorithm involved solving a single ILP that aimed to maximize the number of links generated over all sessions. While simple, however, this approach proved to be too inefficient in practice: Gurobi struggled to find optimal solutions for even the smallest (e.g., under 10-participant) assemblies. Following these experiments, we hypothesized that the solution space over all sessions was too large for Gurobi to optimize over within a reasonable time limit. We thus chose to proceed with a greedy approach in which sessions are optimized sequentially (i.e., we solve a new ILP for each session) and describe this approach in the next section.

For a full description of the initial model and algorithm, see Appendix A.

3.3 Greedy Algorithm

We now describe our algorithm for the assembly partitioning problem.

As explained in Section 3.2, we proceed with a greedy approach in which sessions are optimized sequentially. This means that for each session, we solve a new ILP to generate an allocation that minimizes the cost of the current allocation.
rent session’s meetings given the total number of meetings each pair of participants has had prior to the current session. Before we proceed, note that we refer to the user inputs defined in Section 2.2 throughout the rest of this section.

3.3.1 ILP (per Session)

We first formalize the ILP used to generate an allocation for each session. Note that we default Gurobi’s time limit for each ILP to 30 seconds, but an organizer may specify their own time limit as they see fit.

Variables

Note that all definitions below apply to just one session (i.e., the “current” session, which we will denote $s$).

- $x$: For $i \in [n_P]$ and $g \in [n_G]$, let $x_{i,g}$ be a binary variable that equals 1 if and only if participant $i$ is assigned to group $g$ in session $s$.

- $y$: For $i, j \in [n_P]$ where $i < j$ and $g \in [n_G]$, let $y_{(i,j),g}$ be a binary variable that equals 1 if and only if participants $i$ and $j$ met in group $g$ during session $s$. These variables will help us define the cost function for the objective.

- $z$: For $i, j \in [n_P]$ where $i < j$, let $z_{i,j}$ be an integer variable equal to the cost of a meeting between participants $i$ and $j$ in session $s$ (where the cost of such a meeting is defined in Subsection 2.2.4). These variables will help us define the cost function for the objective.

Solution

For each ILP, the solution we wish to record is an assignment from participants to groups. This information is easily gathered from the values of the $x$-variables: Each participant $i$ is assigned to the group $g$ for which $x_{i,g} = 1$.

Constraints

We first constrain the $y$- and $z$-variables according to their definitions above.
• \( y \): For \( i, j \in [n_P] \) where \( i < j \) and \( g \in [n_G] \), we have by definition that
\[
y(i,j)_g = (x_{i,g} \land x_{j,g}).
\]
In other words, participants \( i \) and \( j \) met in group \( g \) if both \( i \) and \( j \) were assigned to group \( g \). The logical AND in the statement above can then be rewritten using the following 3 linear constraints:
\[
\begin{align*}
y(i,j)_g & \geq x_{i,g} + x_{j,g} - 1, \\
y(i,j)_g & \leq x_{i,g}, \\
y(i,j)_g & \leq x_{j,g}.
\end{align*}
\]

• \( z \): As part of the algorithm, we track the total number of meetings each pair of participants has had prior to the current session (see lines 11-14 of Algorithm 2). Thus, for each \( i, j \in [n_P] \) where \( i < j \), let \( m_{i,j} \) denote this number. We have by definition that
\[
z_{i,j} = m_{i,j} \sum_{g \in [n_G]} y(i,j)_g,
\]
where \( \sum_{g \in [n_G]} y(i,j)_g \) indicates whether \( i \) and \( j \) meet this current session.

We now formalize the constraints outlined in Section 2.2.

• Each participant must be assigned to exactly one group:
\[
\forall i \in [n_P], \sum_{g \in [n_G]} x_{i,g} = 1.
\]

• Clustered participants must be assigned to one of their designated groups:
\[
\forall i \in C, \sum_{g \in [n'_G]} x_{i,g} = 1.
\]

• The number of participants that can be assigned to each group is constrained by the size function \( Z \) defined in Subsection 2.2.3:
\[
\forall g \in [n_G], \sum_{i \in [n_P]} x_{i,g} = Z(g).
\]

• Diversified attribute representation is constrained according to user-inputted bounds:
First, for each participant \( i \in [n_P] \), let \( P_i \) denote \( i \)'s attribute vector. Then, the demographic that \( i \) belongs to for attribute \( a \in [n_A] \) is simply \( P_i[a] \). For a diversified attribute value \( v \) and group size \( z \), also let
\[ R(v, z) = (l_{v, z}, u_{v, z}). \] We desire that
\[ \forall d \in D, \forall v_d, \forall g \in [n_G], l_{v_d, z(g)} \leq \sum_{i \in [n_P] \text{ s.t. } P[d] = v_d} x_{i, g} \leq u_{v_d, z(g)}, \]

where \( v_d \) denotes one of \( d \)'s attribute values.

**Objective**

Finally, we translate the cost function from Subsection 2.2.4 as follows:

\[
\min \left\{ \sum_{i, j \in [n_P] \text{ s.t. } i < j} z_{i, j} \right\}.
\]

3.3.2 **Algorithm**

We now use the ILP defined in Subsection 3.3.1 to present the greedy algorithm (p. 25).
Algorithm 2 Greedy

1: function Greedy(inputs) \(\triangleright\) See user inputs defined in Subsection 2.2.
2: Validate user inputs
3:
4: allocs ← []
5: meetings_record ← {} \(\triangleright\) Map from pairs \((i, j)\) (where \(i < j\)) to # of prior meetings they’ve had
6: for session in range(n_s) do
7: alloc ← SOLVEILP(inputs, session)
8: Add alloc to allocs
9: for pair do
10: if pair met in alloc then
11: meetings_record[pair] += 1 \(\triangleright\) Add meetings from alloc to meetings_record
12: end if
13: end for
14: Print summary metrics \(\triangleright\) See Subsection 3.3.3
15: end for
16: Visualize meetings distributions \(\triangleright\) See Subsection 3.3.4
17: return allocs
18: end function
19:
20: function SOLVEILP(inputs, session)
21: return (solution to session’s ILP, as presented in Subsection 3.3.1)
22: end function
3.3.3 **Metrics (per Session)**

After generating an allocation for the current session, we print a series of summary metrics (see line 16 of Algorithm 2). In Chapter 4, which covers our Empirical Results, we use these metrics to compare the quality of allocations generated by our algorithm and GroupSelect. We thus formally define them below:

- **Optimization status code**: For each ILP, we output Gurobi’s optimization status code for the result of that ILP. These codes indicate, for example, whether a model was solved to optimality (code `OPTIMAL` with value 2), or if optimization had to be terminated because the time limit was exceeded (code `TIME_LIMIT` with value 9); see [gurobi.com/documentation/9.5/refman/optimization_status_codes.html](http://gurobi.com/documentation/9.5/refman/optimization_status_codes.html) for additional information.

- **Number of new links**: We use the term *new links* here to refer to links generated in the current session.

- **Number of (overall) links**: We use the term *overall links*, or just *links*, here to refer to links generated over all sessions thus far (i.e., all prior sessions, in addition to the current session).

- **Avg. number of links per participant**: Recall that a link is defined between two participants $i$ and $j$ where $i,j \in [n_P]$ and $i < j$, so the average number of links per participant is computed as
  \[
  \frac{2 \cdot \text{(number of links)}}{n_P}
  \]

- **Maximum possible number of links**: This is computed naively as
  \[
  \binom{n_P}{2},
  \]
  i.e., we count one possible link for each pair of participants. Thus, when computing the percentage of possible links formed, we do so with respect to this number. In Chapter 5, which covers our Theoretical Results, however, we define a more precise notion of the maximum possible number of links.

- **Updates to meetings_record**: For each session, we track updates to `meetings_record` (see lines 11-14 of Algorithm 2) by outputting the number of pairs that “went from $m$ to $m + 1$ meetings,” where $m$ is the number of meetings that the pairs have had prior to the current session. If 10 pairs met for the first time in the current session, for example, we would report that “10 pairs went from 0 meetings to 1 meeting.”

- **Discrepancies**: As defined in Section 2.1, discrepancies are computed for each demographic.

For assemblies with at least one clustered participant\(^{10}\), we also output a few additional metrics. For convenience, let $\Delta = [n_P] \setminus C$ denote the set of non-clustered participants.

\(^{10}\)Recall from Subsection 2.2.1 that choosing to cluster participants is optional.
• **Avg. number of links per clustered participant:** This is analogous to the average number of links per participant, except on the set $C$.

• **Avg. number of links per non-clustered participant:** This is analogous to the average number of links per participant, except on the set $\Delta$.

• **Avg. number of meetings between pairs of clustered participants:** This is computed as

$$\frac{\sum_{i,j \in C \text{ s.t. } i < j} \text{meetings_record}[i,j]}{\binom{n_C}{2}},$$

where `meetings_record` is defined in Algorithm 2.

• **Avg. number of meetings between pairs of non-clustered participants:** This is computed as

$$\frac{\sum_{i,j \in \Delta \text{ s.t. } i < j} \text{meetings_record}[i,j]}{\binom{n_P - n_C}{2}}.$$

• **Avg. number of meetings between pairs of one clustered and one non-clustered participants:** This is computed as

$$\frac{\sum_{(i \in C, j \in \Delta \text{ s.t. } i < j) \cup (i \in \Delta, j \in C \text{ s.t. } i < j)} \text{meetings_record}[i,j]}{n_C(n_P - n_C)}.$$

### 3.3.4 Visualizations

At the end of the algorithm, we also generate a few visualizations to help organizers gain a better understanding of how meetings are distributed among pairs of participants; below is an overview of these visualizations:

• **New meetings distribution:** This essentially visualizes the “updates to meetings_record” metric for the final session, i.e., we plot the number of meetings pairs had prior to the final session along the x-axis and the number of pairs that went from $x$ to $x + 1$ meetings (in the final session alone) along the y-axis. Such a plot gives organizers a sense of how “new” the meetings generated by the final session are (i.e., are many pairs still meeting for the first time, or are they mostly meeting for the second or third time?).
In Figure 3.1a, for example, we can see in session 2 that 35 pairs are meeting for the first time (with $x = 0$ prior meetings), while only 5 pairs are meeting for the second time (with $x = 1$ prior meetings). This indicates that the majority of pairs are still meeting for the first time, though the algorithm is also beginning to generate repeated (i.e., two or more) meetings between pairs.

If we extend the run by 6 sessions, however, we can see in Figure 3.1b that the algorithm is now mostly generating repeated meetings, with only 2 pairs meeting for the first time. As the algorithm (via the ILP setup described in Subsection 3.3.1) gives pairs with fewer previous meetings higher priority when it comes to meeting in the current session, a plot like Figure 3.1b may indicate to organizers that the majority of links have already been achieved, and a significant number of pairs moving forward (if, say, the run is extended again) will not be meeting for the first time.

