UNIVERSIDAD COMPLUTENSE DE MADRID
FACULTAD DE INFORMÁTICA

TRABAJO DE FIN DE GRADO DEL GRADO EN INGENIERÍA INFORMÁTICA BILINGÜE

Complejidad y resolución del problema del K-anonimato

Complexity and resolution of the K-Anonymity problem

Autor: Tomás Moro Lías

Director: Ismael Rodríguez Laguna

September, 2024

Complejidad y resolución del problema del K-anonimato

Este texto ha sido preparado con la plantilla LaTeX de TFG para la UCLM publicada por Jesús Salido en GitHub[1] y Overleaf [2] como parte del curso *«LaTeX esencial para preparación de TFG, Tesis y otros documentos académicos»* impartido en la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

[1] https://github.com/JesusSalido/TFG_ESI_UCLM, DOI: 10.5281/zenodo.4574562
[2] https://www.overleaf.com/latex/templates/plantilla-de-tfg-escuela-superior-de-informatica-uclm/phjgscmfqtsw

Dedicado a: la cebra Camila, el principito, Gerónimo Stilton, Kika Superbruja, el Capitán Calzoncillos, Winston Smith, Ignatius O. Reilly y a los atardeceres que iluminan las vías del tren de Delicias dirección Méndez Álvaro.

Agradecimientos

Quiero expresar mi más sincero agradecimiento a mi tutor, Ismael. Gracias por tu dedicación. Has sido para mí un ejemplo de entrega y profesionalidad al guiarme, con atención al más mínimo detalle, en la realización de este trabajo.

Quisiera también darle las gracias a mis padres y a mi hermano, por haberme apoyado en mi odisea académica a lo largo de los años.

Quisiera agradecer a mis gatos la calma que me han brindado al dormirse a mi lado mientras yo, estresado a más no poder, luchaba contra la cuenta atrás y la resistencia de mi cerebro por aprenderme aquello de lo que el examen del día siguiente trataba.

Quisiera agradecerle a mi entrenador el haberme brindado la intensidad y la sensación de urgencia necesaria por quererme a mí mismo como para dejar resbalar el peso del mundo de mi espalda y cargarme a mí mismo en su lugar.

Finalmente quiero agradecerme a mí mismo el nunca haberme rendido, por haber sido paciente con mis defectos, por crecer, por cambiar, y como el principito; por caminar, conocer y aprender.

Tomás Moro Lías
Madrid, 2024

You have time.
- I. S.

Groucho Marx:
Vamos, Ravelli, ande un poco más rápido.
Chico Marx:
¿Y para qué tanta prisa, jefe? No vamos a ninguna parte.
Groucho Marx:
En ese caso, corramos y acabemos de una vez con esto.
- Los Hermanos Marx.

¡Que guapo estás sin las gafas!
- Iván Era

No te puedes agradecer a tí mismo en los agradecimientos.
- Mamá

TRABAJO DE FIN DE GRADO - FACULTAD DE INFORMÁTICA (UCM)

Estudio del $K-$Anonimato

Tomás Moro Lías
Madrid, September 2024

Resumen

Hay muchas situaciones en las que el anonimato es importante, como por ejemplo en los registros médicos o en internet. Por ello, los datos que pueden identificar a una persona de manera inequívoca, en la mayoría de los casos, no se recopilan. Sin embargo, los datos necesarios para proporcionar los servicios sí deben ser recolectados y almacenados. Existen situaciones en las que la recolección de esta información puede identificar a alguien.

El K-Anonimato es un problema de ingeniería informática y matemáticas que implica distribuir en un cierto número de lugares un cierto número de elementos, cada uno identificable por algunos *cuasi-identificadores*, de la manera menos identificable posible en términos de grupo.

Por ejemplo, los lugares pueden ser diferentes consultorios de cirujanos y los elementos pueden ser personas que necesitan quimioterapia para su tratamiento contra el cáncer, por lo que sus identificadores pueden ser el código postal, el sexo o el tipo de sangre, entre otros.

