On the Optimisation of Dinner Seating Plans

Daniel Joffe

A dissertation submitted for the degree of

Bachelor of Science

of the

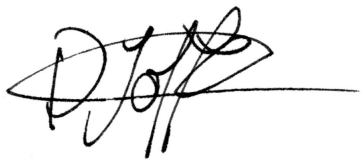
University of Aberdeen

Department of Computing Science

2021

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed: 

Date: Thursday, 3rd of May, 2021

Abstract

For the problem of finding a good seating arrangement at a wedding, where guests have preferences regarding whom they would like to sit with, three heuristics are tested empirically - namely, an integer programming solution, naive hill-climbing and late-acceptance hill-climbing. A large data set is presented to each, and the results were compared by quality and execution time. Hill-climbing is found to be fast but ineffective. Integer programming produced excellent results, but in some cases too slow for tests to complete. Late-acceptance hill-climbing completed all tests but the results, while better than naive hill-climbing, were not as good as integer progamming.

Acknowledgements

My sincerest thanks to Nir Oren, who served dutifully as my advisor over the course of this project; Oron Joffe, whose probing questions always brought out my best wording; and to Laura Joffe, Albert Boehm, Geroge Stoian, and Florence-Edward Hook, who drew my attention to many of the flaws in this paper, some of which I hope I managed to fix. I would like to acknowledge the support of the Maxwell Compute Cluster funded by the University of Aberdeen and the staff who answered my questions about its use. Thanks also to Ioana Pavel, watched cartoons with me to ease the stress of the project.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

When a couple in love decide to tie the knot, it is customary for each spouse to invite large numbers of their friends and family to participate in a celebration. This often involves a feast, during which guests are divided up into tables and expected to chat amicably with those at their table. The problem is that while all the guests are likely friends with at least one of the happy couple, they may not be friends with each other. If care is not taken to control who sits with whom, someone may be stabbed with a salad fork.

Thus, it falls on the couple (or sometimes a close relative) to find an arrangement of guests that minimises the probability of shouting matches and maximises the number of people who have a good time. There are two main requirements to solving the problem: an in-depth knowledge of the relationship-networks in play, and the willingness to spend a long time thinking about combinatorial problems.

Humans can be reasonably expected to hold the first quality, but the second is in shorter supply. Moreover, when stressed, people actually lose cognitive function[11]. Solving combinatorial problems is one of the first and most basic applications of computing[17], so it is a natural to ask whether computers can make the process faster and find better solutions.

Indeed, Bellows and Peterson did ask this and planned the dinner-seating chart for their wedding with the aid of a computer[4]. In this paper, I intend to examine their approach and others in order to compare and find a good, efficient heuristic.

1.2 Applications

The language of this paper will refer to weddings, guests, friendship, tables and seats, however the underlying mathematical problem has other applications. Teachers may want to seat their students such that no pair at a table are likely to cause trouble - this means avoiding relationships that are too negative (such as bully-victim) and relationships that are too positive (good friends who would rather chat loudly than get on with their work). In trains and airplane, passengers would rather sit next to their friends (this is a special

case of the general problem, in that we can expect most friendship groups to be isolated from each other).

1.3 Report Structure

Chapter 1 motivates the problem. Chapter 2 lays out the groundwork for the heuristics tested in this paper. Chapter 3 presents the problem formally and discusses how the heuristics were applied. Chapter 4 outlines the testing strategy and presents the results with some discussion of immediate interpretation. Chapter 5 draws conclusions and suggests further paths of research.

Chapter 2

Background and Related Work

2.1 Background

The problem is in the realm of combinatorial optimisation - we have a finite set of objects which we must arrange to achieve a goal[16]. Thus, this section explores three methods that have been proposed for solving combinatorial problems.

Definition: A solution to an optimisation problem is said to be *feasible* if it satisfies all the constraints of the problem. A feasible solution is *optimal* if no other feasible solution provides a higher value of the objective function[12].

Combinatorial problems always have a finite number of solutions (minimum 0). If there is at least one feasible solution, then an optimum exists.

2.1.1 Integer Programming

Integer programming is an optimisation technique in which a problem is modelled with three components: a set of variables which may take only non-negative integer values (representing the solution), a linear function of these variables (representing the objective), and a set of linear equations or inequalities (representing constraints). For conciseness, this is often written

$$\begin{aligned} \max\ & cx \\ \text{subject to } & Ax \leq b \\ & x \geq 0 \text{ integral} \end{aligned}$$

in which A is a matrix, b is a column-vector and c is a row vector[12]. For example, the problem

$$\begin{array}{l} \max\ 3x+8y \\ \text{subject to} \begin{cases} x \leq 10 \\ x+y \leq 15 \end{cases} \\ x,y \geq 0 \text{ integral} \end{array}$$

can be written

$$\begin{array}{l} \max \begin{bmatrix} 3 & 8 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \\ \text{subject to} \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \leq \begin{bmatrix} 10 \\ 15 \end{bmatrix} \\ x,y \geq 0 \text{ integral} \end{array}$$

Integer programming is closely linked to linear programming, in which problems are described in a similar fashion but without the restriction that solutions have integer values. If a linear programming problem and an integer programming problem have the same objective cx and constraints $Ax \leq b$ then the linear problem is called the *natural relaxation* of the integer problem[12].

Branch and Bound

The *branch and bound* algorithm is a method for finding optimal solutions to integer programming problems. A formal description is not given here, but the main ideas (branch, bound) are explained. For a more complete explanation and more algorithms, see [12].

The capability to solve linear programming problems optimally is assumed. Note that the optimal solution to the natural relaxation of an integer problem will never be worse than the solution to the problem itself.

The first step is to count how many feasible solutions there are. If there are none, then the problem is unsolvable. If there is exactly one, then it is the optimal solution. If there are more than one, we continue.

We use a linear programming solver to find the optimal solution to the natural relaxation. If this solution happens to be integer-valued, then we are done. In the more likely event that at least one co-ordinate x_i is fractional with value x_i^*, we *branch*.

Create two new problems from the current, L and R. To L, we add the constraint $x_i \leq \lfloor x_i^* \rfloor$. To R, we add $x_i \geq \lceil x_i^* \rceil$. We then recursively find the optimal solution to both

branches and select the better one.

So far, branching has given us little more than an organised brute-force search. In order to speed things up, we prune branches that cannot hold an optimal solution. Suppose that we partially explore branch L and find a feasible solution whose objective value is 50, and that the natural relaxation of R has an maximum objective value of 45. Since all solutions within R have their objective function *bounded* above by 45, (hence also by 50), R need not be explored at all.

Even after pruning non-viable branches, an implementation of this algorithm must have some way of selecting an order to explore the remaining branches in, just as in any search of a binary tree.

Branch and bound is guaranteed to find an optimal solution if there is one, and is normally faster than an exhaustive search of all feasible solutions[14], but it cannot always do so in polynomial time. Integer programming can be used to model computationally hard problems, such as the knapsack problem[12], so any algorithm with a guarantee of optimality has no guarantee of terminating quickly for arbitrary problems.

The programming library *OR-Tools* from Google allows users to specify problems using the framework of integer programming and uses its own algorithms and heuristics to solve them.

2.1.2 Hill-Climbing

Hill-climbing is a simple and widely-known optimisation technique[18][3]. It requires an initial feasible solution to its problem, a way of comparing solutions' quality pairwise, and a way of introducing small feasible mutations. A termination condition is required. The algorithm is as follows:

```
S = a random solution;
while (the termination condition is not met) {
    T = a solution similar to S, but with a small mutation;
    if (T is better than S) {
        S = T
    }
}
return S;
```

Termination conditions are normally either a limit on computation resources or a threshold quality for the solution S.

Definition: A *local optimum* is a feasible solution which is better than or equivalent to all its immediate neighbours by the mutation method.

Given enough time, hill-climbing is guaranteed to find a local optimum for its problem, as it only accepts new solutions which are better than its current candidate. However,

there is no guarantee that this local optimum is the global optimum, or is even of similar quality. Once a local optimum is found, there is no way to escape it, as all possible mutations of the current candidate solution are by definition no better. In the trade-off between exploitation and exploration[2], hill-climbing solidly prefers exploitation.

Hill-climbing is closely related to gradient-descent, an algorithm commonly used for training artificial neural networks. Instead of selecting small changes randomly, gradient-descent uses differential calculus to find the direction of change with the highest impact[8].

2.1.3 Late-Acceptance Hill-Climbing

Hill-climbing (henceforth also called naive hill-climbing for clarity) never leaves its local optimum because it refuses all solutions that are not strictly better than its current candidate. The direct solution to this problem is to accept solutions that are worse than the current candidate, but that brings in the obvious problem - if we don't eliminate bad solutions, how will we find the good ones?

Various methods have been proposed, including using randomness to occasionally accept worse solutions[15]. Late-acceptance hill-climbing is a variant that keeps a list of previous solutions. By comparing a proposed solution against the most- and least- recent solutions on the list, we weaken the requirement for acceptance without losing all quality control.

Given a parameter h - the amount of history remembered, the algorithm for late-acceptance hill-climbing is as follows[7]:

```
S = an empty array of solutions;
for (i = 0; i < h; i++) {
    S[i] = a random solution;
}
while (termination condition is not met) {
    T = a solution similar to S[|S| - 1],
        but with a small mutation;
    if ((T is better than S[|S| - 1])
        OR (T is better than S[|S| - 1 - h])
    ) {
        S[|S|] = T;
    }
}
return the best S[i] for integer i in [0, h)
```

The algorithm above assumes an infinite-capacity array, but the same idea is possible with a fixed-capacity ring buffer - we only read S from indices between $|S| - 1 - h$ and $|S| - 1$

Late-acceptance hill-climbing was found to work well on benchmark problems, and won first place in the International Optimisation Competition of 2011[7].

2.2 Related Work

Bellows and Peterson[4] described the dinner seating problem as an integer programming problem. They represent problem instances using a relationship matrix - a square matrix wherein each pair of guests has their relationship described by a number. 0 is used for strangers, 1 for friends, and 50 for couples. They propose using negative values in the relationship matrix for negative relationships (divorced, etc.), though leave this for future work.

Their approach was tested on their own wedding, for which the results required only minor modification and were reached in 36 hours.

Chapter 3

Methodology

3.1 Formulation of the Problem

To each pair of guests we assign an integer number representing the quality of their relationship. 0 means no relationship - two people do not know each other. Positive values (normally 1) indicate an amicable relationship, such as friendship or marriage. These people should be sat next to each other if possible. Negative values represent adversarial relationships, and these people should be sat away from each other if possible. In general, the friendlier a relationship is, the higher its value.

This is represented as a matrix with the following properties:

- symmetric - the relationship between guest i and guest j is the same as the relationship between guests j and guest i.

- zeroes on the diagonal - for the purposes of being in good company, a person does not have a relationship with themselves.

Definition: A wedding is considered to be *binary-representable* if its relationship matrix consists only of 0s and 1s.

In addition to the relationship matrix, whose size indicates the number of guests, the number of tables is specified. It is assumed that the number of tables perfectly divides the number of guests, so that each table is full. In practice, this presents a problem if the number of guests has few divisors or is prime. In this case, one can pad out the number of guests by adding pretend guests who have no preferences for any of the real ones in particular.

A solution is a partition of the guests, where each bin represents a physical table. All tables are the same size. Solutions are not concerned with the locations of the tables relative to each other or the seating arrangement within a table.

To each guest, a happiness score[1] is calculated by the formula

$$H(i) = \sum_{j \in T(i)} R_{ij} \tag{3.1}$$

where

- i is the guest whose happiness is being measured.
- $T(i)$ is the set of guests at i's table.
- R is the relationship matrix.

Using this, the overall success of a wedding can be quantified with various statistical summaries of happiness: total, mean, median, maximum, and minimum.

For an individual i, we can put an upper bound on their happiness with $\sum_{R_{ij}>0} R_{ij}$, and the upper bound for total happiness is the sum of everyone's individual upper bounds. As seen in the section 3.2, this is not always possible to achieve.

An additional metric is used - the number of *lonely guests*. A guest is considered to be lonely if they have no positive relations at their table. This can be mathematically formulated as

$$\{j \in T(i) \mid R_{ij} > 0\} = \emptyset \tag{3.2}$$

3.2 A Small Example

Say we have a wedding with 16 guests and 4 tables to seat them at, with the relationship matrix given by

[1]The assumptions in this formula may not hold. For example, say that Alice is friends with Bob from work and Carol from bowling. However, a conversation about bowling may not include Bob, and a conversation about work Carol. Alice's happiness cannot benefit from both of them. Furthermore, a large table poses some challenge in facilitating conversation between people far from each other. Any quantitative measure of happiness is somewhat arbitrary; a more detailed model is beyond the scope of this paper.

$$\begin{bmatrix}
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

Since the matrix is binary-representable, it is natural to interpret it as the adjacency matrix of a graph. We can more easily understand the network with the visualisation in figure 3.1. Nodes represent people, indexed from 0. The presence of an edge represents friendship. There are no negative relationships, so we don't need to worry about showing those.