- **(Overall) meetings distribution:** This visualizes the distribution of meetings over all sessions, i.e., we plot the number of meetings pairs had along the $x$-axis and the number of pairs that met $x$ times (over all sessions) along the $y$-axis. Such a plot gives organizers a sense of how frequently participants met each other over all sessions.
In Figure 3.2a, for example, we can see from the 2-session distribution that 115 pairs have yet to meet (i.e., 115 links have yet to be achieved), 70 pairs have met once, and 5 pairs have met twice. If we again extend the run by 6 sessions, however, we can see in Figure 3.2b that only 31 links have yet to be achieved, and thus the majority of pairs moving forward will not be meeting for the first time.

3.3.5 Code Availability

The implementation of our greedy algorithm is available at github.com/rosemhong/sortition-table-allocation.
4

Empirical Results

In this chapter, we present the results of running both the greedy algorithm and GroupSelect on synthetic and real-world assembly data. In order to compare the quality of allocations generated by the two algorithms, we use the metrics and visualizations described in Subsections 3.3.3 and 3.3.4 respectively.

4.1 Datasets

We first introduce the datasets used to compare the two algorithms in the experiments that follow. Note that we randomly generate 3 versions of each toy dataset (with the generation ratios of demographics fixed as specified in the descriptions below) in order to run 3 trials of each experiment.

• toy_20: These toy datasets each contain 20 participants, 2 diversifiable attributes with values listed in
Table 4.1, and 1 binary Cluster attribute that can be toggled to cluster a group of 6 participants. The demographics for each attribute were randomly generated in the ratios 3 : 2 for Woman to Man and 1 : 1 : 1 for High School to Undergraduate to Postgraduate, while the group of 6 clustered participants was randomly chosen from the set of 20 participants. One version of these datasets, with the realized counts for each demographic included in parentheses, is listed below.

<table>
<thead>
<tr>
<th>Diversifiable Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
<td>Woman (12), Man (8)</td>
</tr>
<tr>
<td>Education Level</td>
<td>High School (6), Undergraduate (6), Postgraduate (8)</td>
</tr>
</tbody>
</table>

**Table 4.1: toy_20 Dataset (Trial 1)**

- **toy_100**: These toy datasets each contain 100 participants and 4 diversifiable attributes with values listed in Table 4.1; note that they do not contain Cluster attributes. The attribute values for each attribute were randomly generated in the ratios 1 : 1 for Woman to Man, 1 : 1 for Employed to Unemployed, 2 : 2 : 1 for California to Texas to Florida, and 5 : 3 : 2 for High School to Undergraduate to Postgraduate. One version of these datasets, with the realized counts for each demographic included in parentheses, is listed below.

<table>
<thead>
<tr>
<th>Diversifiable Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gender</td>
<td>Woman (53), Man (47)</td>
</tr>
<tr>
<td>Employment Status</td>
<td>Employed (45), Unemployed (55)</td>
</tr>
<tr>
<td>State</td>
<td>California (36), Texas (40), Florida (24)</td>
</tr>
<tr>
<td>Education Level</td>
<td>High School (49), Undergraduate (30), Postgraduate (21)</td>
</tr>
</tbody>
</table>

**Table 4.2: toy_100 Dataset (Trial 1)**

- **hd_30**: This dataset contains 30 participants, 6 diversifiable attributes with values listed in Table 4.1, and 1 binary “f” attribute that can be toggled to cluster a group of 6 participants. It was acquired from a real-world Healthy Democracy panel held in the past, and all fields have been anonymized.

<table>
<thead>
<tr>
<th>Diversifiable Attribute</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>a1 (4), a2 (4), a3 (4), a4 (3), a5 (4), a6 (2), a7 (4), a8 (2), a9 (3)</td>
</tr>
<tr>
<td>b</td>
<td>b1 (15), b2 (15)</td>
</tr>
<tr>
<td>c</td>
<td>c1 (8), c2 (4), c3 (6), c4 (4), c5 (3), c6 (3), c7 (2)</td>
</tr>
<tr>
<td>d</td>
<td>d1 (15), d2 (13), d3 (2)</td>
</tr>
<tr>
<td>e</td>
<td>e1 (2), e2 (2), e3 (4), e4 (2), e5 (2), e6 (18)</td>
</tr>
<tr>
<td>g</td>
<td>g1 (2), g2 (5), g3 (11), g4 (12)</td>
</tr>
</tbody>
</table>

**Table 4.3: hd_30 Dataset**

[healthydemocracy.org/]
Finally, recall that GroupSelect requires an attribute ordering to be specified. In all of the following experiments, the order we use for the toy_20, toy_100, and hd_30 datasets is simply the order attributes are listed in Tables 4.1, 4.2, and 4.3 respectively. For the hd_30-6g-2cg-3d experiments (4.4.2), for example, we diversify attributes in the order b, d, and g.

4.2 Basis of Comparison

As explained in 4.1, we run 3 trials of each experiment. In order to fairly compare the links generated by the greedy algorithm and GroupSelect, we define our representation function $R$ for each trial with respect to GroupSelect’s results on the same inputs. In particular, for each trial, we first generate allocations using GroupSelect. We then define $R$ as follows: For each diversified attribute value $v$ and group $g \in [n_G]$, let $n_{v,g}$ denote the number of participants GroupSelect has assigned to $g$ that belong to $v$. Then, for each diversified attribute value $v$ and group size $z$, let

$$l_{v,z} = \min_{g \in [n_G] \text{ s.t. } Z(g) = z} \{n_{v,g}\},$$

$$u_{v,z} = \max_{g \in [n_G] \text{ s.t. } Z(g) = z} \{n_{v,g}\}.$$

We define $R(v, z) = (l_{v,z}, u_{v,z})$. Because the representation function $R$ constrains how many participants with attribute value $v$ the greedy algorithm may assign to groups of size $z$, defining $R$ in such a way guarantees that the discrepancies of allocations generated by the greedy algorithm are at least as good as (i.e., less than or equal to) those of allocations generated by GroupSelect. We can thus establish a fair comparison of the links generated by the two algorithms. Note, however, that we only define $R$ in such a way for our experimental comparisons with GroupSelect; in practice, organizers are free to define $R$ as they see fit.

\[^{12}\text{Recall from 1.3.1 that for a particular set of inputs, the distribution of attributes across all allocations generated by GroupSelect is constant. Thus, in order to define } n_{v,g} \text{ here, we only need to look at one of the allocations generated by GroupSelect.}\]
Hardware

All experiments were run on a 2019 MacBook Pro with a 2.3 GHz 8-core Intel i9 processor.

4.3 Experiment Naming Conventions

For convenience, we use the following abbreviations to name and describe experiments:

- G: group(s)
- CG: clustered group(s)
- D: diversified attribute(s)
- CP: clustered participant(s)
- NCP: non-clustered participant(s)
- CP-CP: pairs of clustered participants
- CP-NCP: pairs of one clustered and one non-clustered participants
- NCP-NCP: pairs of non-clustered participants

For the hd_30-6g-2cg-3d experiments (4.4.2), for example, we run both algorithms on the hd_30 dataset with 6 groups (2 of which are clustered) and 3 diversified attributes.

Finally, we occasionally abbreviate GroupSelect as GS.

4.4 Experiments with Clustered Participants

In this section, we examine how the two algorithms perform on experiments with at least one clustered participant. Before we proceed, we first clarify how the notion of clustering differs between the two algorithms.

Recall from Subsection 1.3.1 that GroupSelect clusters attributes rather than participants, i.e., for each clustered attribute, the algorithm roughly tries to group participants with the same attribute values together. In our approach, however, we model whether each participant needs to be clustered as a binary attribute (see Subsection
2.1 for justification). In practice, this means that organizers who want to cluster a group of participants can simply add a “Cluster” attribute that indicates whether each participant should be clustered. In order to establish a fair comparison between the two algorithms, however, we would also like to emulate this behavior in GroupSelect. Luckily, this is easy—the same Cluster attribute can simply be put in GroupSelect’s Cluster mode in order to achieve the same effect.

Another difference between the two algorithms, as described in Subsection 1.3.1, is that GroupSelect clusters a binary attribute (say, the Cluster attribute described above) by distributing all of the clustered participants among as few groups as possible. For example, if an organizer has 8 clustered participants and wants to split their assembly into groups of size 4, then GroupSelect will assign them to 2 groups with 4 clustered participants each. However, this behavior significantly limits the ability of these clustered participants to form new links and may also negatively impact the resulting allocations’ discrepancies, as there is no guarantee that groups with many clustered participants will be as representative as those without. The greedy algorithm, on the other hand, alleviates this issue by actually allowing organizers to specify the number of clustered tables. Thus, the greater this number is, the less restrictive clustered participants’ seating assignments are, and the more likely these participants are to form new links and be a part of diverse groups each session. In order to establish a fair comparison between the two algorithms, however, we set the number of clustered tables in each of the greedy experiments to be equal to the number allocated by GroupSelect.

Finally, because we define our representation function \( R \) for each experiment with respect to GroupSelect’s results on the same inputs\(^{13}\) (see our Basis of Comparison in Section 4.2), and we also noted earlier that poor representation in clustered groups may negatively impact the resulting GroupSelect allocations’ discrepancies, we acknowledge that the representation bounds given as input to the greedy algorithm for experiments with clustered participants may be rather loose. One way to address this concern could be to constrain clustered tables separately so that the bounds drawn from GroupSelect’s clustered tables apply only to our clustered tables rather

\(^{13}\)Recall that this “mirroring” only applies to our experimental comparisons with GroupSelect; in practice, organizers are free to define \( R \) as they see fit.
than all tables; we could then define a separate, potentially tighter set of bounds drawn from GroupSelect’s non-clustered tables to apply to our own non-clustered tables. For our experimental comparisons with GroupSelect, however, we argue that this is not necessary for two reasons: First, discrepancies are defined with respect to representation across all groups, so even with looser bounds applied to all groups, the discrepancies of allocations generated by the greedy algorithm will always be at most as large as those of allocations generated by GroupSelect. Moreover, in the next section, we compare the performances of the two algorithms on experiments without any clustered participants. Because there is no longer a distinction between clustered and non-clustered tables, those experiments help us address the concern raised here of whether the greedy algorithm only performs well because representation on non-clustered tables is too loosely constrained.

4.4.1 toy_20 (with Clustered Participants)

We first ran experiments on the toy_20 assemblies with 6 clustered participants and both diversifiable attributes (i.e., Gender and Education Level) diversified, varying the number of groups.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>$n_G$</th>
<th>$n'_G$</th>
<th>$n_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_20-2g-1cg</td>
<td>2</td>
<td>1</td>
<td>8</td>
</tr>
<tr>
<td>toy_20-4g-2cg</td>
<td>4</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>toy_20-6g-2cg</td>
<td>6</td>
<td>2</td>
<td>24</td>
</tr>
</tbody>
</table>

*Table 4.4: toy_20 with CP - Experiments*

Recall that the number of clustered groups for each experiment is defined as the number of groups that GroupSelect assigns clustered participants to, i.e.,

$$[n_C/n_G],$$

and that participants are distributed among groups as evenly as possible by both algorithms. In the toy_20-6g-2cg experiment with 6 groups, for example, the 20 participants are split into 2 groups of size 4 and 4 groups of size 3 for each session.