Entendiendo el anonimato entre estas personas como tener los mismos cuasi-identificadores para que sean indistinguibles, la mejor solución al problema es la que singulariza la menor cantidad de pacientes. Esto se debe a que, si en un día se tratan 30 pacientes, y solo 1 paciente vive en una cierta área de código postal, al revisar el registro del día, esta persona no será anónima en absoluto. Si esta persona es de interés y se conoce su código postal, se filtrará la información.

En este estudio del problema determinaremos la complejidad computacional así como diferentes formas de resolverlo, como el uso de un algoritmo exhaustivo, un algoritmo genético y un enfoque basado en programación por restricciones.

Palabras clave: $K-$Anonimato, Programación por restricciones, Problema de partición, Orden de complejidad.

BACHELOR DISSERTATION - IT FACULTY (UCM)

$K-$Anonymity Problem Study

Tomás Moro Lías
Madrid, September 2024

Abstract

There are many situations in which anonymity is important. For example in medical records or the internet. This is why data that can identify a person unambiguously in most cases is not collected. Data necessary for providing the services, though, must be collected and recorded. There can be situations where collecting this information can identify someone.

The K-Anonymity is a computer engineering and mathematical problem which entails distributing in a certain number of places a certain number of items, each identifiable by some *quasi-identifiers*, in the least identifiable way possible group-wise.

For example, the places can be different surgeon's offices and the items can be people that need chemotherapy for their cancer treatment, so their identifiers can be ZIP code, sex or blood type, among others.

Understanding anonymity among these people as having close to identical quasi-identifiers so they are indistinguishable, the best solution to the problem is the one that singles out the least amount of patients. This is because, if in a day 30 patients are treated, and only 1 patient lives in a certain ZIP code area, and the record of the day is looked at, this person will not be anonymous at all. If this is a person of interest and its ZIP code is known, the result will be leaked.

In this problem study we will ascertain the computational complexity as well as different ways of solving it, such as using an exhaustive algorithm, a genetic algorithm, and a constraint programming-based approach.

Key words: $K-$anonymity, Constraint programming, Partition Problem, Complexity order.

Notation and Acronyms

NOTATION

Notation or nomenclature used in the final degree project's memory.

$\mathcal{S}, \mathcal{P}, \mathcal{O}, ...$	:	Sets or Multisets, depending on context
$\cup, \cap, \setminus, \times$	:	Set union, intersection, difference, and Cartesian product
$X, A, L, ..., r, g, l, ..., \epsilon, \gamma, \phi, \delta, ...$	:	Logical variables
f, g, h	:	Logical functions
$\cdot$	:	Arithmetical product
$+$	:	Arithmetic sum
$-$	:	Arithmetic subtraction

LIST OF ACRONYMS

List of acronyms used in the final degree project's memory.

BFS	:	Breadth-First Search
BnB	:	Branch and Bound
DFS	:	Depth-First Search
GAs	:	Genetic Algorithms
Noop	:	No-operation, an instruction that does nothing
TFG	:	Trabajo Fin de Grado
TSP	:	Travelling Salesman Problem

Contents

CHAPTER 1
Introduction

1.1. CONTEXT

Nowadays, most organizations are facing data accumulation in massive amounts and storing them in large databases. Public and private institutions deal with data that characterize individual entities. In particular, the healthcare industry has recognized the potential use of these data to make informed decisions. However, these personal data can be used by other unauthorized institutions or people, so there is an increasing concern about the dissemination of confidential information. In fact, data from electronic health record systems are prone to privacy violations, especially when stored in healthcare medical servers.

However, there are some data that are generated with the proper permission of people involved, such as those due to a screening program, and are a good starting point for further medical research. So the question is: how can a data holder (a bank, a hospital, an university) protect the identity of individual entities by constructing a new version of the data that can be disclosed for further studies o research in such a way that people are not going to be re-identified from them?