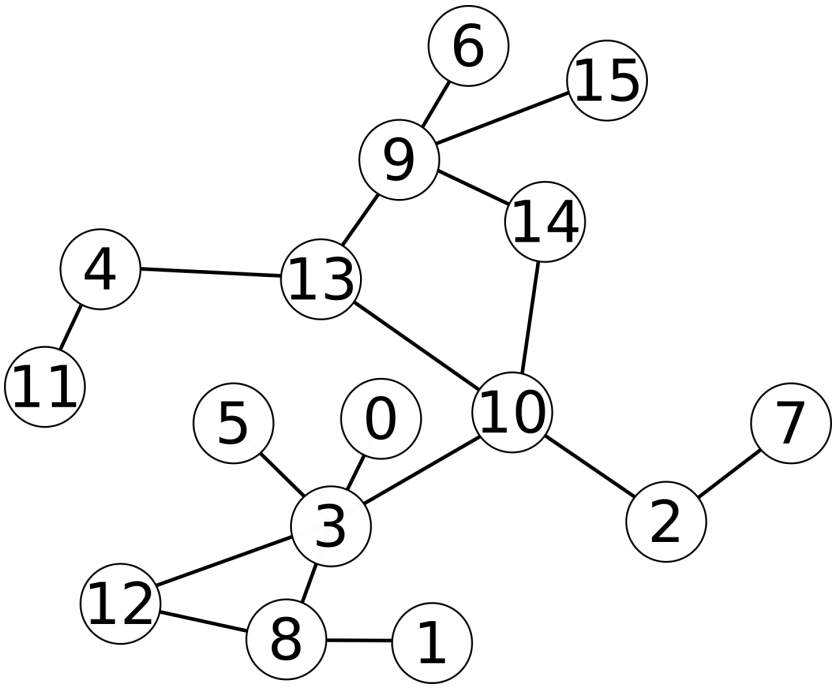


Figure 3.1: A visualisation of the example problem

Consider the solution $\{\{4, 10, 11, 13\}, \{6, 9, 14, 15\}, \{1, 2, 7, 8\}, \{0, 3, 5, 12\}\}$. Everyone is paired with at least one of their friends, so no guest remains lonely. Guest number 8 would be happier if their table was $\{1, 3, 8, 12\}$, but then without 3 to talk to, guests 0 and 5 would be left lonely. Also, guest 3 would be happiest at a table with $\{0, 3, 5, 8, 10, 12\}$, but tables only seat four people.

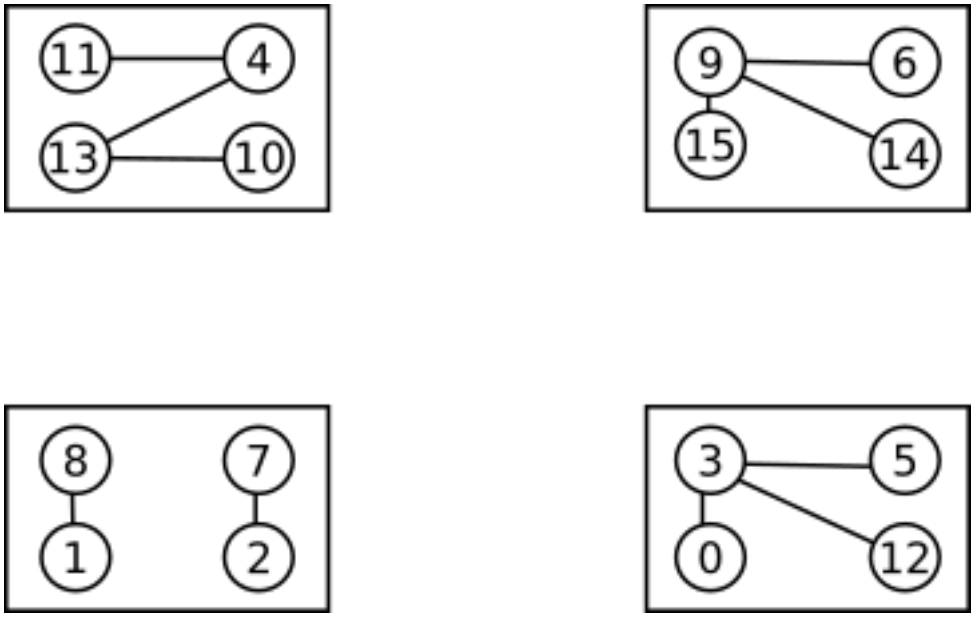


Figure 3.2: A possible solution to the example problem

3.3 The Futility of Brute-Force

The simplest algorithm to solve this problem is that of brute force. Consider all possible arrangements and select the best one according to one of the metrics above. The result will certainly be optimal, and the algorithm will terminate eventually, but how long can it be expected to take?

Supposing there are $n = ts$ guests, where s is the size of each table and t is the number of tables. How many seating charts must be considered, and how long will it take to calculate the metric for each one?

3.3.1 Number of Charts

Suppose that the brute-force algorithm internally represents a chart as a $t \times s$ matrix, where each row represents a table. As there are n entries, there are $n!$ possible such matrices. However, it is possible to exploit certain symmetries to reduce the number that must be considered.

First, the tables can be permuted as units without changing the underlying seating chart. Since there are t tables, the number of candidate charts is reduced by a factor of $t!$.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \approx \begin{bmatrix} 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 \\ 8 & 9 & 10 & 11 \end{bmatrix} \tag{3.3}$$

Second, guests can be permuted within their tables; so long as no guest moves from one table to another, the underlying seating chart does not change. This reduces the number of charts to consider by an impressive factor of $s!^t$.

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \approx \begin{bmatrix} 3 & 2 & 1 & 0 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \tag{3.4}$$

So the total number of charts to consider is $\frac{(st)!}{t!\ s!^t}$.

3.3.2 Time Spent on Each Chart

All of the metrics defined in section 3.1 require $\Theta(ns) = \Theta(ts^2)$ time to compute for a given chart. However, since we are computing these metrics for many charts, we can speed this up by caching repeated computation. To demonstrate, consider the following two charts:

$$\begin{bmatrix} 0 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 \\ 8 & 9 & 10 & 11 \end{bmatrix} \text{ and } \begin{bmatrix} 0 & 1 & 2 & 3 \\ 5 & 7 & 8 & 10 \\ 4 & 6 & 9 & 11 \end{bmatrix} \tag{3.5}$$

The happiness (and loneliness) of guests 0, 1, 2, and 3 is unchanged. Suppose we created a lookup table that took an entire table of guests and returned the happiness of each of them as a tuple (by means of hashmap, prefix tree, or something similar). The cost per lookup would be at least $\Theta(s)$ time since each guest in the table must be inspected at least once to tell [0 1 2 3] apart from [0 1 2 4]. This is better than the $\Theta(s^2)$ method of computing happiness for each guest in the table from scratch.

Supposing we had a complete lookup table, calculating any aggregate metric for a chart would take $\Theta(ts)$ time instead of $\Theta(ts^2)$.

There is also the further matter of generating the lookup table: an $\Theta(s^2)$ function must be run once for each possible table of guests. A table consists of s items chosen from n, where order is irrelevant and repetitions are not allowed, so the total number of possibilities is ${}^nC_s = \frac{(st)!}{s!(st-s)!}$. Generating the lookup table would then take $\Theta\left(\frac{(st)!s^2}{s!(st-s)!}\right)$ time.

In conclusion, a brute-force algorithm that maximally leverages symmetry to avoid recomputation would have a time-complexity of $\Theta\left(\frac{(st)!s^2}{s!(st-s)!} + \frac{(st)!st}{t!\ s!^t}\right)$. Fixing s arbitrarily at 6, a table is presented below with the approximate computing cost for various sizes of weddings.

Number of Tables	Number of Guests	Approximate Number of Operations
1	6	42
10	60	3.7×10^{48}
20	120	2.4×10^{125}
30	180	2.6×10^{213}
40	240	6.1×10^{308}

Figure 3.3: The approximate cost of brute-force

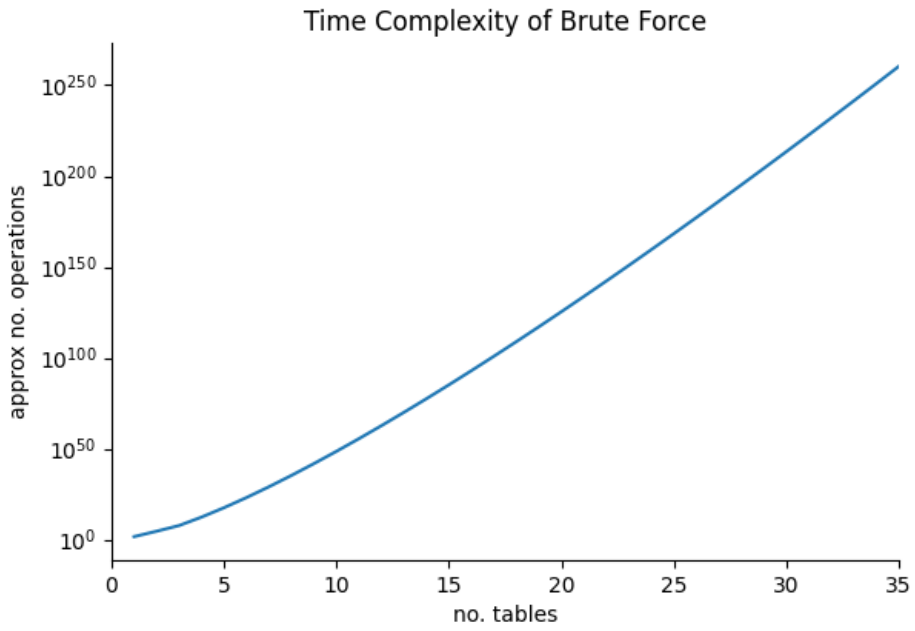


Figure 3.4: A log-plot of the time-complexity of a maximally-efficient brute-force algorithm

3.4 Heuristics

Three heuristics were selected as possible methods - the integer programming method similar to that used in [4], a naive hill-climbing algorithm, and late-acceptance hill-climbing.

3.4.1 Integer Programming

A solution based on integer programming requires a set of integer variables and constants, an objective function, and a set of constraints. The objective function and constraints may only use linear combinations of the variables - scaling by a variable by a constant is possible, but multiplication of variables is not.

Variables and Constants

Taking the convention that i is used for indexing tables, j and k are used for indexing people, and that indexing starts at 1,

Variable	Dimension	Type	Meaning
n	scalar	constant	the number of guests
s	scalar	constant	the seating capacity of each table
t	scalar	constant	the number of tables
R	$n \times n$	constant	R_{jk} represents the quality of the relationship between guests j and k.
a	$t \times n$	variable	a_{ij} is 1 if guest j is seated at table i, 0 otherwise.
p	$t \times n \times n$	variable	p_{ijk} is 1 if both guest j and guest k are seated at table i, 0 otherwise.

Figure 3.5: List of variables used in integer programming

When the solution is found, it can be transformed into a partition of guests by reading the a matrix.

Objective

The objective function is total happiness, expressed linearly as

$$\sum_{i=1}^{t} \sum_{j=1}^{n} \sum_{k=1}^{n} R_{jk} p_{ijk} \tag{3.6}$$

Constraints

The constraints must capture a few keys ideas:

- Each guest must be seated at exactly one table.
- Tables have a fixed capacity of people.

- Two people are seated together if and only if they are both seated at the same table.

These are expressed in the language of integer programming as follows:

$$\forall i = 1..t \; \forall j = 1..n \quad 0 \leq a_{ij} \leq 1 \tag{3.7}$$

$$\forall i = 1..t \; \forall j = 1..n \; \forall k = 1..n \quad 0 \leq p_{ijk} \leq 1 \tag{3.8}$$

$$\forall j = 1..n \quad \sum_{i=1}^{t} a_{ij} = 1 \tag{3.9}$$

$$\forall i = 1..t \quad \sum_{j=1}^{n} a_{ij} \leq s \tag{3.10}$$

$$\forall i = 1..t \; \forall k = 1..n \quad \sum_{j=1}^{n} p_{ijk} \leq s a_{ik} \tag{3.11}$$

$$\forall i = 1..t \; \forall j = 1..n \quad \sum_{k=1}^{n} p_{ijk} \leq s a_{ij} \tag{3.12}$$

Equation (3.9) states that each guest must be seated at exactly one table. Equation (3.10) enforces a maximum capacity on tables. Equations (3.11) and (3.12) extend the idea of (3.10) from the a variable to p. Ideally, p_{ijk} would be equal to $a_{ij}a_{ik}$ - for two people to be seated at a table together, they both need to independently be sitting at that table. However, multiplication is not allowed in integer programming. Equation (3.11) is can be viewed as scaling (3.10) by a factor of a_{ik}. Similar for equation (3.12).

The last two constraints ensure that not too many p_{ijk}'s can be 1, but there is still the question of why any would be at all. Recall the objective function (3.6) - the optimiser needs to set p_{ijk} to 1 for some i in order to take advantage of R_{jk}, so if the constraints allow it to, then it will.

This approach is taken almost verbatim from [4]. The first difference is that they consider loneliness of guests to be a constraint and avoid it at all costs, while I consider it a metric so that it is easier to find feasible solutions - not all of the automatically-generated problems (section 3.5) may have solutions with no lonely guests. The second difference is immediately below.