In Table 4.5, we list the average runtime per session for each experiment.
### Table 4.5: `toy_20` with CP – Average Runtimes per Session

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Greedy – Avg. Runtime / Session (s)</th>
<th>GS – Avg. Runtime / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>toy_20-2g-1cg</code></td>
<td>0.042 ± 0.016</td>
<td>0.018 ± 0.006</td>
</tr>
<tr>
<td><code>toy_20-4g-2cg</code></td>
<td>0.579 ± 0.207</td>
<td>0.011 ± 0.003</td>
</tr>
<tr>
<td><code>toy_20-6g-2cg</code></td>
<td>0.279 ± 0.202</td>
<td>0.007 ± 0.001</td>
</tr>
</tbody>
</table>

We can see that while GroupSelect’s runtimes are lower, both algorithms are able to generate allocations extremely quickly, with all trials terminating in a matter of seconds. It is also clear across all of these experiments that the greedy algorithm is able to find optimal allocations well ahead of the default time limit of 30s per session.

For each experiment, the number of links formed among participants was also collected at regular intervals (“checkpoints”). In Figure 4.1, we plot the percentage of possible links formed (where, as described in Subsection 3.3.3, the maximum number of possible links is simply computed as \( \binom{n}{2} \)) against the number of sessions (up to \( n_S \) total sessions) generated thus far for each algorithm.

![toy_20-2g-1cg: % of possible links formed vs. session](image)

![toy_20-4g-2cg: % of possible links formed vs. session](image)

![toy_20-6g-2cg: % of possible links formed vs. session](image)

**Figure 4.1**: `toy_20` with CP – Percentage of Possible Links Formed vs. Session
For the toy_20-4g-2cg and toy_20-6g-2cg experiments, it is clear that the greedy algorithm has generated significantly more links at each checkpoint than GroupSelect has, with the algorithms tapering off at $\approx 0.93 \pm 0.064$ and $\approx 0.726 \pm 0.036$ respectively after 14 sessions of toy_20-4g-2cg, and $\approx 0.753 \pm 0.087$ and $\approx 0.496 \pm 0.046$ respectively after 24 sessions of toy_20-6g-2cg. For toy_20-2g-1cg, on the other hand, the difference between the percentage of possible links formed by each algorithm at each checkpoint is not statistically significant. After 8 sessions, for example, the greedy algorithm has generated $0.947 \pm 0.091$ of possible links, while GroupSelect has generated $0.809 \pm 0.058$ of possible links. Furthermore, we actually observe in one of these toy_20-2g-1cg trials that the greedy algorithm occasionally generates slightly fewer links than GroupSelect does.

<table>
<thead>
<tr>
<th>Session</th>
<th>Greedy – Links Formed</th>
<th>GroupSelect – Links Formed</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>138</td>
<td>138</td>
</tr>
<tr>
<td>4</td>
<td>156</td>
<td>158</td>
</tr>
<tr>
<td>6</td>
<td>158</td>
<td>160</td>
</tr>
<tr>
<td>8</td>
<td>160</td>
<td>160</td>
</tr>
</tbody>
</table>

*Table 4.6: toy_20-2g-1cg (Trial 1) – Links Formed vs. Session*

This can ultimately be attributed to the greedy algorithm’s objective: It not only prioritizes links (i.e., first meetings between pairs of participants) but also aims to reduce the overall cost of meetings so that the more of-
ten pairs have met in the past, the less likely they are to meet again in the current session. In Figure 4.2, which compares the meetings distributions of the two algorithms after 6 sessions of toy_20-2g-1cg (trial 1), for example, we can see that the greedy allocations result in fewer pairs that have met in over half of the sessions. Moreover, as we stated earlier, the difference between the percentage of possible links formed by each algorithm at each check-point over all toy_20-2g-1cg trials is not statistically significant. It is thus clear from the experiments in Figure 4.1 that the greedy algorithm generally does a better job of generating new links among toy_20 participants than GroupSelect does.

From Figure 4.1, we can also see that the “gap” between the percentage of possible links generated by the two algorithms grows as the number of groups increases. After 8 sessions, for example, the GroupSelect-to-greedy ratio of the percentage of possible links formed is $\approx 0.902 \pm 0.085$ for toy_20-2g-1cg and $\approx 0.712 \pm 0.083$ for toy_20-6g-2cg. We hypothesize that this difference in the performances of the two algorithms is related to the growing size of the solution space as the number of groups increases from 2 to 6. In particular, because GroupSelect resorts to a constant distribution heuristic (see Subsection 1.3.1), while the greedy algorithm actually searches for an optimal allocation over this space, the greedy algorithm is more likely find desirable (even if not optimal) allocations that GroupSelect simply cannot generate using its heuristic in a larger space. This trend thus suggests that as the size of the solution space grows, GroupSelect experiences more difficulty generating new links among toy_20 participants than the greedy algorithm does.

Next, we quantify the experiences of clustered participants. In Table 4.7, we compute ratios between the average number of links formed per clustered participant and the average number of links formed per non-clustered participant.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Avg. Links / CP</th>
<th>Avg. Links / NCP</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>2g-1cg</td>
<td>16.667 ± 2.517</td>
<td>18 ± 1.079</td>
</tr>
<tr>
<td>GS</td>
<td>2g-1cg</td>
<td>12.111 ± 2.457</td>
<td>14.714 ± 3.347</td>
</tr>
<tr>
<td>Greedy</td>
<td>4g-2cg</td>
<td>16.778 ± 2.037</td>
<td>18.048 ± 0.873</td>
</tr>
<tr>
<td>GS</td>
<td>4g-2cg</td>
<td>10.5 ± 1.856</td>
<td>15.214 ± 0.378</td>
</tr>
<tr>
<td>Greedy</td>
<td>6g-2cg</td>
<td>14.279 ± 1.602</td>
<td>14.31 ± 1.679</td>
</tr>
<tr>
<td>GS</td>
<td>6g-2cg</td>
<td>5.111 ± 3.791</td>
<td>11.286 ± 0.378</td>
</tr>
</tbody>
</table>

Table 4.7: toy_20 with CP – Ratio of Average Links Formed by Clustered and Non-Clustered Participants

In general, we expect non-clustered participants to form more links because they aren’t restricted to clustered tables, so a ratio closer to 1 indicates that clustered participants—despite their restricted seating assignments—are forming nearly as many links as non-clustered participants are. From Table 4.7, we can see for the toy_20-4g-2cg and toy_20-6g-2cg experiments that the greedy ratio between average links formed by clustered and non-clustered participants is significantly larger than the respective GroupSelect ratio; for toy_20-2g-1cg, on the other hand, the difference between the ratios is not statistically significant. Overall, the greedy algorithm thus does a better job of helping toy_20’s 6 clustered participants meet new people. Upon closer inspection of the data, we also observe that the difference between the ratios is particularly large for the toy_20-6g-2cg experiments, with GroupSelect generating very few links on average for each clustered participant. This behavior can ultimately be attributed to GroupSelect’s choice to “pack” the 6 clustered participants onto two size-3 tables for trials 2 and 3 rather than two size-4 tables, which prevents the 6 clustered participants from meeting any non-clustered participants. The greedy algorithm, on the other hand, does not experience this issue because it guarantees that any extra seats available are first allocated to clustered tables (see Subsection 2.2.3). These experiments thus demonstrate the importance of not only allowing organizers to specify the number of clustered tables but also choosing the larger tables wisely.

In Table 4.8, we compute the average number of meetings generated between different types of participant pairings.
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy 2g-1cg</td>
<td>8 ± 0</td>
<td>2.29 ± 0</td>
<td>4.484 ± 0</td>
</tr>
<tr>
<td>GS 2g-1cg</td>
<td>8 ± 0</td>
<td>2.286 ± 0</td>
<td>4.484 ± 0</td>
</tr>
<tr>
<td>Greedy 4g-2cg</td>
<td>5.78 ± 0.168</td>
<td>1.94 ± 0.03</td>
<td>3.414 ± 0.235</td>
</tr>
<tr>
<td>GS 4g-2cg</td>
<td>5.6 ± 0</td>
<td>2 ± 0</td>
<td>3.385 ± 0</td>
</tr>
<tr>
<td>Greedy 6g-2cg</td>
<td>9.73 ± 0.115</td>
<td>1.67 ± 0.02</td>
<td>3.187 ± 0</td>
</tr>
<tr>
<td>GS 6g-2cg</td>
<td>9.6 ± 0</td>
<td>0.571 ± 0.09</td>
<td>4.22 ± 0.913</td>
</tr>
</tbody>
</table>

Table 4.8: toy_20 with CP – Average Meetings per Type of Participant Pairing

We can see that across most of the experiments\textsuperscript{44}, the respective averages for the two algorithms are similar; this indicates that the algorithms distribute meetings over the different types of participant pairings in similar ways. In general, the average number of meetings generated between clustered and non-clustered participants is also smaller than the average number generated for both clustered pairs and non-clustered pairs. This means that in any given session, clustered participants are more likely to meet other clustered participants, while non-clustered participants are more likely to meet other non-clustered participants.

4.4.2 \textit{hd}_30 (with Clustered Participants)

Next, we ran experiments on the \textit{hd}_30 assembly with 6 clustered participants, varying the number of groups and diversified attributes.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>$n_G$</th>
<th>$n_c$</th>
<th>$n_D$</th>
<th>$n_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{hd}_30-3g-1cg-3d</td>
<td>3</td>
<td>1</td>
<td>3 (b, d, g)</td>
<td>10</td>
</tr>
<tr>
<td>\textit{hd}_30-6g-2cg-3d</td>
<td>6</td>
<td>2</td>
<td>3 (b, d, g)</td>
<td>30</td>
</tr>
<tr>
<td>\textit{hd}_30-3g-1cg-6d</td>
<td>3</td>
<td>1</td>
<td>6 (a, b, c, d, e, g)</td>
<td>14</td>
</tr>
<tr>
<td>\textit{hd}_30-6g-2cg-6d</td>
<td>6</td>
<td>2</td>
<td>6 (a, b, c, d, e, g)</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 4.9: \textit{hd}_30 with CP – Experiments

In Table 4.10, we can see that GroupSelect’s runtimes continue to be lower than the greedy algorithm’s. For the \textit{hd}_30-6g-2cg-3d and \textit{hd}_30-6g-2cg-6d experiments, the average runtimes for the greedy algorithm also begin

\textsuperscript{44}As noted in the previous paragraph, the major exception is the toy_20-6g-2cg experiments, for which GroupSelect generates very few (i.e., 0 for trials 2 and 3) meetings on average between clustered and non-clustered participants.
to approach the default time limit of 30s per session, which indicates that Gurobi is timing out before optimal allocations can be found.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Greedy – Avg. Runtime / Session (s)</th>
<th>GS – Avg. Runtime / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd_30-3g-1cg-3d</td>
<td>3.302</td>
<td>0.019</td>
</tr>
<tr>
<td>hd_30-6g-2cg-3d</td>
<td>26.873</td>
<td>0.011</td>
</tr>
<tr>
<td>hd_30-3g-1cg-6d</td>
<td>0.813</td>
<td>0.162</td>
</tr>
<tr>
<td>hd_30-6g-2cg-6d</td>
<td>22.046</td>
<td>0.232</td>
</tr>
</tbody>
</table>

Table 4.10: hd_30 with CP – Average Runtimes per Session

Now, as noted in Section 3.1, integer programming is NP-complete, and we find for both these experiments and ones without clustered participants that follow in Subsections 4.5.2 and 4.5.3 that some of our ILPs cannot be solved to provable optimality within a reasonable time limit. In particular, while optimal solutions can typically be found for the first few sessions, Gurobi times out before the optimal solution can be found for each session thereafter. Thus, for these sessions, Gurobi simply uses the best solution it has found thus far. While this solution does not minimize the ILP’s cost function, we will actually soon see that these solutions are still quite good with respect to the metrics we care about.