Since the late of last century, several disclosure control methods have been proposed in order to protect the identity of individuals from the microdata. Data anonymization methods provide a structured approach to modifying data for privacy protection. One of them is K-anonymity, which lets us suppose that identifying variables such as name, address, phone number, and social security number have been removed. Then K-Anonymity consists in anonymizing data by hiding the individual record in a group of similar records, thus significantly reducing the possibility that the individual can be identified. For instance, a set of variables such as gender, age, ZIP code can be used as quasi-identifiers in order to anonymize the data. The datasets are said to be k-anonymous only when any single row is identical to a minimum of $K - 1$ rows. In other words, K-anonymity is reached when all the records in a set of quasi-identifiers (QI) are indistinguishable from at least $K - 1$ other records in the data set [1]. Therefore, K-anonymity can be used to prevent database linkages.

When dealing with datasets, it is mandatory to consider which fields or attributes contain sensitive, identifiable or quasi P-identifiable information in order to obfuscate or conceal any sensitive data about an individual, thus limiting the person's re-identification. However, some combinations of these attributes may turn to be dangerous for the privacy and end in a leak in sensitive data that leads to lawsuits, loss of customer confidence and bad press, as anybody can make an unethical use of this information.

In this project, we are concerned with the problem of anonymizing medical databases that emerge out of screening programs. These programs consider appointment data, which are attributes that become part of the quasi-identifiers and, hence, an additional risk factor for the anonymity. There are several techniques for anonymizing a database a *posteriori*. In this project we are going to follow the strategy defined in [2], where the desired level of anonymity is considered when setting up the appointments, thus building up the database.

1.2. PROJECT GOALS

In this project, we will attempt to achieve the following goals:

1. Study and analyze the computational complexity of the $K-$Anonymity problem as defined in article [2].
2. Solve the problem stated in that article via different algorithms and paradigms:
   a) Exhaustive algorithm.
   b) Genetic algorithm.
   c) Constraint programming.
3. Study and analyze the effectiveness of the solutions of each algorithm, and in the case of GAs, study the differences in performance between configurations.

1.3. DOCUMENT STRUCTURE

The document is organized as follows: in Chapter 2 we will discuss the state of the art regarding $K-$Anonymity, as well as the historical context in section 2.1. Then, section 2.2 we will introduce the minimum mathematical machinery required to establish the foundations of the complexity class of a problem. These include the classification of problems and what a problem reduction is.

Once the mathematical foundations are established, Chapter 3 describes formally the $K-$Anonymity problem, and explores how this problem is related with the partition problem, which is relevant for its classification, which is developed in section 3.2. Chapter 3 concludes establishing that the $K-$Anonymity problem belongs to NP-Complete class. Chapters 4, 5 and 6 address possible ways to solve the problem based on different algorithms, using exhaustive and non-exhaustive approaches. We will study the different performances achieved with these different methods. Finally, in Chapter 7 we summarize the goals achieved, difficulties encountered, skills acquired or developed and discuss the obtained results, and briefly sketch some possible lines for future work.

All of the files used and needed to make this work are included in [3].

CHAPTER 2
State of Art

2.1. HISTORICAL BACKGROUND

Large amount of person-specific data has been collected in recent years by both governments and private entities. In fact, data and knowledge extracted by data mining techniques represent a key asset to the society, because they are used as a tool for analyzing trends and patterns or formulating public policies. Moreover, laws and regulations require that some collected data must be made public, for example, Census data. That is why governmental, public, and private institutions that systematically release data are increasingly concerned with possible misuses of their data that might lead to disclosure of confidential information (see, for instance, [4]).

Disclosure Control is the discipline concerned with the modification of confidential information about individual entities such as persons, households, businesses, etc. in order to prevent third parties working with these data from recognizing individuals in the data, and, thereby, disclosing information about these individuals. The initial data is called microdata, and it represents a series of records, each one of them containing information on an individual unit such as a person, a firm, an institution, etc. Microdata can be represented as a single data matrix where the rows correspond to the units (individual units) and the columns to the attributes (as name, address, income, sex, etc.). Before being released, initial microdata is masked by the data owner, that is, the data are, somehow, anonymized. Usually, "personally identifying information" (PII), such as Name, Social Security number, phone number, email or address is removed. In short, anything that identifies the person directly. But this procedure may not be enough to guarantee anonymity. There are attacks that consist, basically, on crossing information of different sources that are public, such as census or voter list, like the example showed in Table 2.1. It is based on a famous case of data privacy breaching occurred at the beginning of 2000, which is mentioned and studied in detail by Latanya Sweenwy in [5]. This attack is called re-identification by linking.