The above algorithm only works under the assumption that the entries of R are all positive, although the flaw is harmless if some entries are zero. Consider the example

$$R = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -100 \\ 0 & 0 & -100 & 0 \end{bmatrix} \tag{3.13}$$

$$t = 2$$

A tempting solution is

$$a = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \tag{3.14}$$

$$p_1 = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$p_2 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The objective function has value $R_{12} + R_{21} = 1 + 1 = 2$. However, if we actually look at the chart that a represents we have $\{\{1,2\},\{3,4\}\}$, in which the total happiness is $R_{12} + R_{21} + R_{34} + R_{43} = 1 + 1 - 100 - 100 = -198$. The general principle is that when the relationship between two people is negative, it is all too convenient to ignore.

To get around this problem, if there are any negative entries in R, a positive bias is added to all entries. The problem above would become

$$R' = \begin{bmatrix} 100 & 101 & 100 & 100 \\ 101 & 100 & 100 & 100 \\ 100 & 100 & 100 & 0 \\ 100 & 100 & 0 & 100 \end{bmatrix} \tag{3.15}$$

and it is clearer to the system that guests 3 and 4 must not be seated together.

3.4.2 Naive Hill-Climbing

A hill-climbing algorithm requires a method for comparing solutions, a method for mutating solutions, and a termination condition. In this case, total happiness was chosen as the measure of comparison for consistency with integer programming. A mutation is a single swap of any two guests, chosen equiprobably. The process ends when the frequency of improvement falls below a threshold of 0.001, measured by an exponential moving average with the update formula[1]:

$$f_{t+1} = \alpha u + (1 - \alpha) f_t \tag{3.16}$$

where

- f_t is the frequency of updating the solution at time t.

- u is whether the proposed change is an improvement. 1 if so, 0 if not.

- α is a constant in the open interval (0, 1). Higher values represent more weight given to more recent data. To prevent premature termination, a value of 0.01 was chosen.

3.4.3 Late-Acceptance Hill-Climbing

Late-acceptance hill-climbing has most of the same properties as naive hill-climbing, but also requires a queue length. 1000 was chosen arbitrarily. The purpose of the queue is to remember the past 1000 candidate solutions, and accept a proposed solution is if it is better than either the most or least recent of them. This allows the heuristic to more-easily escape local optima.

3.5 Generation of Example Data

Examples of the following categories were automatically generated for testing: *complete*, *ring*, *sparse*, and *tense*. Complete, ring, and sparse weddings are binary-representable.

3.5.1 Complete

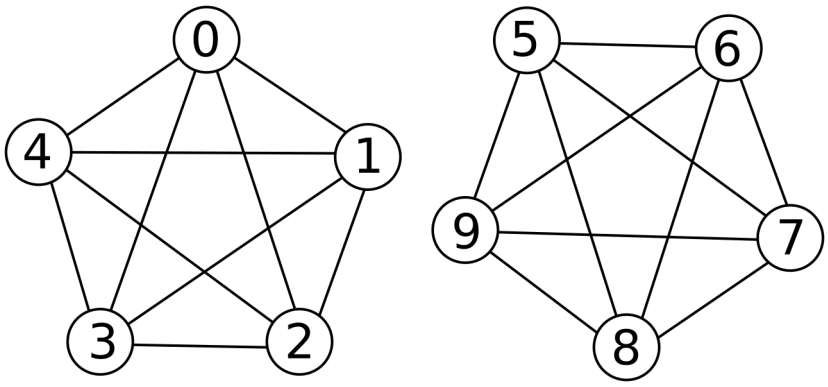


Figure 3.6: An example of a complete problem

In a complete wedding, guests are split into groups the same size as the tables. Everyone in each group has a relationship of 1 with everyone else in the group, and a relationship of 0 with everyone outside it. If we view the group as a graph using the same convention as in 3.2 then all the connected components are complete.

It is easy to find the optimal solution if a wedding is known to be complete, and easy to see that it is indeed optimal.

Supposing that the bride and groom each brought a group of five close friends for a two-table wedding, the matrix would be

$$\begin{bmatrix}
0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}$$

and the optimal solution is

$$\{\{0,\ 1,\ 2,\ 3,\ 4\},\ \{5,\ 6,\ 7,\ 8,\ 9\}\}$$

3.5.2 Ring

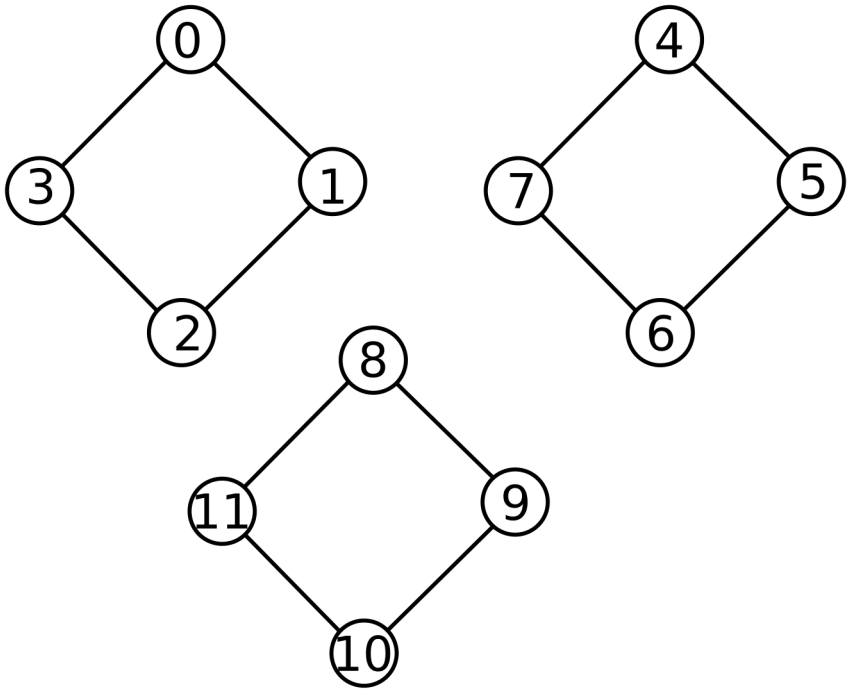


Figure 3.7: An example of a ring problem

In a ring wedding, each guest knows precisely two other people and when ideally seated, each person would have those two at their table. A wedding with three rings of four would have matrix and solution:

$$\begin{bmatrix}
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0
\end{bmatrix}$$

$$\{\{0,\ 1,\ 2,\ 3\ \},\ \{4,\ 5,\ 6,\ 7\},\{8,\ 9,\ 10,\ 11\}\}$$

Ring weddings are similar to complete weddings in that there is a single optimal solution which is easy to find. Unlike complete weddings, the maximum happiness of an individual is capped at 2 rather than scaling with table size.

Rather than complete sub-graphs, the relationship network is a collection of equally-sized rings, hence the name.

3.5.3 Sparse

Sparse weddings are the first kind to be produced non-deterministically. There are two stages: in the first, everyone is randomly assigned some friends from the whole group. In the second, everyone is randomly assigned some friends from the list of their friends' friends. Pseudocode is given below to make the algorithm more concrete.

```
Initialise the network with no-one knowing anyone.
for (guest_i = 0; guest_i < n; guest_i++) {
    loop {
        L = the number of friends guest_i currently has;
        with probability 1 / (L + 1) {
            guest_j be a random integer in [0, n),
                guest_j != guest_i;
            set guest_i and guest_j as friends.
        }
        else {
            break;
        }
    }
```

```
}
for (guest_i = 0; guest_i < n; guest_i++) {
    loop {
        L = the number of friends guest_i currently has;
        with probability 1 / (L + 1) {
            guest_j = any friend of guest_i,
                chosen randomly with equal probability;
            guest_k = any friend of guest_j,
                chosen randomly with equal probability;
            set guest_i and guest_k as friends;
        } else {
            break;
        }
    }
}
```

When implementing this, one must be careful not to allow self-friendship.

Such weddings have sparse friendship-graphs.

3.5.4 Tense

Tense weddings are also created non-deterministically. In a tense wedding, the numerical value of each relationship is sampled from a normal distribution with a mean of 0 and standard deviation of 30 and rounded to an integer. Since positive and negative relationships are equally likely, such weddings can be tense.

3.6 Implementation

Each heuristic was programmed and made into an executable known as a *solver*. Solvers receive the problem from standard input and write their solution to standard output. Json is used as the serialisation protocol for problems and solutions.

Problems were generated by the programs `data-gen` and `mass-data-gen`. As command-line arguments, data-gen takes the type of problem, the number of tables, and the table-size. It creates a problem instance and writes the json to standard output. `mass-data-gen` is a wrapper for `data-gen`, repeatedly spawning it as a subprocesses and piping the output to files. It takes the path to a text file as a command-line argument, and that text file contains the instructions for `data-gen`.

To evaluate solvers' performance, a program called `score` was written. As command line arguments, it takes the path to a solver and the path to a directory containing problem instances. It reads the content of each problem, spawns the solver as a subprocesses and communicates with it via pipes. When the solver's solution is found, the metrics given in section 3.1 are calculated and written into a csv file, as well as the duration that the solver

ran for. Each solver is run multiple times on each problem, to reduce the effect of external factors on measurements of duration and the effect of random-number-generation on the solution.

The integer programming solver was programmed in Python 3 using the Python bindings to Google's integer programming library OR-Tools. The solvers based on hill-climbing were programmed in Rust. Since the integer programming solver spends most of its time in the C++ code of OR-Tools, the time-efficiency of the solvers is directly comparable; C++ and Rust are both compiled to machine-code and optimised similarly.

Chapter 4

Results

4.1 Testing Data

A suite of problems was created for each type of wedding listed in section 3.5. Each suite consists of weddings with the following problem sizes: two tables of two, three tables of three, four tables of four, ten tables of ten, twelve tables of twelve, and fourteen tables of fourteen. The number of people in the wedding is then approximately one of 5, 10, 50, 100, 150, 200.

Since the sparse and tense problem-types can produce different data given the same inputs, five different weddings were produced for each wedding size in those suites.

4.2 Happiness

Numerous graphs are shown below, various metrics for all solvers on all suites. We assign a happiness score to each individual in the wedding, and for each wedding, calculate the minimum, maximum, mean, and median of that. Maximum and minimum happiness represent the happiest and saddest people at the party respectively, and are considered as their own metrics, not as an error-bound on the average.

For the deterministic suites, each data point represents the average of that metric across ten runs of the same solver on the same wedding. For the non-deterministic suites, each data point represents the average of fifty tests for each solver and each wedding size: five weddings of each size and ten runs on each wedding.

Complete Suite

In complete weddings, there are disjoint sets of people who all know each other, but no-one else. The size of the sets is equal to the size of the tables. Therefore, it is possible for everyone to know everyone else at their table, and for the happiness score to be uniform across the guest list, equal to the number of other people at the table. In our data set, table size and table number are equal, so the maximum individual happiness is $\sqrt{n} - 1$.

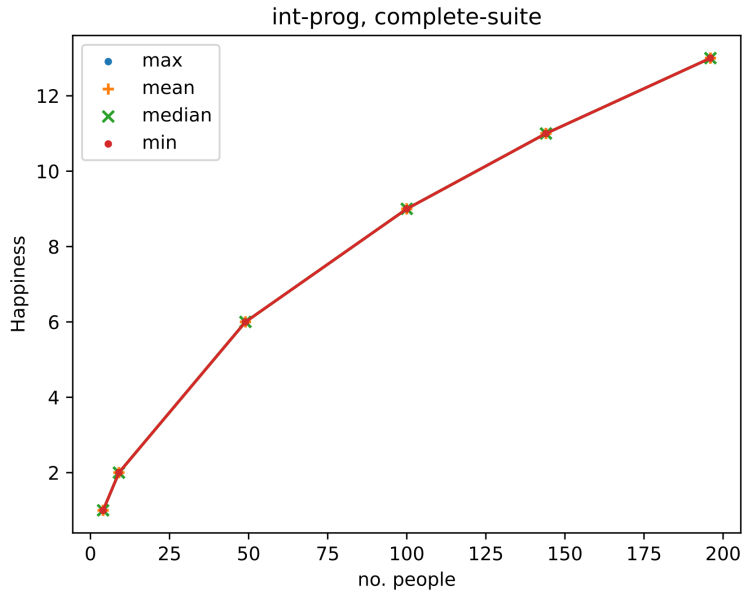


Figure 4.1: Happiness by wedding size on complete weddings. Integer programming

Figure 4.1 shows the results of the integer programming method, which finds optimal solutions in all cases. Note that the mean, max, median, and min lines all overlap because all guests have the same happiness.

A human would find these solutions just as reliably. It's "obvious" - just put everyone with their friends.