In Table 4.11, we list Gurobi’s optimization status codes for each experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Status Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd_30-3g-1cg-3d</td>
<td>2’s for all sessions</td>
</tr>
<tr>
<td>hd_30-6g-2cg-3d</td>
<td>2’s for sessions 1-4, 9’s thereafter</td>
</tr>
<tr>
<td>hd_30-3g-1cg-6d</td>
<td>2’s for all sessions</td>
</tr>
<tr>
<td>hd_30-6g-2cg-6d</td>
<td>2’s for sessions 1-6, 9’s thereafter</td>
</tr>
</tbody>
</table>

Table 4.11: hd_30 with CP – Status Codes

Recall from Subsection 3.3.3 that 2 corresponds to the code OPTIMAL and 9 corresponds to the code TIME_LIMIT, so we can see in hd_30-6g-2cg-3d, for example, that the sessions 1-4 ILPs were solved to optimality, while the remaining 26 ILPs were terminated because the default time limit of 30s per session was exceeded. These status codes thus confirm our suspicions that the greedy runtimes for the hd_30-6g-2cg-3d and hd_30-6g-2cg-6d experiments were especially high because Gurobi had exhausted the default time limit while searching for optimal
In Figure 4.3, we again plot the percentage of possible links formed against the number of sessions generated thus far for each algorithm.

Despite timing out after 4 and 6 sessions of hd_30-6g-2cg-3d and hd_30-6g-2cg-6d, it is clear across all experiments that the greedy algorithm has generated significantly more links at each checkpoint than GroupSelect has. The performance of the greedy algorithm thus appears to be robust to timeouts that may arise due to the nature of the optimization problem in assemblies with clustered participants. As in the experiments on toy_20 without clustered participants (4.4.1), the gap between the percentage of possible links generated by the two algorithms also grows as the number of groups increases from 3 to 6. This suggests that as the size of the solution space grows, GroupSelect experiences more difficulty generating new links among hd_30 participants than the greedy algorithm does. Furthermore, across all experiments, the percentage of possible links generated by each
algorithm exhibits logarithmic growth, and the difference between the percentage of possible links generated by each algorithm only continues to grow with the number of sessions. There thus appears to be a fundamental gap in the number of links the two algorithms are able to achieve: Even if we were to continue running the GroupSelect algorithm on \textit{hd\_30}, it would likely not be able to generate all of the links that the greedy algorithm does.

The \textit{hd\_30}-3g-1cg-6d and \textit{hd\_30}-6g-2cg-6d experiments in Figure 4.3 also highlight a major flaw in the GroupSelect algorithm: The percentage of possible links generated over all sessions is constant with GroupSelect but continues to grow with the greedy algorithm. As described in Subsection 1.3.1, this behavior can ultimately be attributed to GroupSelect’s constant distribution heuristic for generating new allocations, which essentially treats participants with identical attribute vectors as interchangeable. Thus, while simple and efficient, this heuristic ultimately causes issues in practice. Because the \textit{hd\_30}-3g-1cg-6d and \textit{hd\_30}-6g-2cg-6d experiments involve 6 diversified attributes, nearly every participant’s attribute vector is unique, and GroupSelect is not able to generate any new links beyond the first 2 allocations.

Finally, we quantify the experiences of clustered participants in the \textit{hd\_30}-3g-1cg-3d and \textit{hd\_30}-3g-1cg-3d experiments (i.e., ones with nonconstant GroupSelect allocations). From Table 4.12, we can see for both experiments that the greedy ratio between the average number of links formed by clustered and non-clustered participants is significantly larger than the respective GroupSelect ratio, indicating that the greedy algorithm does a better job of helping \textit{hd\_30}'s 6 clustered participants meet new people, while the metrics listed in Table 4.13 exhibit the same trends as the ones listed in Table 4.8 do.

<table>
<thead>
<tr>
<th></th>
<th>Experiment</th>
<th>Avg. Links / CP</th>
<th>Avg. Links / NCP</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>3g-1cg-3d</td>
<td>19</td>
<td>26.417</td>
<td>0.719</td>
</tr>
<tr>
<td>GS</td>
<td>3g-1cg-3d</td>
<td>15</td>
<td>22.917</td>
<td>0.655</td>
</tr>
<tr>
<td>Greedy</td>
<td>6g-2cg-3d</td>
<td>26.5</td>
<td>27.458</td>
<td>0.965</td>
</tr>
<tr>
<td>GS</td>
<td>6g-2cg-3d</td>
<td>10.667</td>
<td>15.167</td>
<td>0.703</td>
</tr>
</tbody>
</table>

*Table 4.12: \textit{hd\_30} with CP – Ratio of Average Links Formed by Clustered and Non-Clustered Participants*
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>3g-1cg-3d</td>
<td>10</td>
<td>1.667</td>
<td>3.478</td>
</tr>
<tr>
<td>GS</td>
<td>3g-1cg-3d</td>
<td>10</td>
<td>1.667</td>
<td>3.478</td>
</tr>
<tr>
<td>Greedy</td>
<td>6g-2cg-3d</td>
<td>12</td>
<td>2.5</td>
<td>4.565</td>
</tr>
<tr>
<td>GS</td>
<td>6g-2cg-3d</td>
<td>14</td>
<td>2.083</td>
<td>4.674</td>
</tr>
</tbody>
</table>

Table 4.13: hd_30 with CP – Average Meetings per Type of Participant Pairing

4.4.3 Discussion

Overall, we can see across all experiments with clustered participants that the greedy algorithm consistently generates at least as many links as GroupSelect does; in fact, across all experiments but toy_20-2g-1cg, the greedy algorithm actually generates strictly more links than GroupSelect does. As the size of the solution space grows, the quality (measured as the percentage of possible links formed) of greedy solutions is also rather consistent, while the quality of GroupSelect solutions declines more rapidly. From our experiments, we hypothesize that this gap in the performances of the two algorithms can ultimately be attributed to GroupSelect’s constant distribution heuristic, which often struggles to generate unique allocations as problem complexity increases. The greedy algorithm thus appears to be more robust to problem complexity than GroupSelect is, and we hypothesize that the former will be more versatile in practice for the variety of use cases that assembly organizers may need such an algorithm for.

Furthermore, the differences in the performances of the two algorithms are still apparent across all experiments despite Gurobi timing out before optimal solutions can be found after the first few sessions of the hd_30 (with clustered participants) experiments. It thus appears that as long as Gurobi is able to find at least one solution per session (even if not optimal), the greedy algorithm’s performance will still be quite good against GroupSelect’s. Moreover, as finding at least one solution per session is a much easier problem than finding the optimal solution (e.g., any naive allocation, such as one generated by GroupSelect, will do), the greedy algorithm’s performance in these experiments is quite promising despite the NP-completeness of integer programming.

Between the two algorithms, we also see a difference in the experiences of clustered participants. In particular,
the greedy algorithm consistently does a better job of generating allocations that help clustered participants meet new people. This is desirable, as in general, we would like for a clustered participant’s experience to be as much like a non-clustered participant’s experience as possible (i.e., in that they meet approximately the same number of people over all sessions) despite the clustered participant’s restricted seating assignments. The greedy algorithm thus does a better job of delivering on this ideal.

Finally, we noted earlier that the greedy algorithm actually allows organizers to specify the number of clustered tables, while GroupSelect does not. With more clustered tables and hence fewer restrictions on clustered participants, however, it is clear that the greedy algorithm would generate even more links than it already does; in other words, the gap in the performances of the two algorithms would only continue to grow. By giving organizers this flexibility, the greedy algorithm thus further differentiates itself from GroupSelect.

### 4.5 Experiments without Clustered Participants

We now examine how the two algorithms perform on experiments without any clustered participants.

#### 4.5.1 toy_20 (without Clustered Participants)

We first ran experiments on the toy_20 assembly with both diversifiable attributes (i.e., Gender and Education Level) diversified, varying the number of groups.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>( n_G )</th>
<th>( n_S )</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_20-2g</td>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>toy_20-4g</td>
<td>4</td>
<td>14</td>
</tr>
<tr>
<td>toy_20-6g</td>
<td>6</td>
<td>24</td>
</tr>
</tbody>
</table>

*Table 4.14: toy_20 without CP - Experiments*

In Table 4.15, we list the average runtime per session for each experiment and see the same trends as in the experiments on toy_20 with clustered participants: Both algorithms are able to generate allocations extremely
quickly, and it is also clear across all of these experiments that the greedy algorithm is able to find optimal allocations well ahead of the default time limit.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Greedy – Avg. Runtime / Session (s)</th>
<th>GS – Avg. Runtime / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_20-2g</td>
<td>1.122 ± 0.138</td>
<td>0.016 ± 0.002</td>
</tr>
<tr>
<td>toy_20-4g</td>
<td>3.708 ± 2.318</td>
<td>0.047 ± 0.065</td>
</tr>
<tr>
<td>toy_20-6g</td>
<td>0.942 ± 0.410</td>
<td>0.007 ± 0.001</td>
</tr>
</tbody>
</table>

Table 4.15: toy_20 without CP – Average Runtimes per Session

In Figure 4.4, we again plot the percentage of possible links formed against the number of sessions generated thus far for each algorithm.

Figure 4.4: toy_20 without CP – Percentage of Possible Links Formed vs. Session

As in the experiments on toy_20 with clustered participants (4.4.1), it is clear for toy_20-4g and toy_20-6g that the greedy algorithm has generated significantly more links at each checkpoint than GroupSelect has, while for toy_20-2g-1cg, the difference between the percentage of possible links formed by each algorithm at
each checkpoint is not statistically significant. It is thus clear from the experiments in Figure 4.4 that the greedy algorithm generally does a better job of generating new links among toy_20 participants than GroupSelect does.