The information that conforms the initial microdata can be classified as (see [6]):

- *Key attributes or Identifiers*: Name, address, phone number, etc., that is, data that are uniquely identifying and, therefore, are always removed before release. Those attributes are present only in initial microdata because express information which can lead to a specific entity.
- *Quasi-identifiers*: 5-digit ZIP code, birth date, gender. These data together uniquely identify 87% of the population in the U.S. They can be used for linking anonymized dataset with other datasets. These attributes are present in masked or released microdata as well as in the initial microdata.
- *Sensitive or Confidential attributes*: Medical records, salaries, etc. These attributes are the data that researchers deal with, so they are always released directly. Sensitive attributes are present in released microdata as well as in the initial microdata.

Medical Data Released as Anonymous

SSN	Name	Ethnicity	Date of Birth	Sex	ZIP	Marital Status	Problem
...	...	Asian	09/21/1963	female	02139	Divorced	hypertension
...	...	Asian	09/12/1963	female	02139	Divorced	heart disease
...	...	Asian	05/07/1963	female	02139	Married	Broken Arm
...	...	Asian	05/07/1963	female	02139	Married	Broken Arm
...	...	black	05/07/1964	female	02139	Married	obesity
...	...	black	05/17/1964	male	02138	Married	hypertension
...	...	white	04/30/1965	male	02142	married	cancer
...	...	white	7/17/1964	female	02142	widow	obesity
...	...	...	...	...	...	...	...

Voter List

Name	Address	City	ZIP	DOB	Sex	Party	...
...	...	...	...	...	...	...	...
Smith, John	1234 Sunny Street	Boston	02142	04/30/1965	male	Democrat	...
...	...	...	...	...	...	...	...

Table 2.1: Identifying anonymous data by linking to external data

The major goal of disclosure control for microdata is to protect the confidentiality of the individuals, that is, to ensure that released data are sufficiently anonymous taking into account two requirements: existing regulations are complied and information loss for statistical inference is minimized. The data owner must select a compromise between disclosure risk and information loss values because completely fulfilling both measures is not possible: decreasing disclosure risk will usually lead to increase information loss and vice versa (see [5]).

Several disclosure control techniques were proposed in the literature. Two of the most used techniques are generalization and suppression. Both are at the background of the anonymity process, as we will see next.

2.1.1. $K-$Anonymity model. First approach

The idea behind the $K-$Anonymity model —whose main target is to make it hard to link sensitive and insensitive attributes— is that the information for each person contained in the released table cannot be distinguished from at least $K-1$ other individuals whose information also appears in the release. In other words, each record is indistinguishable from at least $K-1$ other records. These blocks of at least K elements define a partition of the records, that is, define an equivalence relation and each block forms an equivalence class. In sum, rows are "clustered" (partitioned) into sets of size at least K. There are, mainly, two strategies to improve K level (see, for instance, [6], [2]):

- Generalization: replace quasi-identifiers with less specific, but semantically consistent values until K identical values are obtained and partition ordered-value domains into intervals.
- Suppression: when generalization causes too much information loss then the quasi-identifier is omitted, not released at all. This is common with outliers.

Getting into the subject, the problem posed in the present work is related with *screening programs* and was proposed by Rafael Caballero, Sagar Sen and Jan Nygard in [2]. In these programs people are invited to take a screening test without any medical symptoms, just as a preventive medical policy. The screening programs have different algorithms for assign people for appropriate characteristics to a screening center. Adding the appointment attribute to a database can modify the risk of being re-identified. Our case of study is concerned with this question:

> *Can we anticipate the risk of a person of being re-identified in advance and modify the screening process such that the screening database is anonymized by construction?*

For a better understanding of the problem, a description of the process is made by using an example. Let us consider the population displayed in Table 2.2