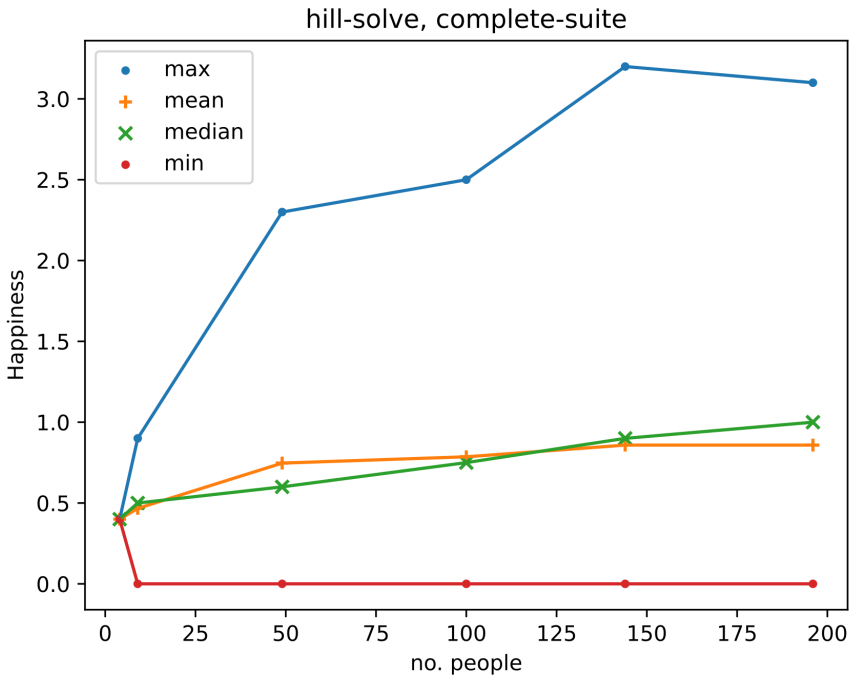


Figure 4.2: Happiness by wedding size on complete weddings. Naive hill-climbing

Figure 4.2 shows the performance of naive hill-climbing, which is considerably worse. The average maximum for integer programming reaches 13 in the largest case, and only approximately 3 for hill-climbing. The minimum is 0 in most cases. The median and mean crawl in the right direction, but the range is from 0.5 – 1 as opposed to 2 – 13.

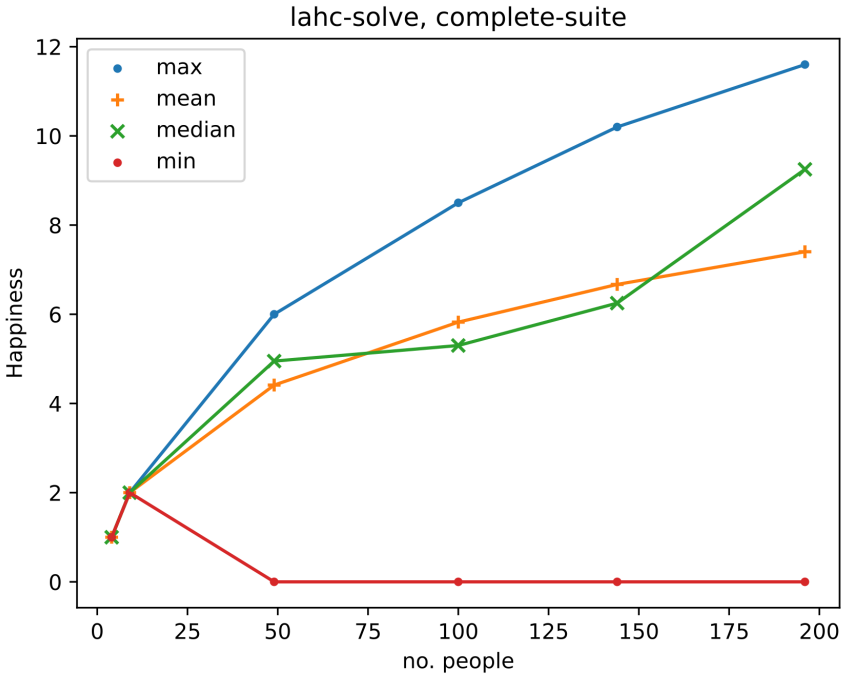


Figure 4.3: Happiness by wedding size on complete weddings. Late-acceptance hill-climbing

Late-acceptance hill-climbing's performance is shown in figure 4.3. The maximum happiness attained is similar to that of integer programming, which indicates that at least one table is arranged almost perfectly in each wedding. The mean and median follow a broadly ascending curve, reaching approximately half the score of integer programming. The minimum once again is 0 in most cases.

For complete problems, both hill-climbing methods provide poor results compared to a human, while integer programming does optimally.

Ring Suite

In a ring wedding, each person knows exactly two other people. In an optimal solution, we may imagine circular tables in which everyone knows those at their immediate left and right. In this case, everyone would have a happiness of 2. A human who knows the structure of the problem would find it equally easy to resolve as the complete wedding.

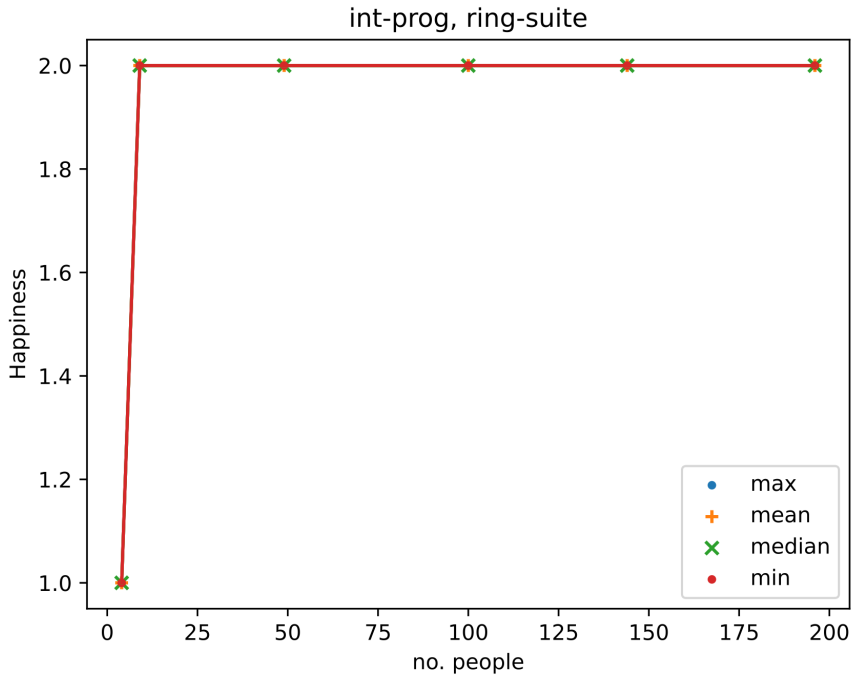


Figure 4.4: Happiness by wedding size on ring weddings. Integer programming

Figure 4.4 shows perfect results from the integer programming solver. The happiness score of 1 is an artefact of a wedding party consisting of two tables seating two each. Each guest is seated next to only one other.

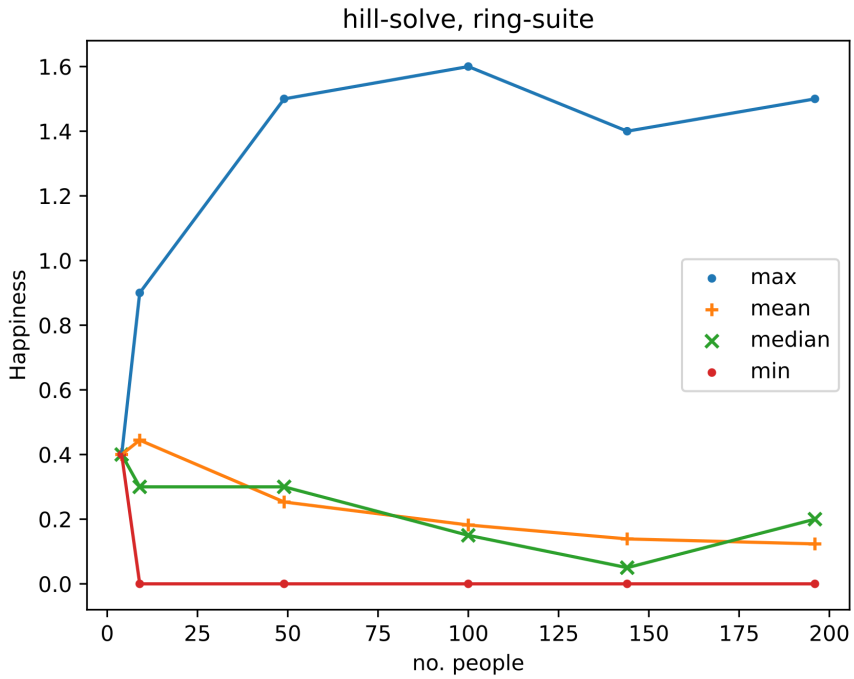


Figure 4.5: Happiness by wedding size on ring weddings. Naive hill-climbing

Hill-climbing, shown in figure 4.5, does not appear to find optimal solutions for any ring wedding. The maximum is greater than 1 for large enough weddings, indicating that it is capable of seating some people next to both of their acquaintances, but not reliably so, as the mean and median are less than 1. These are averaged across different runs on the same wedding, so an average median of 0.2 can be interpreted as the actual median value often being 0 - most people are seated next to no-one they know.

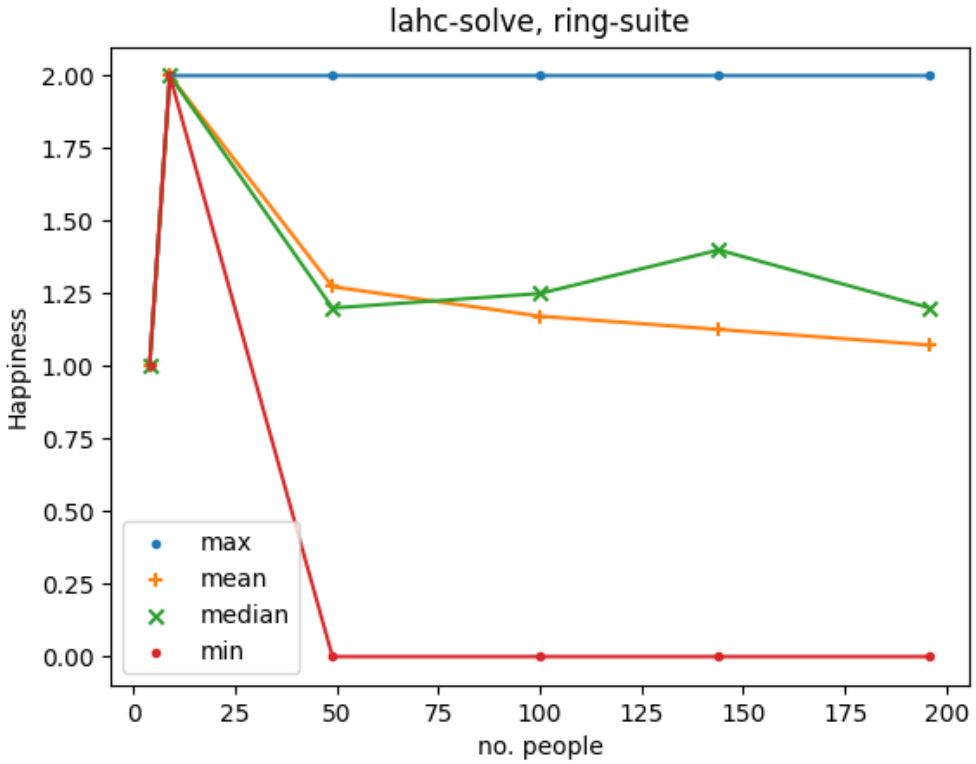


Figure 4.6: Happiness by wedding size on ring weddings. Late-acceptance hill-climbing

Late-acceptance hill-climbing, shown in figure 4.6 performs better than the naive variant. The maximum is consistently 2, indicating that someone is always treated optimally. Averages hover slightly above 1, indicating that most people are normally seated next to one other person they know, or occasionally both. As with naive hill-climbing, the minimum quickly drops to 0, leaving some guests lonely.

Overall performance of the solver for ring weddings is similar to that for complete weddings.

Sparse (Random)

In sparse weddings (referred to as "random" in the code), people are randomly assigned friends - always at least 1, normally around 2, but sometimes more. For details of the algorithm of data generation, see section 3.5.3.

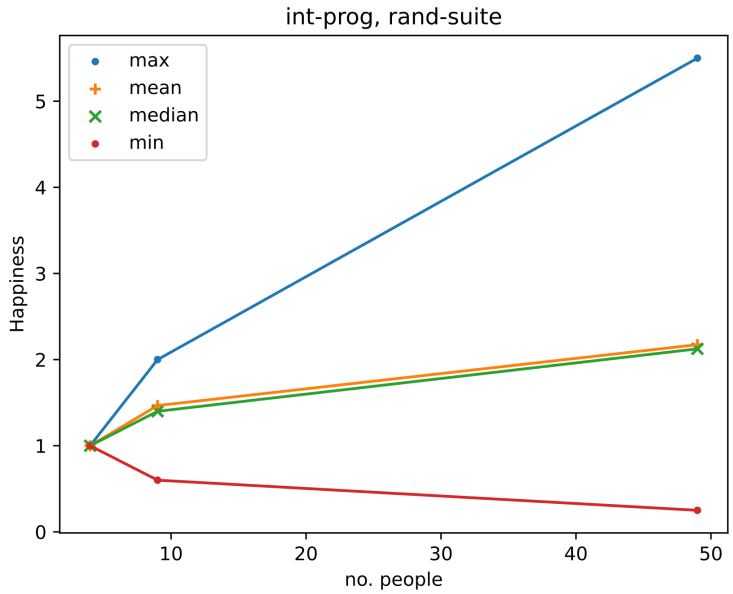


Figure 4.7: Happiness by wedding size on sparse weddings. Integer programming

Integer programming could only solve weddings of up to 49 guests within the time-frame allocated for testing (24 hours per wedding, with each wedding run 10 times.) Comparison between heuristics is limited, but it can be seen that there if often at least one person separated from all their friends. The maximum increases with wedding size, so the heuristic is able to seat one of the more popular people with several friends.