Comparing each of these experiments to their counterparts with clustered participants (4.4.1), we can also see that both algorithms are able to generate more links without clustered participants. This makes sense, as clustered participants have restricted seating assignments, while non-clustered participants do not. Eliminating these restrictions on clustered participants (i.e., by having no clustered participants at all) thus gives both algorithms more freedom to assign all participants as they see fit. Moreover, even without clustered participants, we can see that the same trend regarding the gap between the percentage of possible links generated by the two algorithms holds: As the size of the solution space grows, GroupSelect experiences more difficulty generating new links than the greedy algorithm does. This trend continues to hold for the remaining experiments (toy_100 without clustered participants and hd_30 without clustered participants) as well.

4.5.2 toy_100 (without Clustered Participants)

Next, we ran experiments on the toy_100 assembly with all 4 attributes (i.e., Gender, Employment Status, State, and Education Level) diversified, varying the number of groups.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>$n_G$</th>
<th>$n_S$</th>
<th>Time Limit / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_100-8g</td>
<td>8</td>
<td>24</td>
<td>60</td>
</tr>
<tr>
<td>toy_100-16g</td>
<td>16</td>
<td>30</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 4.16: toy_100 without CP - Experiments

Note that for the toy_100-8g and toy_100-16g experiments, we extended the time limit per session from the default of 30s to 60s and 300s respectively; due to the larger solution space with 100 participants and up to 16 groups, 30s was not long enough for Gurobi to find solutions (even if not optimal) for up to 30 sessions. In practice, it is unlikely that organizers will need to generate as many as 30 sessions (and for an assembly as large as 100 participants), but we demonstrate here that the greedy algorithm is still able to handle these more extreme use

\[\text{More specifically, with 30s per session, Gurobi was able to find solutions for up to } \approx 12 \text{ out of 24 and } \approx 20 \text{ out of 30 sessions of toy_100-8g and toy_100-16 respectively before it timed out with no solution afterwards.}\]
cases well. Furthermore, while simply extending the default time limit was sufficient to preempt “no solution” scenarios in these experiments, we also propose other ways to preempt this “no solution” scenario in our Discussion (4.5.4) and Future Work (6.2.1).

In Table 4.17, we can see that GroupSelect’s runtimes continue to be lower than the greedy algorithm’s. For both experiments, the average runtimes for the greedy algorithm also begin to approach the time limits listed in Table 4.16, which indicates that Gurobi is timing out before optimal allocations can be found.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Greedy – Avg. Runtime / Session (s)</th>
<th>GS – Avg. Runtime / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_100-8g</td>
<td>58.868 ± 0.0418</td>
<td>0.050 ± 0</td>
</tr>
<tr>
<td>toy_100-16g</td>
<td>246.155 ± 3.715</td>
<td>0.055 ± 0.002</td>
</tr>
</tbody>
</table>

Table 4.17: toy_100 without CP – Average Runtimes per Session

Indeed, as noted in Subsection 4.4.2, some of our ILPs for these experiments cannot be solved to provable optimality within a reasonable time limit. In Table 4.18, we list Gurobi’s optimization status codes for each experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Status Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>toy_100-8g</td>
<td>2 for session 1, 9’s thereafter</td>
</tr>
<tr>
<td>toy_100-16g</td>
<td>2’s for sessions 1-6, 9’s thereafter</td>
</tr>
</tbody>
</table>

Table 4.18: toy_100 without CP – Status Codes

In Figure 4.5, we again see similar trends as in the previous experiments: Despite timing out after 1 session and 6 sessions of the toy_100-8g and toy_100-16g experiments respectively, the greedy algorithm has still generated significantly more links at each checkpoint than GroupSelect has. The performance of the greedy algorithm thus also appears to be robust to timeouts that may arise due to the nature of the optimization problem in assemblies without clustered participants.
In Figure 4.6, we compare the meetings distributions of the two algorithms after 30 sessions of \textit{toy\_100-16g}.

![Figure 4.6: toy\_100-16g - 30-Session Meetings Distributions](image)

<table>
<thead>
<tr>
<th></th>
<th>Avg. Mtgs. / Pair</th>
<th>Max Mtgs. / Pair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Greedy</td>
<td>1.791 ± 0.022</td>
<td>5.333 ± 0.577</td>
</tr>
<tr>
<td>GS</td>
<td>3.464 ± 0.387</td>
<td>30 ± 0</td>
</tr>
</tbody>
</table>

Table 4.19: toy\_100-16g - 30-Session Nonzero Meetings Distribution Measures

Interestingly, the GroupSelect distribution is strongly right-skewed, with a long tail of pairs that have met up to every session out of the 30 sessions, while the greedy distribution has a much shorter tail, with no pairs that have met more than 6 out of the 30 sessions. Ignoring pairs that have not yet met (i.e., only considering pairs with a nonzero number of meetings), Table 4.19 further confirms that pairs of participants have met significantly more
often on average across GroupSelect allocations than greedy allocations, and that the variance in the number of
meetings pairs have had across the 30 sessions is significantly higher for the former than the latter. These stark
differences are a direct result of the greedy algorithm’s objective, which, for each session, not only prioritizes links
but also aims to reduce the overall cost of meetings. Thus, even after 30 sessions, we see that the greedy algorithm
does a much better job of not only generating new links but also ensuring that participants are not meeting one
another more often than is necessary.

4.5.3 **hd_30 (without Clustered Participants)**

Finally, we ran experiments on the hd_30 assembly, varying the number of groups and diversified attributes.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>$n_G$</th>
<th>$n_D$</th>
<th>$n_S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd_30-3g-3d</td>
<td>3</td>
<td>(b, d, g)</td>
<td>12</td>
</tr>
<tr>
<td>hd_30-6g-3d</td>
<td>6</td>
<td>(b, d, g)</td>
<td>25</td>
</tr>
<tr>
<td>hd_30-3g-6d</td>
<td>3</td>
<td>(a, b, c, d, e, g)</td>
<td>14</td>
</tr>
<tr>
<td>hd_30-6g-6d</td>
<td>6</td>
<td>(a, b, c, d, e, g)</td>
<td>20</td>
</tr>
</tbody>
</table>

*Table 4.20: hd_30 without CP – Experiments*

In Table 4.21, we list the average runtime per session for each experiment, and in Table 4.22, we list Gurobi’s
optimization status codes for each experiment.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Greedy – Avg. Runtime / Session (s)</th>
<th>GS – Avg. Runtime / Session (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd_30-3g-3d</td>
<td>27.097</td>
<td><strong>0.015</strong></td>
</tr>
<tr>
<td>hd_30-6g-3d</td>
<td>27.057</td>
<td><strong>0.01</strong></td>
</tr>
<tr>
<td>hd_30-3g-6d</td>
<td>12.065</td>
<td><strong>0.089</strong></td>
</tr>
<tr>
<td>hd_30-6g-6d</td>
<td>27.158</td>
<td><strong>0.063</strong></td>
</tr>
</tbody>
</table>

*Table 4.21: hd_30 without CP – Average Runtimes per Session*
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Status Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>hd_30-3g-3d</td>
<td>2's for session 1, 9's thereafter</td>
</tr>
<tr>
<td>hd_30-6g-3d</td>
<td>2's for sessions 1-3, 9's thereafter</td>
</tr>
<tr>
<td>hd_30-3g-6d</td>
<td>2's for all sessions</td>
</tr>
<tr>
<td>hd_30-6g-6d</td>
<td>2's for sessions 1-2, 9's thereafter</td>
</tr>
</tbody>
</table>

Table 4.22: hd_30 without CP – Status Codes

In Figure 4.7, we again see that despite timing out after 1 session, 3 sessions, and 2 sessions for the hd_30-3g-3d, hd_30-6g-3d, and hd_30-6g-6d experiments respectively, the greedy algorithm has generated significantly more links at each checkpoint than GroupSelect has. Moreover, despite the early termination of many of the hd_30-3g-3d and hd_30-6g-3d sessions, the greedy algorithm has still generated nearly all possible links by the final session of each experiment. The performance of the greedy algorithm thus again appears to be robust to timeouts that may arise due to the nature of the optimization problem in assemblies without clustered participants. Finally, comparing each of these experiments to their counterparts on hd_30 with clustered participants (4.4.2) (as we did with the toy_20 experiments), we again see that both algorithms are able to generate more links without clustered participants.
4.5.4 Discussion

Overall, we can see across all experiments without clustered participants that the greedy algorithm generally does a better job of generating new links than GroupSelect does; the gap in the performances of the two algorithms as problem complexity increases and the quality of GroupSelect’s solutions declines is also evident. Moreover, as in the experiments with clustered participants, the differences in the performances of the two algorithms are still apparent across all experiments despite Gurobi timing out before optimal solutions can be found after the first few sessions of the toy_100 and hd_30 (without clustered participants) experiments. The greedy algorithm’s performance in experiments both with and without clustered participants thus appears to be robust to these time-outs and is quite promising despite the NP-completeness of integer programming. Furthermore, we see that the greedy algorithm does a much better job of not only generating new links but also ensuring that participants are
not meeting one another more often than is necessary. The greedy algorithm’s behavior thus serves participants well, as they get to interact with many new people while minimizing the likelihood that they see the same peers over and over.

As we continue to work with the Sortition Foundation to test and ultimately deploy the greedy algorithm, we plan to make the switch from Gurobi (a commercial optimizer) to an open-source (and hence less state-of-the-art) optimizer. We thus wish to ensure that after the switch, the greedy algorithm is still robust to problem complexity and capable of finding at least one solution per session within a reasonable time limit. In particular, while simply extending the default time limit was sufficient to preempt “no solution” scenarios in the toy_100 experiments, it may not always be clear to us or the organizers how much additional time is needed without further trial and error. We thus propose another approach that may eliminate this issue entirely: We noted earlier that finding at least one solution per session should be relatively easy, and we also observed in the toy_100-8g and toy_100-16g experiments that with 30s per session, Gurobi was actually able to find solutions for up to \( \approx 12 \) and \( \approx 20 \) sessions respectively before it timed out with no solution afterwards. Because the greedy algorithm aims to reduce the overall cost of meetings, we thus hypothesize that Gurobi eventually timed out with no solution because the meeting history of the assembly (and hence the magnitude of the overall cost) had grown too large; indeed, this is a plausible theory, as a feasible solution for the first “no solution” session could simply have been the allocation for the previous session, which was found within 30s while the former was not. One way to preempt the “no solution” scenario is thus to partially “reset” the meeting history. For example, for each pair of participants \((i,j)\) where \(i,j \in [n_P]\) and \(i < j\), we could reset

\[
\text{meetings_record}[i,j] := \lceil \text{meetings_record}[i,j]/2 \rceil.
\]

This could be done every few sessions, or after a pair of participants reaches some threshold number of meetings.