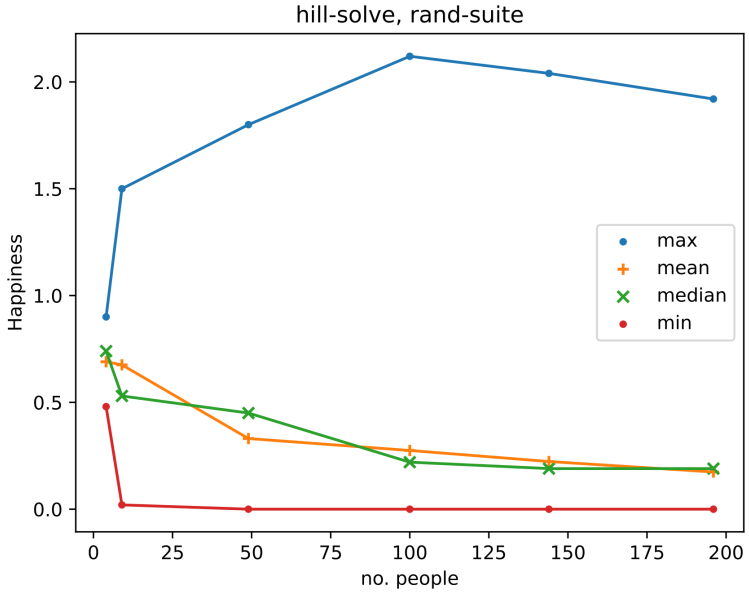


Figure 4.8: Happiness by wedding size on sparse weddings. Naive hill-climbing

On sparse weddings, hill-climbing (figure 4.8) seldom puts any guest with more than two friends, and always places at least one guest with none. Furthermore, most guests for

any wedding size are placed with no friends, and the larger the wedding, the larger the proportion of such guests.

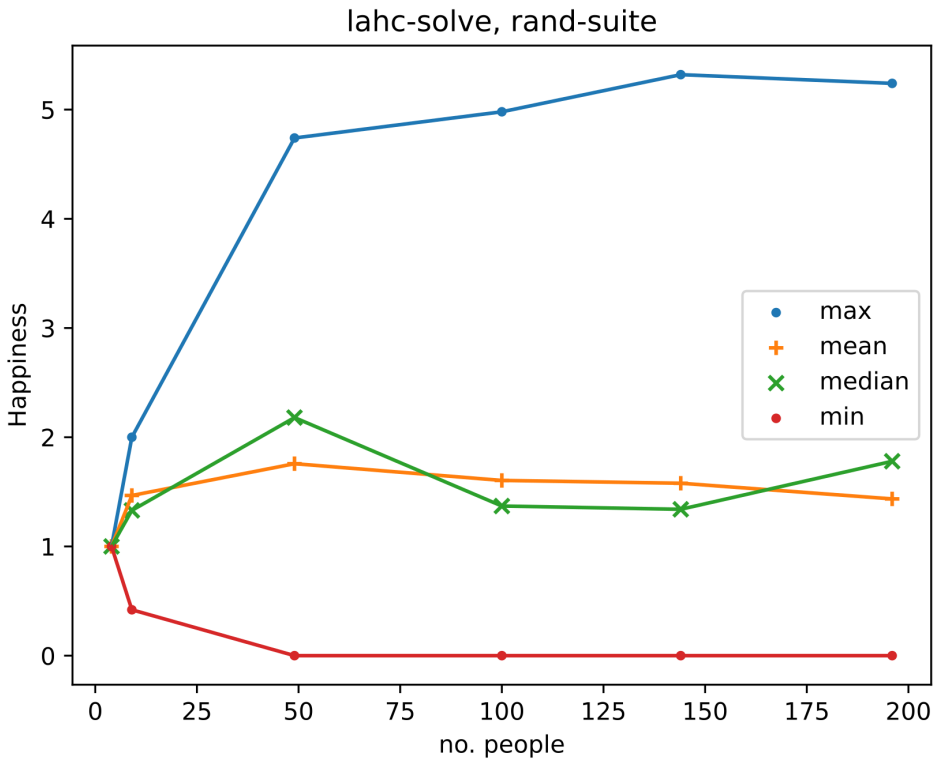


Figure 4.9: Happiness by wedding size on sparse weddings. Late-acceptance hill-climbing

Late-acceptance hill-climbing (figure 4.9) gives similar results to integer programming on the existing data. On larger weddings, the overall trend is roughly constant. The mean and median are consistently above 1.

In summary, integer programming takes too long for large sparse weddings, but works well on small enough data. Late-acceptance hill-climbing gives similar quality and can deal with larger data. Naive hill-climbing produces inferior results.

Tense

In a tense wedding, the relationship between any two people is sampled from a normal distribution with a mean of 0 and a standard deviation of 30.

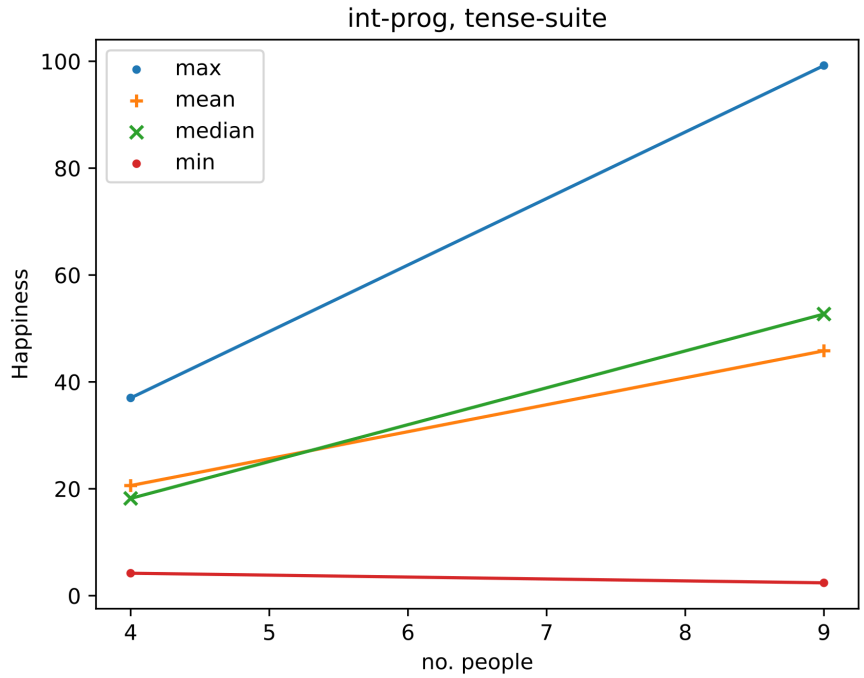


Figure 4.10: Happiness by wedding size on tense weddings. Integer programming

Integer programming (figure 4.10) could only run on very small weddings in the time allotted. Due to lack of data, there is very little to say about it.

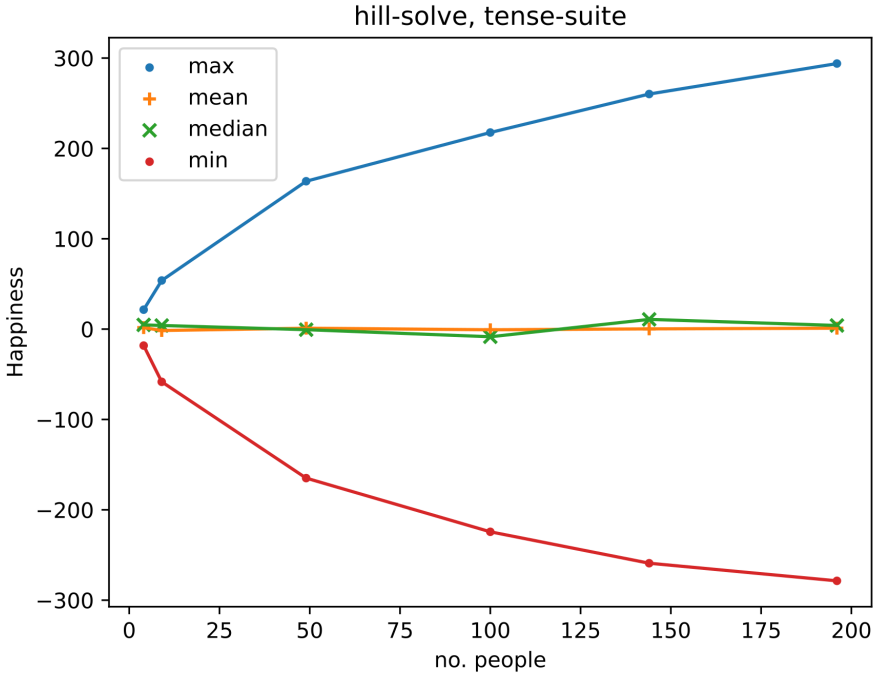


Figure 4.11: Happiness by wedding size on tense weddings. Naive hill-climbing

For naive hill-climbing (figure 4.11), the average happiness is approximately 0

throughout the range. The maximum and minimum are symmetrically increasing in magnitude, though minimum happiness is negative. This is what one would expect from selecting a seating chart at random without even looking at people's preferences.

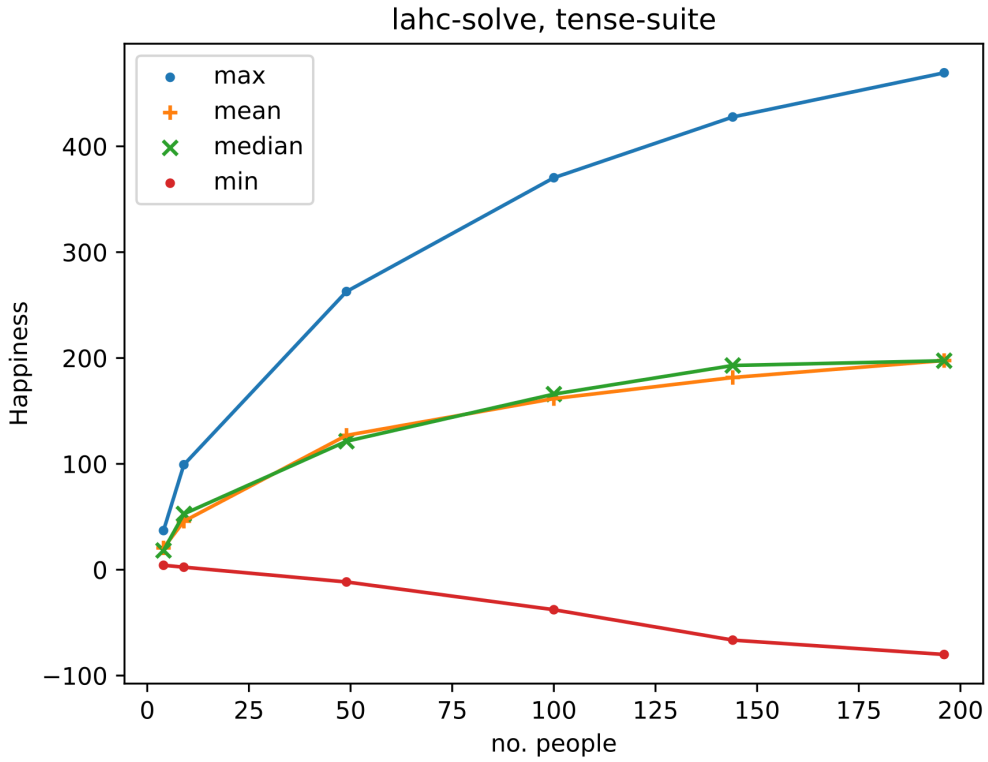


Figure 4.12: Happiness by wedding size on tense weddings. Late-acceptance hill-climbing

Late-acceptance hill-climbing (figure 4.12), happiness generally increases as weddings and tables get larger. While the minimum happiness is decreasing and negative, it is lesser in magnitude than median and mean, indicating that most people have satisfactory arrangements.

For tense weddings, late-acceptance hill-climbing is the only heuristic to provide a usable result.

4.3 Loneliness

The number of lonely guests (defined in section 3.1) is shown below for each solver and each suite. As with happiness, averages are taken for each solver and each wedding size.

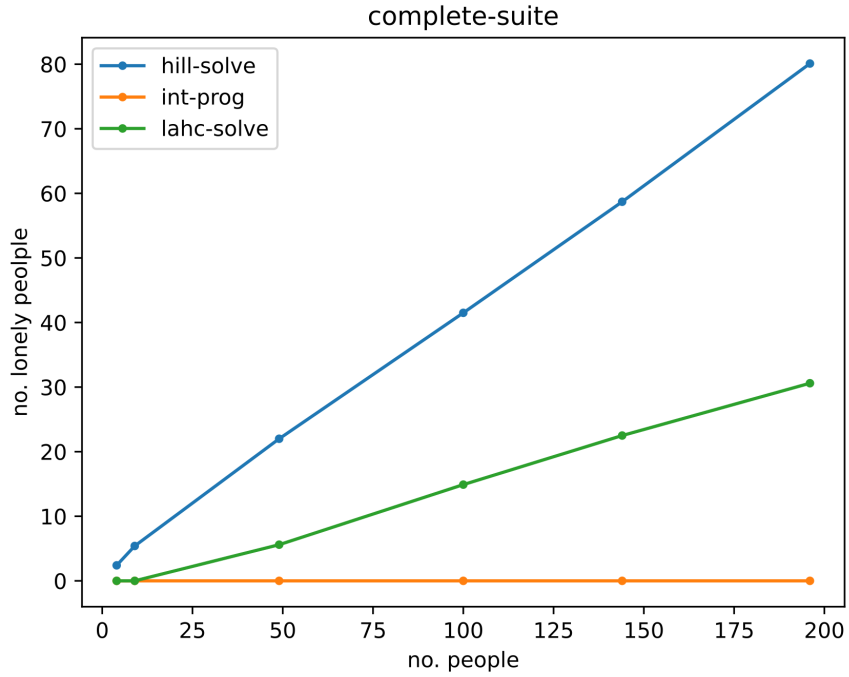


Figure 4.13: Number of lonely people by wedding size. Complete weddings

In complete weddings (figure 4.13) Integer programming shows 0 lonely guests throughout. Naive hill-climbing produces seating plans in which more than half of guests are lonely. Late-acceptance is in between the two. All trend lines are linear.

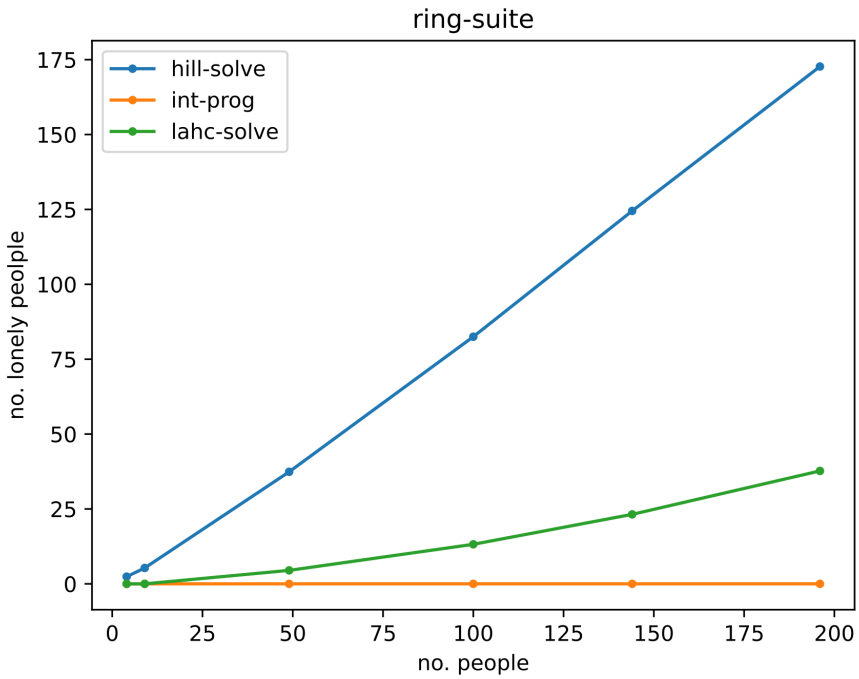


Figure 4.14: Number of lonely people by wedding size. Ring weddings

Ring weddings (figure 4.14) produce a similar loneliness graph to complete weddings, though the poor performance of hill-climbing is even more apparent, with nearly all guests being lonely.

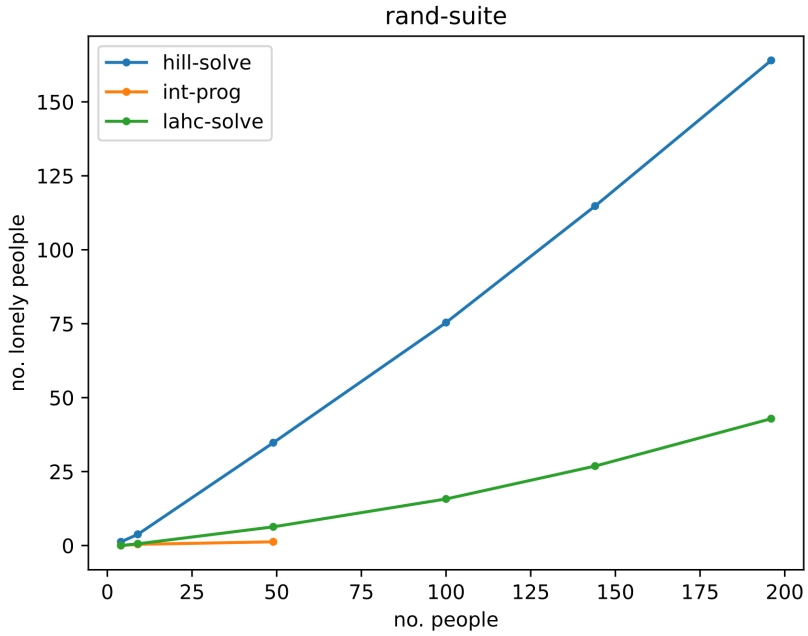


Figure 4.15: Number of lonely people by wedding size. Sparse weddings

The graph for loneliness in sparse weddings (figure 4.15) follows the same general pattern as ring weddings, but integer programming only ran up to weddings of 49 guests.

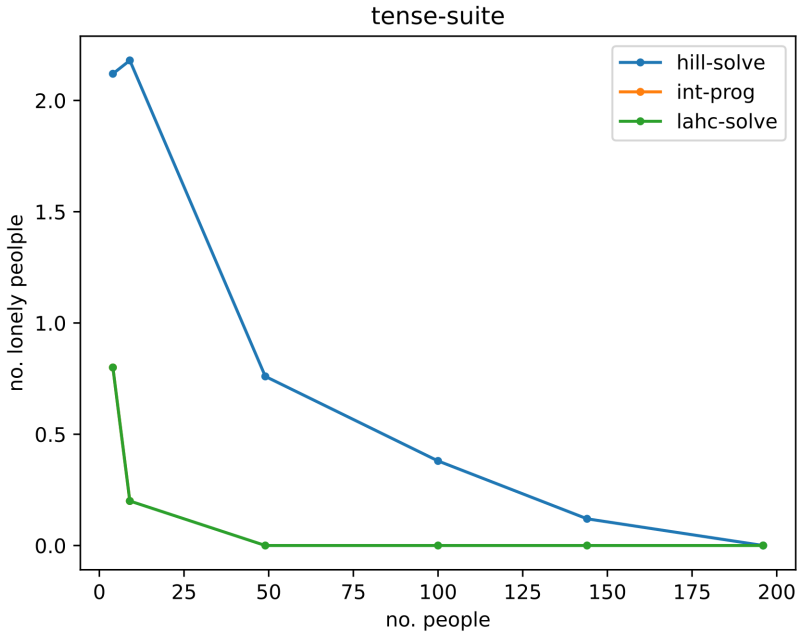


Figure 4.16: Number of lonely people by wedding size. Tense weddings

For tense suites (figure 4.16), integer programming only successfully ran on weddings of 4 and 9 guests. For those data points, the results were identical to those of late-acceptance hill-climbing, so the line is obscured. For these small weddings, late-acceptance hill-climbing produced an average number of lonely guests less than 0.5, meaning that most weddings had no lonely people. For sufficiently-large weddings, this fell to 0 consistently. Naive hill-climbing only achieves this on the largest weddings tested.

4.4 Running Time

Each time a solver was tested on a wedding, the duration was recorded. All durations are plotted in the graphs below - these are not aggregate results. If a dot appears solitary, there may be many others behind it.