Finally, we observe across all experiments that the percentage of possible links generated by each algorithm ex-
hibits logarithmic growth. While the percentage for the greedy algorithm approaches 1 (i.e., generating all possible links) in many of the experiments, however, there appears to be a limit that is less than 1 for other experiments. This makes sense, as with restrictions on both representation and clustered participants (if any are present), it may not actually be possible to achieve all possible links. We thus wish to refine our notion of the maximum possible number of links by searching for a theoretically motivated bound; in Chapter 5, which covers our Theoretical Results, we begin to explore this very idea.
The work presented in this chapter is the result of collaboration with Jake Barrett, a PhD student at the University of Edinburgh, and Paul Gölz, a PhD student at Carnegie Mellon University.

In this chapter, we derive a theoretical bound on the minimum possible number of repeated meetings that can be achieved between arbitrary back-to-back sessions in assemblies without clustered participants; as non-repeated meetings are simply links, this result also yields a bound on the maximum possible number of links that can be achieved between such sessions.
5.1 Definitions & Assumptions

As in our model (2.2), suppose we have a set of \( n_P \) participants \([n_P] = \{0, \ldots, n_P - 1\}\) and a set of \( n_G \) groups \([n_G] = \{0, \ldots, n_G - 1\}\), and that participants are distributed among groups as evenly as possible (i.e., according to the size function \( Z \) defined in Subsection 2.2.3). We consider arbitrary back-to-back sessions \( s, t \in [n_S] \) where \( t = s + 1 \), so for convenience, let \( s = 1 \) and \( t = 2 \). Note that we only consider assemblies without clustered participants, and that the bound we derive does not take into account representation constraints (i.e., groups do not need to meet any specific representation quotas), so there is no need to associate attributes with each participant.

Now, for an arbitrary allocation \( A \), let \( A \) be a function such that for a session \( s \in [n_S] \) and any participant \( i \in [n_P] \), \( A_s(i) \) returns the group that \( i \) has been assigned to for \( s \). For each pair of participants \((i, j)\) where \( i, j \in [n_P] \) and \( i < j \) and sessions \( s, t \in [n_S] \) where \( s < t \), we say that \((i, j)\) is a repeated meeting if the pair meets during both sessions; in other words, we can define an indicator \( I_{s,t}(i, j) \) that corresponds to whether \((i, j)\) is a repeated meeting as follows:

\[
I_{s,t}(i, j) = \begin{cases} 
1 & \text{if } (A_s(i) = A_s(j)) \land (A_t(i) = A_t(j)) \\
0 & \text{otherwise}
\end{cases}
\]

5.2 Objective

For an arbitrary allocation \( A \) over the back-to-back sessions \( s \) and \( t \), the number of repeated meetings generated between the two sessions can be expressed as

\[
RM_{s,t}(A) = \sum_{i,j \in [n_S] \text{ s.t. } i < j} I_{s,t}(i, j).
\]

We thus aim to minimize \( RM_{s,t}(A) \) over all feasible allocations \( A \).
5.3 Theorem (Minimum Repeated Meetings)

The minimum number of repeated meetings $RM_{s,t}^*$ that can be achieved between back-to-back sessions in assemblies without clustered participants is

$$RM_{s,t}^* = \begin{cases} n_G^* \binom{k}{2} + rk & \text{if } n_P > n_G^* \\ 0 & \text{otherwise} \end{cases}$$

(5.1)

where $k = \lfloor n_P/n_G^* \rfloor$ and $r = n_P \mod n_G^*$.

5.3.1 Proof

For convenience, we will simply refer to participants assigned to a group $g \in [n_G]$ in session 1 as “$g_1$-participants.”

Consider an arbitrary allocation for session 1. We first prove the following lemma:

Lemma 5.3.1. For any $g \in [n_G]$, distributing the $Z(g)$ $g_1$-participants as evenly as possible among all $n_G$ groups in session 2 minimizes the number of repeated meetings between $g_1$-participants.

Proof. Fix an arbitrary group $g \in [n_G]$. For an arbitrary session-2 allocation $A$ of the $Z(g)$ $g_1$-participants, we can adapt the notation from Section 5.1 and let $A(i)$ denote the group that the $g_1$-participant $i$ has been assigned to in $A$. Then, for any group $b \in [n_G]$, $A^{-1}(b)$ denotes the set of $g_1$-participants that were assigned to $b$ in $A$.

First, consider any session-2 allocation $A$ of the $g_1$-participants that contains two groups $x, x' \in [n_G]$ such that $|A^{-1}(x)| \geq |A^{-1}(x')| + 2$. We claim that $A$ does not minimize the number of repeated meetings between $g_1$-participants; in particular, we can construct another allocation $B$ from $A$ that results in fewer such repeated meetings: Let $B$ be identical to $A$, except with one of the $A^{-1}(x)$ participants reassigned to $x'$. Then, by defini-
tion, the number of repeated meetings between the \( g_1 \)-participants in \( A \) is

\[
RM_{1,2}(A) = \sum_{h \in [n_G]} \left( \frac{|A^{-1}(h)|}{2} \right),
\]

while the number of repeated meetings between the \( g_1 \)-participants in \( B \) is

\[
RM_{1,2}(B) = \sum_{h \in [n_G]} \left( \frac{|B^{-1}(h)|}{2} \right)
= \left( \frac{|A^{-1}(x)|}{2} - 1 \right) + \left( \frac{|A^{-1}(x')|}{2} + 1 \right) + \sum_{h \in [n_G] \backslash \{x, x'\}} \left( \frac{|A^{-1}(h)|}{2} \right).
\]

Rewriting \( RM_{1,2}(A) \) as

\[
\left( \frac{|A^{-1}(x)|}{2} \right) + \left( \frac{|A^{-1}(x')|}{2} \right) + \sum_{h \in [n_G] \backslash \{x, x'\}} \left( \frac{|A^{-1}(h)|}{2} \right),
\]

we can then compute \( RM_{1,2}(A) - RM_{1,2}(B) \) as follows:

\[
RM_{1,2}(A) - RM_{1,2}(B) = \left( \frac{|A^{-1}(x)|}{2} \right) + \left( \frac{|A^{-1}(x')|}{2} \right) - \left( \frac{|A^{-1}(x)|}{2} - 1 \right) - \left( \frac{|A^{-1}(x')|}{2} + 1 \right)
= \frac{|A^{-1}(x)|(|A^{-1}(x)| - 1)}{2} + \frac{|A^{-1}(x')|(|A^{-1}(x')| - 1)}{2}
- \frac{|A^{-1}(x)| - 1}{2} \frac{|A^{-1}(x')| - 2}{2} - \frac{|A^{-1}(x')|(|A^{-1}(x')| + 1)}{2}
= |A^{-1}(x)| - |A^{-1}(x')| - 1
\geq (|A^{-1}(x')| + 2) - |A^{-1}(x')| - 1 = 1.
\]

We have thus shown that \( RM_{1,2}(A) - RM_{1,2}(B) > 0 \iff RM_{1,2}(B) < RM_{1,2}(A) \), and \( A \) does not minimize the number of repeated meetings between \( g_1 \)-participants as desired.

Now, consider any session-2 allocation \( A \) of the \( g_1 \)-participants that does not contain two groups \( x, x' \in [n_G] \)
such that $|A^{-1}(x)| \geq |A^{-1}(x')| + 2$, i.e., $A$ such that for all $x \in [n]$, the $|A^{-1}(x)|$ differ by at most 1. We claim that all such allocations achieve the same number of repeated meetings between the $g_i$-participants: Let $A$ be such an allocation, and for convenience, let $k_g = \lfloor Z(g)/n_G \rfloor$ and $r_g = Z(g) \mod n_G$. Then, in order for the condition to hold, it is easy to verify that $A$ must assign exactly $(k_g + 1)g_1$-participants to $r_g$ of the $n_G$ groups and $k_g g_1$-participants to the remaining $n_G - r_g$ groups. The number of repeated meetings between $g_1$-participants in $A$ is thus always

$$RM_{1,2}(A) = r_g(k_g + 1) + (n_G - r_g)(\frac{k_g}{2})$$

$$= r_g \frac{k_g(k_g + 1)}{2} + \frac{(n_G - r_g)(k_g(k_g - 1))}{2}$$

$$= n_G(\frac{k_g}{2}) + r_g k_g.$$  

(5.2)

Putting everything together, we can see that in order for a session-2 allocation to minimize the number of repeated meetings between $g_1$-participants, it must distribute the $g_1$-participants as evenly as possible among all $n_G$ groups. But since all allocations that distribute the $g_1$-participants as evenly as possible achieve the same number of repeated meetings between $g_1$-participants, the allocations that minimize this number are exactly the allocations that distribute the $g_1$-participants as evenly as possible among all $n_G$ groups. The statement of the lemma thus holds as desired.

Corollary 5.3.1.1. For each $g \in [n_G]$, distributing the $g_i$-participants as evenly as possible among all $n_G$ groups in session 2 minimizes the total number of repeated meetings between the two sessions.

Proof. By Lemma 5.3.1, it follows that for each $g \in [n_G]$, distributing the $Z(g)g_i$-participants as evenly as possible among all $n_G$ groups in session 2 minimizes the number of repeated meetings between $g_i$-participants. But because repeated meetings can only form between participants who were in the same group in session 1, distributing all groups in such a way in session 2 actually minimizes the total number of repeated meetings between the two
sessions. We can thus compute $RM^*_{1,2}$ using Equation 5.2 as follows:

$$RM^*_{1,2} = \sum_{g \in [n_G]} n_G \binom{k_g}{2} + r_g k_g. \quad (5.3)$$

Moreover, it is clear that this minimum is actually attainable: Algorithm 3 generates a session-2 allocation that for each $g \in [n_G]$ distributes the $g_i$-participants as evenly as possible among all $n_G$ groups; it also guarantees that the right number of participants are assigned to each session-2 group (i.e., according to the size function $Z$ defined in Subsection 2.2.3), thus ensuring that the allocation is feasible.

Algorithm 3 Corollary 5.3.1.1 – Sample Assignment Strategy

1: function ASSIGN(session_1_groups)
2:   ▷ session_1_groups is a map from each $g \in n_G$ to the set of $g_i$-participants.
3:   counter ← 0
4:   for session_1_group in session_1_groups do
5:     for participant in session_1_group do
6:       ▷ Cycle through the $n_G$ groups.
7:       session_2_group ← (counter mod $n_G$)
8:       Assign participant to session_2_group
9:       counter += 1
10:   end for
11: end for
12: end function

Returning to the proof of Theorem 5.3, we now show how Equation 5.3 can be rewritten as Equation 5.1. We split the proof into three cases based on the relationship between $n_F$ and $n_G^2$. 
Case 1 \((n_P = n_G^2)\)

We claim that \(RM_{1,2}^* = 0\): Since \(n_P = n_G^2 \equiv 0 \mod n_G\), \(Z(g) = \lfloor n_G^2/n_G \rfloor = n_G\) for all \(g \in [n_G]\). But then \(k_g = \lfloor n_G/n_G \rfloor = 1\) and \(n_G \equiv 0 \mod n_G \implies r_g = 0\) for all \(g \in [n_G]\), so it follows from Equation 5.3 that

\[
RM_{1,2}^* = \sum_{g \in [n_G]} n_G \binom{1}{2} + 0 \cdot 1 = \sum_{g \in [n_G]} 0 = 0.
\]

Intuitively, this makes sense: Suppose we fix an arbitrary group \(g \in [n_G]\). In order to achieve 0 repeated meetings between \(g\)-participants, the \(n_G\)-participants must be assigned to distinct groups (which may include \(g\)) in session 2; otherwise, at least one pair among the \(g\)-participants will be meeting for the second time. But this is clearly possible, as there are \(n_G\) total groups, and we can simply assign one of the \(g\)-participants to each group. Moreover, since \(Z(g) = n_G\) for all \(g \in [n_G]\), it follows by symmetry that all \(n_G^2\) participants can be reallocated such that there are 0 repeated meetings between sessions 1 and 2; one such allocation that attains this minimum, for example, is generated by Algorithm 3.