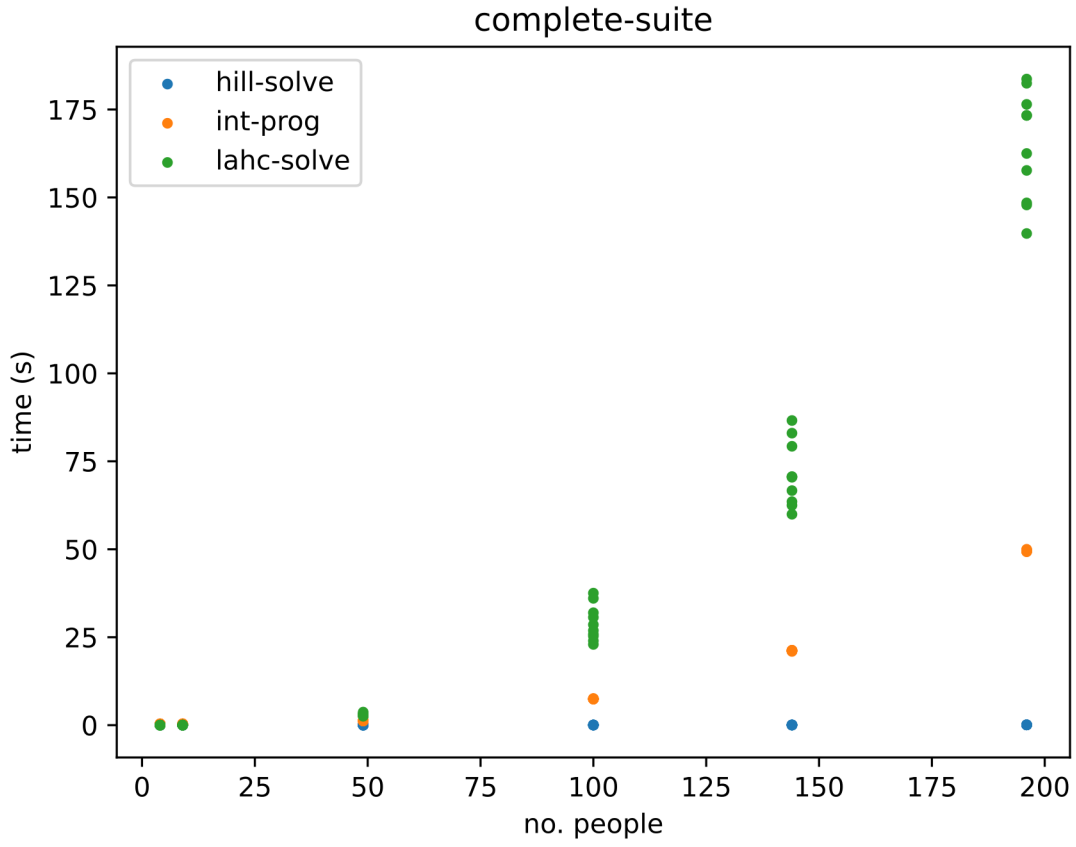


Figure 4.17: Running time for complete weddings

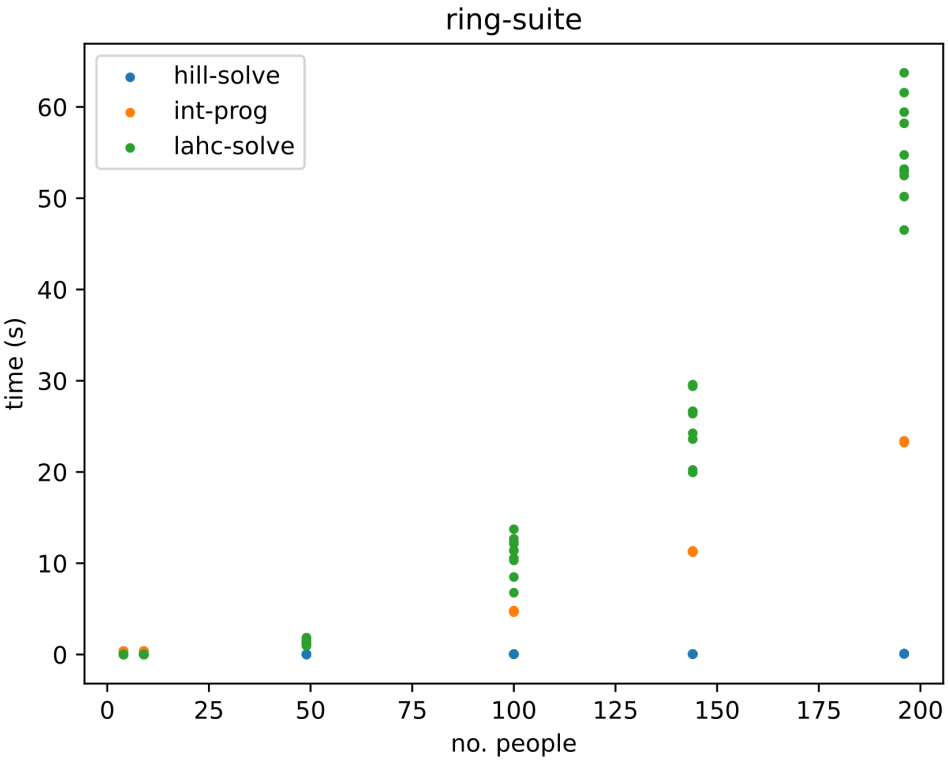


Figure 4.18: Running time for ring weddings

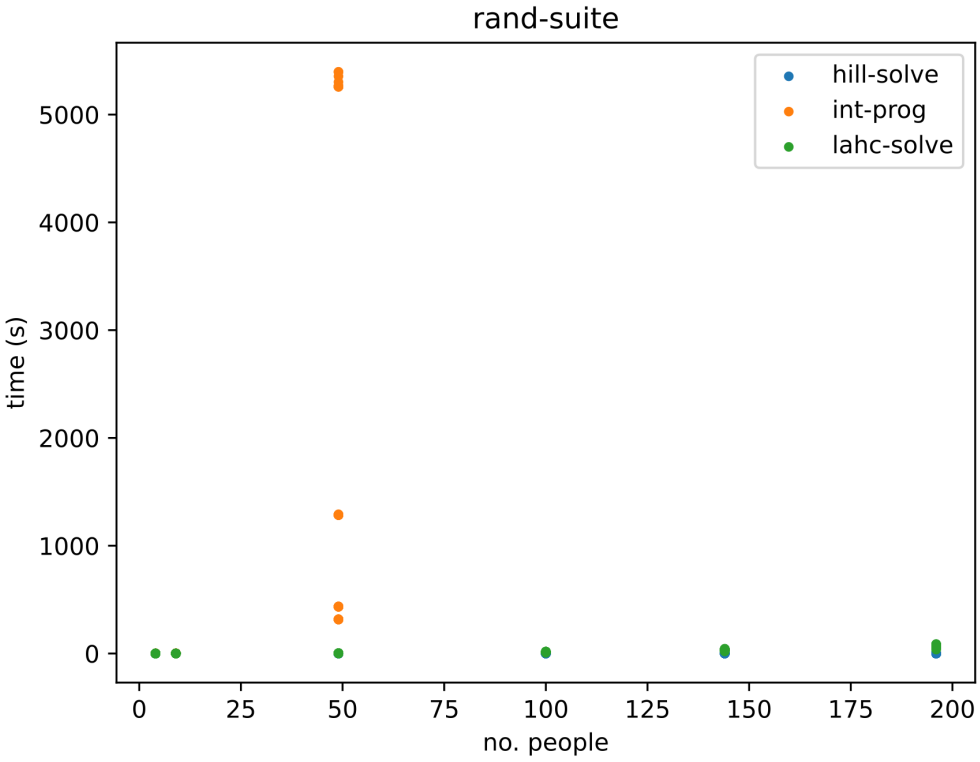


Figure 4.19: Running time for sparse weddings

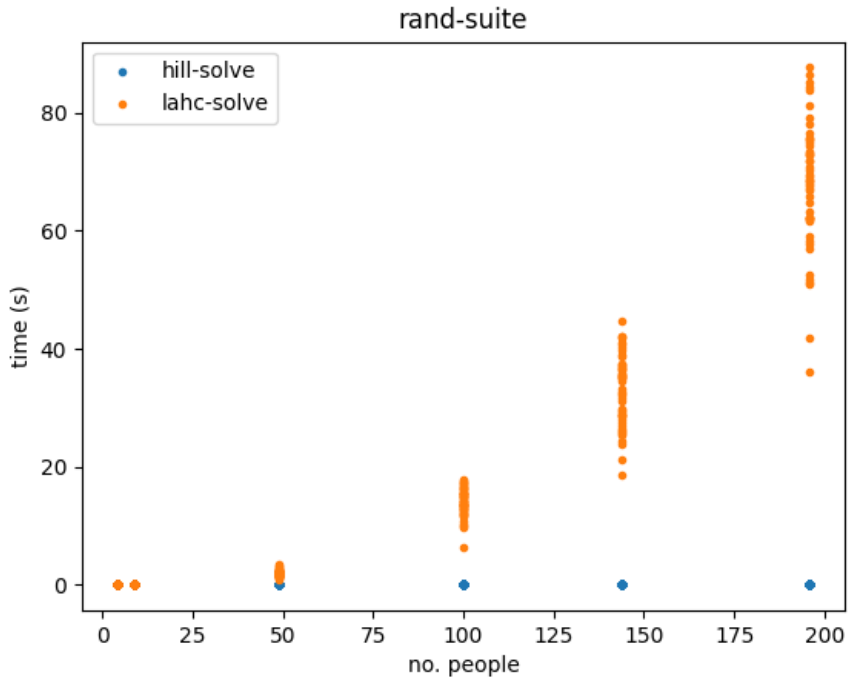


Figure 4.20: Running time for sparse weddings, LAHC and NHC only

Figure 4.20 was added to show the scaling of late-acceptance hill-climbing on sparse weddings, as the presence of integer programming's duration flattens everything else on figure 4.19.

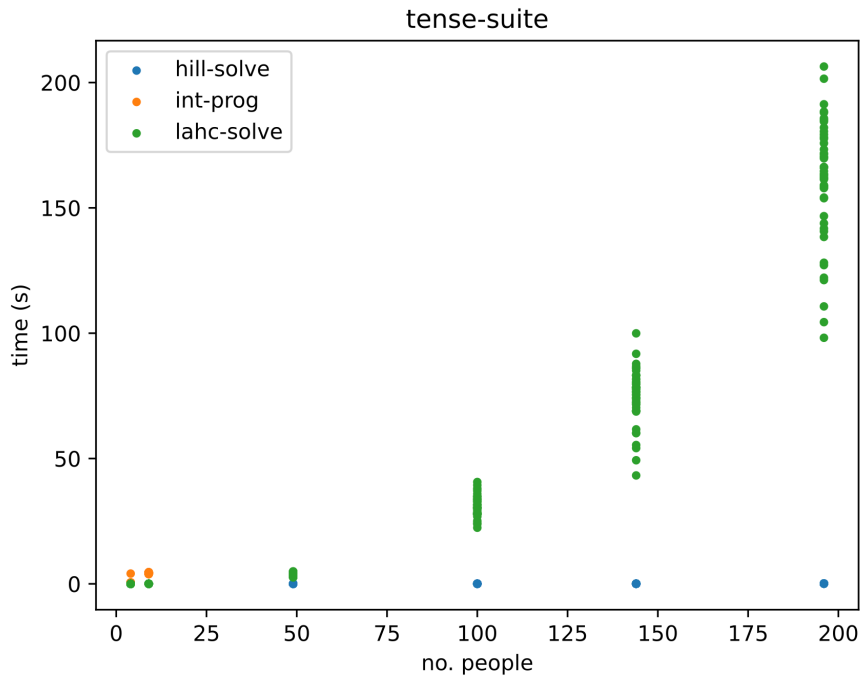


Figure 4.21: Running time for tense weddings

Naive hill-climbing ran in under a second for all wedding types and sizes. Late-acceptance hill-climbing showed significant variation, even when running on the same wedding. For all types of weddings, the run time increases more than linearly. Integer programming, when it terminates, normally follows a similar pattern to late-acceptance hill-climbing, though with much more consistency. It is far more sensitive to the size and details of the wedding, as shown in figure 4.19.

4.5 Significance

Each solver was compared against each of the others on all suites by the metrics of min, median, mean, and max happiness, as well as loneliness and duration using the Mann-Whitney U test. Of the 72 comparisons, only 7 did not show a statistically significant difference at $p < 5\%$. All were between integer programming and late-acceptance hill-climbing.

Suite	Metric	p value
complete	max happiness	0.24
complete	duration	0.15
ring	max happiness	0.50
ring	duration	0.25
sparse	mean happiness	0.14
sparse	median happiness	0.13
sparse	duration	0.21

Figure 4.22: Results which were not statistically significant

The last three concern the sparse suite, in which integer programming could not finish for large weddings. LAHC and IP were evenly matched by max happiness on the ring suite and showed similarity on the complete suite. Though they did not share exact values for duration on those suites, they did show similar scaling with wedding size.

Chapter 5

Conclusions and Future Work

Integer programming provided excellent results, but the computational resources in doing so may render it impractical in some cases. In real life, couples may have several days to spare to allow the computer to work, but it would be nice to have a guarantee of an answer at the end. Late-acceptance hill-climbing provides workable results that are likely to require manual modification, but achieves them in more reasonable time. Naive hill-climbing is not viable to solve the problem.

Naive hill-climbing's failure compared to late-acceptance indicates that the solution-space is full of small local optima, and that moving from one to another requires larger leaps than a single swap. Modifying the mutation method may make hill-climbing more effective. For example, one could use mutations that move large numbers of guests around, or mutations that prioritise the movement of unhappy people.

The success of late-acceptance hill-climbing opens the door to further research using other standard metaheuristics, such as genetic algorithms[13], the duelist algorithm[5] discrete PSO[10], or even other variants of hill-climbing, such as simulated annealing[15]. It is also possible that tweaking the parameters (mutation method, termination condition, ring-buffer capacity) would result in better solutions or faster termination. It is worth noting that these methods can have arbitrary objectives, while integer programming must have a linear function.

One of the problems with optimising for mean/total happiness is that there is no reason to prioritise helping the miserable over the exuberant. Exploring the impact of optimising for minimum or geometric mean happiness instead would be worthwhile but very difficult with integer programming.

There are also countless modifications to the problem itself that simply arise from questioning the assumptions made in this paper. Not all tables are the same size, and the number of tables may be negotiable. It's possible to talk to someone at the table next to yours, while it might be difficult to talk to the person two seats to your right on your own table. One could make different assumptions about what constitutes happiness and adjust the formula appropriately.

Since the precise nature of the data (as opposed to just the size) has quite an impact on the efficiency of late-acceptance hill-climbing and integer programming, testing with other kinds of data, such as that from real weddings, would be informative.

Appendix A

User Manual

A.1 Prerequisites

A unix-like system is required for this project, e.g. Ubuntu 18.04, which was used for development. A complete build of the system should fit within 1GB of storage. 8GB of RAM is recommended for running the code.

The software is written is Python 3.7.5 and Rust stable 1.52.1. If a compatible version of Python is not installed, use your system's package-manager - apt, dnf, or equivalent. To install Rust, follow the instructions at `https://www.rust-lang.org/tools/install`. Due to Rust's backwards compatibility guarantees[6], any later version of also ought to work.

A.2 Installation

Copy the project to a suitable location on your computer. The main directory (henceforth called project directory) should be your working directory when running code unless stated otherwise. As a quick rule, check that `ls` includes `Cargo.toml`.

A.2.1 Python Dependencies

The Python dependency-management relies on a virtual environment. To install, start in the project directory and use the commands

```
cd py
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

A.2.2 Rust Compilation

To build the Rust code, navigate to the project directory and use the command

```
cargo build --release
```

This will download all necessary libraries, build the code and place the executables in `./target/release`

A.3 Using the Software

The main functionalities of the software are as follows: creating seating charts for weddings, creating example weddings, and evaluating the performance of the different planning-heuristics.

A.3.1 Creating a Wedding Problem

Format

Wedding problems are to be specified in the json format. Two things must be given: a relationship matrix containing all the guests' preferences, and the number of tables used in the problem. An example is given below.

```
{
    "relations": {
        "relationships": [
            [0,1,1,1],
            [1,0,0,1],
            [1,0,0,1],
            [1,1,1,0]
        ]
    },
    "n_tables": 2
}
```

Whitespace is not significant. Notice that the `"relations"` field does not have the matrix as its value, but another object with one field holding the matrix. The entry `[i][j]` represents the quality of the relationship between guests i and j.

Invariants

In addition to being parseable, there are additional requirements on wedding problems that may lead to logic errors or crashes of the code if violated.

- Integrality. All numbers must be integers. Decimal points and exponential notation are not allowed, even if the value is actually integral. For example, 4.0 and 2e1 are forbidden - use 4 and 20 instead.
- Squareness. All rows of the matrix must be the same width, and the width of the matrix must be equal to the height.
- Symmetry. The `[i][j]` entry of the matrix must be equal to the `[j][i]` entry.
- Non-narcissism. All diagonal entries of the matrix must be 0.
- Divisibility. The value of the `"n_tables"` field must divide the height of the matrix.

A.3.2 Automatically Creating Wedding Problems

`usage: ./target/release/data-gen <method> <n-tables> <table-size> [output]`

It is not necessary to create all wedding files by hand - there is an executable called `data-gen` which can create weddings of the complete, ring, sparse, or tense structure. The first argument is the type of wedding; the second is the number of tables; the third is the size of each table. The json will then be delivered to standard output. If a fourth argument is specified, it is taken to be the destination file, and output will be printed there instead of stdout.

```
# Create a complete wedding problem with three tables of five.
./target/release/data-gen complete 3 5

# Create a ring wedding with three tables of five.
./target/release/data-gen ring 3 5

 # Create a sparse wedding with two tables of three.
./target/release/data-gen rand 3 2

# Create a tense wedding with four tables of four
# and write to output.txt instead of stdout.
./target/release/data-gen tense 4 4 output.txt
```

A.3.3 Automatically Creating Large Numbers of Weddings

`usage: ./target/release/mass-data-gen <suite>`

The program `mass-data-gen` is a wrapper for `data-gen` that creates large numbers of weddings. It takes a single command-line argument - the path to a text file (called a "suite") containing arguments to `data-gen`. A wedding is created for each line in the suite, and placed in a folder of the same name. For example, say the suite is called "suite.txt" and contains the following.

```
tense 3 4
complete 4 3
```

Then the command

```
./target/release/mass-data-gen suite.txt
```

would produce a directory called `suite` in the parent directory of `suite.txt` that contains a tense wedding and a complete wedding. These wedding files are named according to their specifications, and have a large random number appended to the name to avoid name-collision.

Each line in the suite should only have the **first three** arguments to data-gen.

A.4 Finding Seating Charts

`usage: <solver> < <problem>`

There are three solvers, located in `./py/src/int-prog.py`, `./target/release/lahc-solve`, and `./target/release/hill-solve`. Each is a single executable using a different heuristic to generate their solutions. The problem is read from standard input and the solution is written to standard output. If the code does not appear to run, check that you redirected the problem file and didn't pass it as a parameter.

```
./target/release/hill-solve < my-problem.txt
./target/release/lahc-solve < my-problem.txt
./py/src/int-prog.py < my-problem.txt
```

When running Python, make sure that you are still in the virtual environment set up earlier.

The output is a list of lists of numbers in json format. The numbers represent people, referring to their positions in the input matrix; indexing from 0. (That is, the first person in the list is 0, the second is 1, and so on.) The inner lists represent tables.

For example, the output might be [[3,1],[2,0]], indicating that one table seats guests #3 and #1, and another table seats guests #2 and #6. Different solvers may use whitespace differently, but the output should always be parseable json.

A.5 Evaluating Solvers

`usage: ./target/release/score <path/to/solver> <path/to/problem/directory>`

Since the purpose of this project is to compare different solvers across many different problems, there needs to be a way of running a solver on a large group of problems and evaluating its performance. This is done with the `score` command. It takes a path to a solver and a directory of problems (such as the ones created by mass-data-gen) and spawns the solver as a subprocess to run it ten times on each file in the directory of problems, measuring the quality of the solution. When complete, it summarises the results in a csv file. While running, the software produces a log to stderr showing the statistics as they are produced.

The solver given must be a single executable with no command-line arguments given. `my-python-file.py` is fine if it has execute-permission and begins with `#!path/to/interpreter`, but `"python3 my-python-file.py"` is not.

Each row of the csv file describes one test. The columns are as follows.

- "wedding": the file path of the problem.
- "n_people": the number of guests in the problem

- "n_tables": the number of tables in the problem
- "total_happiness": the sum total of the happiness of all guests.
- "mean_happiness": the mean of the happiness of all guests.
- "median_happiness": the median happiness of all guests.
- "min_happiness": the lowest happiness of any guest.
- "max_happiness": the highest happiness of any guest.
- "n_lonely": the number of guests seated next to no positive relations.
- "seconds": the time it took the solver to find a solution, measured in seconds.

The following shows a worked example:

```
(venv) $ ls
Cargo.lock  Cargo.toml  py  README.md  report  src  suite.txt  target
(venv) $ ./target/release/mass-data-gen suite.txt
(venv) $ ls
Cargo.lock  Cargo.toml  py  README.md  report  src  suite  suite.txt  target
(venv) $ ls suite
tense_003_005_3651957086.txt  tense_004_005_1242002947.txt
(venv) $ ./target/release/score ./target/release/lahc-solve suite
[src/bin/score.rs:184] path_to_solve = "suite/tense_003_005_3651957086.txt"
[src/bin/score.rs:186] &scores = [
    Record {
        wedding: "suite/tense_003_005_3651957086.txt",
        n_people: 15,
        n_tables: 3,
        total_happiness: 932,
        mean_happiness: 62.13333333333333,
        median_happiness: 64.0,
        min_happiness: 18,
        max_happiness: 113,
        n_lonely: 0,
        seconds: 0.035958905,
    },
    Record {
        wedding: "suite/tense_003_005_3651957086.txt",
```

```
        n_people: 15,
        n_tables: 3,
        total_happiness: 872,
        mean_happiness: 58.13333333333333,
        median_happiness: 43.5,
        min_happiness: -8,
        max_happiness: 113,
        n_lonely: 0,
        seconds: 0.009233678,
    },
```

(Debug log continues...)

```
(venv) $ ls
Cargo.lock  Cargo.toml  lahc-solve_suite.csv  py  README.md  report  src  suite
(venv) $ cat lahc-solve_suite.csv
wedding,n_people,n_tables,total_happiness,mean_happiness,median_happiness,
    min_happiness,max_happiness,n_lonely,seconds
suite/tense_003_005_3651957086.txt,15,3,932,62.13333333333333,64.0,18,113,0,
    0.035958905
suite/tense_003_005_3651957086.txt,15,3,872,58.13333333333333,43.5,-8,113,0,
    0.009233678
suite/tense_003_005_3651957086.txt,15,3,876,58.4,68.5,13,116,0,
    0.020586756
suite/tense_003_005_3651957086.txt,15,3,864,57.6,34.5,12,205,0,
    0.021866184
suite/tense_003_005_3651957086.txt,15,3,876,58.4,68.5,13,116,0,
    0.024719352
suite/tense_003_005_3651957086.txt,15,3,876,58.4,68.5,13,116,0,
    0.014198893
suite/tense_003_005_3651957086.txt,15,3,932,62.13333333333333,64.0,18,113,0,
    0.024794527
suite/tense_003_005_3651957086.txt,15,3,932,62.13333333333333,64.0,18,113,0,
    0.024753029
suite/tense_003_005_3651957086.txt,15,3,932,62.13333333333333,64.0,18,113,0,
    0.019821698
suite/tense_003_005_3651957086.txt,15,3,932,62.13333333333333,64.0,18,113,0,
    0.011671174
suite/tense_004_005_1242002947.txt,20,4,1408,70.4,53.0,-14,178,0,
```

```
    0.036196262
suite/tense_004_005_1242002947.txt,20,4,1580,79.0,66.0,-38,189,0,
    0.046546984
suite/tense_004_005_1242002947.txt,20,4,1554,77.7,24.5,1,138,0,
    0.07376284
suite/tense_004_005_1242002947.txt,20,4,1524,76.2,59.5,11,141,0,
    0.04922803
suite/tense_004_005_1242002947.txt,20,4,1720,86.0,57.5,22,191,0,
    0.081084253
suite/tense_004_005_1242002947.txt,20,4,1560,78.0,58.0,-2,197,0,
    0.041920936
suite/tense_004_005_1242002947.txt,20,4,1520,76.0,66.0,-6,151,0,
    0.042735988
suite/tense_004_005_1242002947.txt,20,4,1606,80.3,40.0,22,153,0,
    0.068039512
suite/tense_004_005_1242002947.txt,20,4,1612,80.6,55.0,-18,191,0,
    0.06883209
suite/tense_004_005_1242002947.txt,20,4,1520,76.0,25.5,-12,173,0,
    0.045535036
```

Appendix B

Maintenance Manual

B.1 Installation

See appendix A

B.2 Directory Structure

For development, the important parts of the directory structure are below.

```
Cargo.lock
Cargo.toml
py/
    requirements.txt
    src/
        int-prog.py
    venv/
        ...
src/
    bin/
        data-gen.rs
        hill-solve.rs
        lahc-solve.rs
        mass-data-gen.rs
        score.rs
    hill_climb.rs
    lib.rs
    metrics.rs
target/
    release/
        data-gen
        hill-solve
```

```
            lahc-solve
            mass-data-gen
            score
            ...
        ...
    weddings/
        complete-suite.txt
        hill-solve_complete-suite.csv
        complete-suite/
            complete_002_002_1535681195.txt
            ...
        ...
```

The purpose of each file is given in the following table.

File	Purpose
Cargo.toml, Cargo.lock	Used with the Rust build system.
py/requirements.txt	Holds the list of Python dependencies.
py/src/int-prog.py	A Python executable that solves the dinner-seating problem with integer programming.
py/venv/	Used for the Python virtual environment.
src/lib.rs	Rust library code that defines the interface for solvers.
src/metrics.rs	Rust library code for measuring the quality of charts using different metrics.
src/hill-climb.rs	Rust library code that implements both naive and late-acceptance hill-climbing.
src/bin/*	Rust code compiled to an executable of the same name.
target/release/*	An executable. See the User Manual for usage.
weddings/*.txt	Schematics for creating a directory of weddings.
weddings/*/*.txt	An individual wedding problem.
weddings/*.csv	The results from running a solver on a suite on Maxwell.

Figure B.1: A list of the files used for development.

B.3 Code Structure

`lib.rs` defines an interface for solvers. This is done with the trait (typeclass/abstract class) *SeatingPlanner* and the data-types *GuestRelations*, *Problem* and *Plan*.

The struct *GuestRelations* holds the matrix describing the relationships between all the guests. *Problem* represents the problem in full - both the relationship matrix and the

number of tables available. *Plan* is a type alias for a matrix of array-indices, representing the final seating plan itself.

In order to implement *SeatingPlanner*, a struct must provide a method that takes a *Problem* and returns a *Plan*. Given a seating planner, the function *run*, reads a problem from standard input and writes a solution to standard output.

Many of the metrics used to measure the quality of a chart require the same preprocessing steps. For this reason, the *Metrics* struct is defined in `src/metrics.rs`. Creating an instance does the common work for calculating most metrics, which are calculated on the fly when queried for.

`score.rs` is the source code of the executable responsible for testing solvers on large amounts of data. The *Record* struct represents one row of the csv file outputted. The function *score_path* takes a path argument and decides whether to call *score_single* or *score_suite* based on whether it is a file or directory path. *score_single* does the legwork of spawning the solver on a single problem and *score_suite* iterates through a directory, calling *score_path* on each entry.

`data-gen.rs` can create different categories of weddings - complete, ring, random (sparse) and tense. Each one is associated with a function generates guest relations for the corresponding problem.

B.4 Adding Solvers

Solvers interface with `score` at the process level - any implementation is acceptable so long as it conforms to the json protocol used elsewhere. This is the approach taken of `int-prog.py`. In Rust however, one may simply create a struct that implements *SeatingPlanner* and use the *run* function to handle I/O. Indeed, the *main* functions in `hill-solve.rs` and `lahc-solve.rs` do nothing but instantiate their solvers (*HillClimbingPlanner* and *LahcPlanner* respectively) and pass them along to *run*.

B.5 Known Bugs

- The probabilities used in generation of sparse weddings are not quite the same as they are in the pseudocode, (section 3.5.3) due to the possibility of friend-duplication in the implementation. While this might alter exactly who is friends with whom, the data is still of the correct form.

- Relationships in the generation of tense weddings (section 3.5.4) are not sampled from a true normal distribution, but the sum of twelve uniform distributions[9].

- `int-prog.py` is memory-unsafe. According to `valgrind`, it commits about 8000 memory errors running on the smallest complete wedding in the suite, mostly reads of uninitialised values. Python itself is supposed to be memory-safe, so I believe the problem is with OR-Tools.

- Because of the heavy use of subprocesses with relative paths, all code must be run from the project directory. This is not strictly incorrect, but it is very annoying.

Bibliography

[1] G. G. Abraham Silberschatz, Peter B. Galvin. *Operating System Concepts*. John Wiley & Sons, Inc., 2012.

[2] E. Alba and B. Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 9(2):126–142, 2005.

[3] J. S. Appleby, D. V. Blake, and E. A. Newman. Techniques for Producing School Timetables on a Computer and their Application to other Scheduling Problems. *The Computer Journal*, 3(4):237–245, 01 1961.

[4] M. Bellows and J. L. Peterson. Finding an optimal seating chart. *Annals of Improbable Research*, 2, 2012.

[5] T. R. Biyanto, H. Y. Fibrianto, G. Nugroho, E. Listijorini, T. Budiati, and H. Huda. Duelist algorithm: An algorithm inspired by how duelist improve their capabilities in a duel, 2015.

[6] M. Bos. The plan for the rust 2021 edition. `https://blog.rust-lang.org/2021/05/11/edition-2021.html`, 2021. Accessed: 2021-05-30.

[7] E. K. Burke and Y. Bykov. The late acceptance hill-climbing heuristic. *European Journal of Operational Research*, 258(1):70–78, 2017.

[8] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[9] D. S. (https://stats.stackexchange.com/users/6633/dilip sarwate). How to sample from a normal distribution with known mean and variance using a conventional programming language? Cross Validated. URL:https://stats.stackexchange.com/q/16411 (version: 2011-10-03).

[10] J. Kennedy and R. Eberhart. A discrete binary version of the particle swarm algorithm. In *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*, volume 5, pages 4104–4108 vol.5, 1997.

[11] B. S. McEwen and R. M. Sapolsky. Stress and cognitive function. *Current Opinion in Neurobiology*, 5(2):205–216, 1995.

[12] G. Z. Michele Conforti, Gérard Cornuéjols. *Integer Programming*. Springer International Publishing, 2014.

[13] H. A. L. Omid Bozorg-Haddad, Mohammad Solgi. *Meta-Heuristic and Evolutionary Algorithms for Engineering Optimization*. John Wiley & Sons, Incorporated, 2017.

[14] C. H. Papadimitriou. On the complexity of integer programming. *Journal of the Association for Computing Machinery. Vol 28. No 4*, pages 765–768, 1981.

[15] M. V. S. Kirkpatrick, C.D. Gelatt. Optimization by simulated annealing. *Science, 220*, pages 671–680, 1983.

[16] A. Schijver. *Combinatorial Optimization*. Springer Science & Business Media, 2003.

[17] S. Singh. *The Code Book | The Secret History of Codes and Code-Breaking*. Fourth Estate, Doubleday, 1999.

[18] S. Skiena. *The Algorithm Design Manual*. Springer Science + Business Media, 2008.