Case 2 \((n_P < n_G^2)\)

We again claim that \(RM_{1,2}^* = 0\): By definition, \(Z(g) \leq n_G\) for all \(g \in [n_G]\). It thus follows that \(k_g \leq \lfloor n_G/n_G \rfloor = 1 \implies \binom{k_g}{2} = 0\) for all \(g \in [n_G]\). Now, consider an arbitrary group \(g \in [n_G]\). If \(k_g = 0\), it immediately follows that \(r_g k_g = 0\). Otherwise, if \(k_g = 1\), we must have that \(Z(g) = n_G\). Thus, \(n_G \equiv 0 \mod n_G \implies r_g = 0\), and we
again have that \( r_gk_g = 0 \). Putting everything together, it follows from Equation 5.3 that

\[
RM^*_{1,2} = \sum_{g \in [n_G]} n_G \left( \frac{k_g}{2} \right) + r_gk_g
\]

\[
= \sum_{g \in [n_G]} 0
\]

\[
= 0.
\]

Moreover, as in Case 1, one such allocation that attains this minimum is generated by Algorithm 3.

**Case 3 \((n_P > n_G)\)**

Finally, in the case that \( n_P > n_G \), we use Algorithm 3 to show how Equation 5.3 can be rewritten as Equation 5.1. In particular, we already know from Corollary 5.3.1.1 that Algorithm 3 generates a session-2 allocation \( A \) that minimizes the total number of repeated meetings between sessions 1 and 2, so we can simply compute \( RM^*_{1,2} \) by counting the total number of repeated meetings between the session-1 allocation and \( A \). We split Case 3 into two subcases.

**Subcase 3-1:** This subcase will provide intuition for the more general subcase to follow. Consider when \( n_P = kn_G^2 \) for \( k \in \mathbb{N} \) such that \( k > 1 \). By definition, \( Z(g) = kn_G \) for all \( g \in [n_G] \). Now, suppose without loss of generality that for each \( g \in [n_G] \), the set of \( g \)-participants is labeled \( g_i = \{0, 1, \ldots, Z(g) - 1\} \). In order to count the number of repeated meetings between the session-1 allocation and \( A \), we can first imagine dividing the \( n_P \) participants into \( n_P/n_G^2 = k \) “blocks” of size \( n_G^2 \) such that block 1 contains the set \([n_G]\) \( g_1 \) (i.e., the first \( n_G \) \( g_1 \)-participants) for all \( g \in [n_G] \), block 2 contains the set \([2n_G]\) \( [n_G]\) \( g_1 \) (i.e., the next \( n_G \) \( g_1 \)-participants) for all
$g \in [n_G]$, and so on. More explicitly, the $k$ blocks $b_1, b_2, \ldots, b_k$ are

\begin{align*}
b_1 & : \{0, 1, \ldots, n_G - 1, 0, 1, \ldots, n_G - 1, \ldots, 0, 1, \ldots, n_G - 1\}, \\
b_2 & : \{(n_G, n_G + 1, \ldots, 2n_G - 1, n_G, n_G + 1, \ldots, 2n_G - 1, \ldots, n_G, n_G + 1, \ldots, 2n_G - 1)\}, \\
& \vdots \\
b_k & : \{(k - 1)n_G, (k - 1)n_G + 1, \ldots, kn_G - 1, (k - 1)n_G, (k - 1)n_G + 1, \ldots, kn_G - 1, \ldots, (n_G - 1)n_G + 1, \ldots, (n_G - 1)n_G - 1\}.
\end{align*}

This formulation simplifies our counting process. In particular, we can consider how Algorithm 3 assigns the participants in each block to groups as follows: Fix an arbitrary group $g \in [n_G]$. For each block, it is clear that Algorithm 3 assigns exactly one of the $n_G g_1$-participants to each of the $n_G$ groups in $A$. Then, since there are $k$ total blocks, Algorithm 3 must assign exactly $k g_1$-participants to each of the $n_G$ groups in $A$. We can thus compute $R_{1,2}$ as follows: Because any pair of these $k g_1$-participants (which we henceforth refer to as a “$g_1$-clique”) has already met in session 1, they generate $\binom{k}{2}$ repeated meetings between the two sessions. Then, since there are $n_G$ of these $g_1$-cliques (where $g_1$ is fixed), and $n_G$ possible values for $g_1$, it follows that

\[ R_{1,2} = n_G^2 \binom{k}{2}. \]

Moreover, since $kn_G \equiv 0 \mod n_G^2 \implies r = 0 \implies rk = 0$, this result matches the one given in Equation 5.1.

**Subcase 3-2:** Now, consider when $n_P = kn_G + r$ for $k, r \in \mathbb{N}$ such that $k > 0$ and $0 < r < n_G^2$. In order to count the number of repeated meetings between the session-1 allocation and $A$, we can again imagine dividing the $n_P$ participants into blocks. In particular, we already know from Subcase 3-1 that the first $\left\lceil \frac{n_P}{n_G^2} \right\rceil = k$ blocks
of size $n_G^2$ contribute $n_G^2 \binom{k}{2}$ repeated meetings to the total count. In this subcase, however, we also have an additional “remainder” block of size $r = n_P \mod n_G^2$. We can again consider how Algorithm 3 assigns the participants in this remainder block to groups as follows: First, because participants are distributed among groups as evenly as possible in all sessions (and, in particular, in session 1), the remainder block consists of at most $n_G g_1$-participants for all $g \in [n_G]$. It is thus clear that for each $g \in [n_G]$, Algorithm 3 assigns at most one of the $g_1$-participants to each of the $n_G$ groups in $A$. Then, since there are already $k g_1$-participants in each group, assigning any participant in the remainder block to a group increases the number of repeated meetings by $k$. The remainder block thus contributes $rk$ repeated meetings to the total count, and it follows that

$$RM^*_{1,2} = n_G^2 \binom{k}{2} + rk.$$ 

The statement of the theorem thus holds as desired.

5.3.2 Corollary (Maximum Links)

Finally, as we observed at the beginning of the chapter, the bound on the minimum number of repeated meetings that can be achieved between back-to-back sessions in assemblies without clustered participants also yields a bound on the maximum number of links $L^*_{s,t}$ that can be achieved between such sessions. In particular, the naive maximum (i.e., all possible meetings that can generated between two sessions) is $m = 2 \sum_{g \in [n_G]} \binom{Z(g)}{2}$, so adjusting for the minimum repeated meetings yields

$$L^*_{s,t} = \begin{cases} 
  m - \left( n_G^2 \binom{k}{2} + rk \right) & \text{if } n_P > n_G^2, \\
  m & \text{otherwise}
\end{cases},$$

(5.4)

where $k = \lfloor n_P / n_G^2 \rfloor$ and $r = n_P \mod n_G^2$.  

64
In this final chapter, we comment on the advantages and disadvantages of the two algorithms, summarize our contributions, and suggest next steps for future research.

6.1 Discussion

6.1.1 Motivating Questions, Revisited

We first revisit the Motivating Questions (1.2.2) that led us to develop the greedy algorithm. In particular, we discuss the advantages and disadvantages of the two algorithms by considering algorithmic properties like efficiency, versatility, and ease of use, as well as what algorithmic and theoretical guarantees each algorithm is able to provide.
Efficiency

Empirically, it is clear from running the two algorithms that GroupSelect generates allocations extremely quickly, and for the most part more quickly than the greedy algorithm does. As the complexity of the assembly partitioning problem increases, however, it is also clear that GroupSelect fails to generate more than a few (and sometimes even one) unique allocations, which not only has a negative impact on how many links can be formed but also plays a major role in how quickly the algorithm is able to generate allocations. Thus, despite the efficiency of GroupSelect, the quality of GroupSelect allocations tends to be quite low compared to that of greedy allocations. It is therefore easy to advocate that organizers adopt the greedy algorithm: For simple use cases, the greedy algorithm generates allocations nearly as quickly as GroupSelect does. For more complex use cases, the greedy algorithm generates significantly higher-quality allocations, and within a reasonable amount of time. In the hd_30-6g-6d experiment on real-world data, for example, allocations for 20 sessions could be generated in at most 10 minutes. In the toy_100-16g experiments with the extended time limit of 300s per session, allocations for 30 sessions could be generated in at most 2.5 hours. As it is rather reasonable to assume that organizers will run such an algorithm at least a few hours (and, more likely, at least a few days) in advance of the actual assembly, we believe that these runtimes do not pose any barriers to the adoption of the greedy algorithm. In our Future Work (6.2.1), however, we suggest potential ways to further improve its runtime.

Versatility

As problem complexity increased, we saw in Chapter 4 that GroupSelect struggled to generate unique allocations that would form new links, while the greedy algorithm was rather consistent and continued to generate high-quality allocations. If organizers were to run GroupSelect on the hd_30-1cg-6d experiment (for which GroupSelect is only able to generate one unique allocation), for example, they would likely have to resort to manually...
generating allocations themselves. The greedy algorithm thus appears to be more versatile in practice for the variety of use cases that organizers may need such an algorithm for.

We also noted in Subsection 4.4.3 that the greedy algorithm allows organizers to specify the number of clustered tables, while GroupSelect does not. As increasing this number places fewer restrictions on clustered participants, organizers can then strike their own balance between the number of clustered tables and how many links are generated. The choice of this number thus gives organizers even more flexibility to tailor their inputs to their particular assemblies and use cases. Overall, then, we see that the greedy algorithm is applicable to a wider variety of use cases than GroupSelect is.

Ease of Use

As described in Subsections 2.2.1 and 1.3.1 for the greedy algorithm and GroupSelect respectively, the inputs to the two algorithms are nearly identical, and it is likely from discussions with GroupSelect’s creator Philipp Verpoort that GroupSelect’s current interface can directly be adapted to build the greedy algorithm’s interface. As the current interface is rather intuitive and has already been used by organizers around the world to generate allocations, this continuity means that organizers both new and old will likely have little trouble running the greedy algorithm. Furthermore, it has been clear from conversations with Philipp that organizers typically wish they had more control over how GroupSelect generates allocations. Conveniently, the greedy algorithm not only allows organizers to specify the number of clustered tables but also gives them the power to more precisely control how each attribute value is represented on each table, and thus how diverse groups are. We acknowledge, however, that the ability to make these decisions can be a double-edged sword, as organizers may not initially know how loosely representation constraints should be defined so that the problem is still feasible. In our Future Work (6.2.1), we suggest potential ways to remedy this “burden of choice.”
Algorithmic and Theoretical Guarantees

We saw in Subsection 1.3.1 that GroupSelect fails to provide any algorithmic guarantees for attribute representation across tables. In particular, while organizers can specify an attribute ordering, they are not provided any guarantees with respect to whether each table still reflects the diversity of the assembly. The greedy algorithm, on the other hand, does provide such guarantees: As described in 2.2, the algorithm takes as input a representation function $R$ that constrains how each attribute value is represented on each table. This means that any allocations generated by the algorithm are guaranteed to satisfy these constraints, and organizers now have explicit guarantees with respect to how closely each table reflects the diversity of the assembly.

We also saw in Subsection 1.3.1 that GroupSelect fails to provide any algorithmic guarantees for the number of links generated over all sessions, instead relying on a heuristic. The greedy algorithm, on the other hand, does provide such guarantees: For each session, the algorithm explicitly minimizes a cost function that not only prioritizes links but also aims to reduce the overall cost of meetings. Organizers are thus now provided information regarding the optimality of the solution found for each session.

Finally, in Chapter 5, we derived a theoretical bound on the maximum possible number of links that can be achieved between back-to-back sessions of allocation in assemblies without clustered participants. In our Future Work (6.2.1), we propose additional theoretical questions that would be interesting to explore.

6.1.2 Additional Commentary

For all of our experiments in Chapter 4, we only generated allocations for runs in which the initial set of meetings between participants was the empty set, i.e., we assumed that none of the participants had met before session 1. Because our algorithm greedily constructs allocations by session, however, organizers also have the ability to run the algorithm with a nonempty initial set that takes any previous meetings into account. For example, if organizers wanted to manually generate the first $s$ sessions and then use the greedy algorithm to generate the remaining $t$ sessions, they could simply feed their first $s$ allocations to the algorithm, and it would automatically generate the
remaining \( t \) allocations as if it had generated the first \( s \) allocations itself. Another common use case for this feature could be if organizers generate allocations for \( s \) sessions and only later discover that they actually need allocations for \( t \) more sessions; in this case, they could again feed the first \( s \) allocations to the algorithm, and it would generate \( t \) more allocations that take all meetings from the first \( s \) sessions into account.

6.2 Conclusion

In this report, we introduced the assembly partitioning problem, in which we wish to partition participants among a set of groups over multiple sessions of deliberation. We provided an overview of the baseline algorithm—GroupSelect—that is being used by organizers to generate partitions, as well as its advantages and disadvantages. We then defined a model for the assembly partitioning problem and described our approach to developing the greedy algorithm. Afterwards, we conducted a series of experiments, both with and without clustered participants, to compare the performances of the greedy algorithm and GroupSelect. We also derived a theoretical bound on the maximum possible number of links that can be achieved between back-to-back sessions of allocation in assemblies without clustered participants. Finally, we discussed the advantages of the greedy algorithm over GroupSelect, responding to the original questions that had motivated our search for a better algorithm.

6.2.1 Future Work

There are several directions that can be explored in future work. First, in order to improve greedy algorithm runtimes and, in particular, preempt scenarios where Gurobi is not able to find at least one solution for each session, we may wish to partially reset the meeting history of the assembly after every few sessions, or after a pair of participants reaches some threshold number of meetings; this approach is described in more detail in Subsection 4.5.4. To improve the algorithm’s ease of use, further work can also be done to compute feasible representation constraints that are then “autofilled” for organizers. This would make the work of inputting these constraints both less overwhelming and easier for organizers to adapt to their particular use cases by giving them some sort of
baseline. If the algorithm computes the tightest feasible constraints, for example, organizers can simply decrease lower bounds or increase upper bounds to loosen certain constraints (e.g., ones for lower-priority attributes) and thereby generate more links. Finally, one helpful GroupSelect feature that we have yet to implement as part of the greedy algorithm is manual allocation. As explained in Section 2.1, manual allocation can be used to tackle special cases of clustering in which participants must be grouped on the same table (rather than on any of the clustered tables). As we can simply add constraints that assign these participants to the same table, this feature is relatively easy to integrate into the greedy algorithm, and we plan to do so before deploying the algorithm.

Further theoretical work would also be interesting to explore. Building on the bound we derived in Chapter 5, we are currently working on deriving the corresponding bound on the maximum number of links that can be achieved in assemblies with clustered participants. As clustered participants have restricted seating assignments, this case has proven to be a bit more involved. Beyond studying links, we may also wish to study the hardness of the initial ILP introduced in Subsection 3.2, which aimed to maximize the number of links generated over all sessions, in order to theoretically motivate the greedy algorithm. We may also wish to determine what sort of approximation the greedy algorithm produces. Finally, building on the intuition developed in the Motivating Example (1.2.1), it would be interesting to more precisely define the tradeoff between the strictness of representation on each table and the number of links that can be generated.

We are currently working with the Sortition Foundation to deploy the greedy algorithm. This process involves making the switch from Gurobi to an open-source optimizer, as well as building the greedy algorithm’s interface. After deploying the algorithm, we also plan to conduct various user studies involving both organizers and participants. More specifically, we hope to collect feedback from organizers on how easy the algorithm is to use, what features they like and dislike, and what features they believe the algorithm is still lacking. We also hope to better understand the experiences of participants as they interact with their peers and transition from session to session. Some interesting questions to explore here include whether participants perceive a difference in how often they meet new people when organizers generate allocations using the greedy algorithm rather than GroupSelect, and
whether the diversity of participants’ assigned groups impacts how their opinions on various issues change over the course of the deliberation.
In this chapter, we include a full description of the initial model and algorithm.

A.1 Initial Model

The inputs, output, and constraints of the initial model are identical to those described in Section 2.2. Rather than using a cost function for the objective, however, we simply aim to maximize the number of links generated over all sessions. In particular, for each pair of participants \((i, j)\) where \(i, j \in [n_P]\) and \(i < j\), let \(I_{i,j}\) indicate whether \(i\) and \(j\) met at least once over all sessions. The objective can be written as

\[
\max \left\{ \sum_{i,j \in [n_P] \text{ s.t. } i < j} I_{i,j} \right\}.
\]
A.2 Initial Algorithm

As introduced in Section 3.2, our initial algorithm involved solving a single ILP that aimed to maximize the number of links generated over all sessions.

A.2.1 ILP

Variables

• $x$: For $i \in [n_P], g \in [n_G]$, and $s \in [n_S]$, let $x_{i,g,s}$ be a binary variable that equals 1 if and only if participant $i$ is assigned to group $g$ in session $s$.

• $y$: For $i, j \in [n_P]$ where $i < j$, $g \in [n_G]$, and $s \in [n_S]$, let $y_{(i,j),g,s}$ be a binary variable that equals 1 if and only if participants $i$ and $j$ met in group $g$ during session $s$.

• $z$: For $i, j \in [n_P]$ where $i < j$, let $z_{i,j}$ be a binary variable that equals 1 if and only if participants $i$ and $j$ met at least once over all sessions.

Solution

The solution we wish to record is an assignment from participants to groups over all sessions. This information is easily gathered from the values of the $x$-variables: For each session $s$, each participant $i$ is assigned to the group $g$ for which $x_{i,g,s} = 1$.

Constraints

We first constrain the $y$- and $z$-variables according to their definitions above:

• $y$: For $i, j \in [n_P]$ where $i < j$, $g \in [n_G]$, and $s \in [n_S]$, we have by definition that

$$y_{(i,j),g,s} = (x_{i,g,s} \land x_{j,g,s}).$$

In other words, participants $i$ and $j$ met in group $g$ during session $s$ if both $i$ and $j$ were assigned to group $g$ during session $s$. The logical $\land$ in the statement above can then be rewritten using the following 3
linear constraints:

\[
\begin{align*}
\gamma(i,j)_{g,s} & \geq x_{i,g,s} + x_{j,g,s} - 1, \\
\gamma(i,j)_{g,s} & \leq x_{i,g,s}, \\
\gamma(i,j)_{g,s} & \leq x_{j,g,s},
\end{align*}
\]

- \( z \): For \( i, j \in [n_P] \) where \( i < j \), we have by definition that

\[
z_{ij} = \bigvee_{s, g \in [n_S]} \gamma(i,j)_{g,s}.
\]

The logical OR in the statement above can then be rewritten using the following linear constraints:

\[
z_{ij} \leq \sum_{s, g \in [n_S]} \gamma(i,j)_{g,s}, \\
\forall g \in [n_G], \forall s \in [n_S], z_{ij} \geq \gamma(i,j)_{g,s}.
\]

We now formalize the constraints outlined in Section 2.2. Note that these constraints are identical to the ones described in Subsection 2.2.3, except that we also iterate over all sessions here.

- **Each participant must be assigned to exactly one group:**

\[
\forall s \in [n_S], \forall i \in [n_P], \sum_{g \in [n_G]} x_{i,g,s} = 1.
\]

- **Clustered participants must be assigned to one of their designated groups:**

\[
\forall s \in [n_S], \forall i \in C, \sum_{g \in [n_G]} x_{i,g,s} = 1.
\]

- **The number of participants that can be assigned to each group is constrained by the size function \( Z \) defined in Subsection 2.2.3:**

\[
\forall s \in [n_S], \forall g \in [n_G], \sum_{i \in [n_P]} x_{i,g,s} = Z(g).
\]

- **Diversified attribute representation is constrained according to user-inputted bounds:**

First, for each participant \( i \in [n_P] \), let \( P_i \) denote \( i \)'s attribute vector. Then, the demographic that \( i \) belongs to for attribute \( a \in [n_A] \) is simply \( P_i[a] \). For a diversified attribute value \( v \) and group size \( z \), also let
\[ R(v, z) = (l_{v,x}, u_{v,x}) \]. We desire that

\[
\forall s \in [n_S], \forall d \in D, \forall v_d, \forall g \in [n_G], l_{v_d,z}(g) \leq \sum_{i \in [n_P] \text{ s.t. } P_i[d] = v_d} x_{i,g,s} \leq u_{v_d,z}(g),
\]

where \( v_d \) denotes one of \( d \)'s attribute values.

**Objective**

Finally, we translate the objective from Section A.1 as follows:

\[
\max \left\{ \sum_{i,j \in [n_P] \text{ s.t. } i < j} z_{i,j} \right\}.
\]

**A.2.2 Algorithm**

The initial algorithm simply involves outputting the solution to the ILP, as presented in Subsection A.2.1.
References


This thesis was typeset using \LaTeX, originally developed by Leslie Lamport and based on Donald Knuth’s \TeX. A template that can be used to format a thesis with this look & feel has been released under the permissive AGPL license, and can be found online at github.com/suchow/Dissertate or from its lead author, Jordan Suchow, at suchow@post.harvard.edu.