SSN	Name	Address	Ethnicity	D.O.B.	Sex	ZIP	Mar. Stat.	Age
...	...	1, Cantor Ave.	Asian	11/21/1962	male	02138	Married	60-65
...	...	14, Perlman St.	Asian	09/12/1963	female	02139	Divorced	60-65
...	...	75 Gödel Ave.	Asian	09/12/1963	female	02139	Divorced	60-65
...	...	12, Lovelace St.	Asian	05/07/1963	female	02139	Married	60-65
...	...	65, Kruskal Rd.	Asian	05/07/1963	female	02139	Married	60-65
...	...	5, Gauss St.	Asian	7/17/1964	female	02142	Widow	55-60
...	...	19, Erdoz Ave.	Asian	11/21/1965	male	02138	Married	55-60
...	...	9, Von Newman St.	Black	05/07/1966	female	02139	Married	55-60
...	...	15, Prim Ave.	Black	05/17/1964	male	02138	Married	55-60
...	...	3, Galileo St.	Black	08/16/1962	male	02138	Married	60-65
...	...	5, Noether St.	Black	02/11/1966	female	02142	Widow	55-60
...	...	81, Dijkstra Rd.	White	04/30/1965	male	02142	Married	55-60
...	...	5, Lamarr St.	White	7/17/1964	female	02142	Widow	55-60
...	...	43, Hamilton Rd.	White	11/08/1966	male	02138	Divorced	55-60
...	...	19, Turing Ave.	White	03/23/1965	female	02139	Married	55-60
...	...	19, Whitehead Rd.	White	02/27/1965	male	02141	Divorced	55-60

Table 2.2: Dataset sample I

The initial dataset contains the Social Security Number, full name, address, ZIP code, date of birth, marital status and the age range. The first three attributes are identifiers. The rest of them are quasi-identifiers. The first step consists in separating the identifiers and replacing them by an internal code, for example, the number of the row they are, getting Table 2.3. This new table is the one which is interesting for researchers, demographics studies and so on. In this table it is easy to check that only in the case that we are considering an Asian woman in the sixties, this person will not be unequivocally identified, because there is another one with the same attributes. These two groups are colored in the table. The rest of the people is still identified, even though their full names and addresses were removed.

Using suppression, that is, getting rid of marital status and date of birth, we get Table 2.4, where there is one cluster with four people. When Ethnicity is also suppressed then the number of clusters increases, as it is showed in Table 2.5, i.e., all groups are composed at least two indistinguishable individuals.

	Ethnicity	D.O.B.	Sex	ZIP	Marital Stat.	Age
1	Asian	11/21/1962	male	02138	Married	60-65
2	Asian	09/12/1963	female	02139	Divorced	60-65
3	Asian	09/12/1963	female	02139	Divorced	60-65
4	Asian	05/07/1963	female	02139	Married	60-65
5	Asian	05/07/1963	female	02139	Married	60-65
6	Asian	7/17/1964	female	02142	Widow	55-60
7	Asian	11/21/1965	male	02138	Married	55-60
8	Black	05/07/1966	female	02139	Married	55-60
9	Black	05/17/1964	male	02138	Married	55-60
10	Black	08/16/1962	male	02138	Married	60-65
11	Black	02/11/1966	female	02142	Widow	55-60
12	White	04/30/1965	male	02142	Married	55-60
13	White	7/17/1964	female	02142	Widow	55-60
14	White	11/08/1966	male	02138	Divorced	55-60
15	White	03/23/1965	female	02139	Married	55-60
16	White	02/27/1965	male	02141	Divorced	55-60

Table 2.3: Dataset sample II

	Ethnic.	Sex	ZIP	Age
1	Asian	male	02138	60-65
2	Asian	female	02139	60-65
3	Asian	female	02139	60-65
4	Asian	female	02139	60-65
5	Asian	female	02139	60-65
6	Asian	female	02142	55-60
7	Asian	male	02138	55-60
8	Black	female	02139	55-60
9	Black	male	02138	55-60
10	Black	male	02138	60-65
11	Black	female	02142	55-60
12	White	male	02142	55-60
13	White	female	02142	55-60
14	White	male	02138	55-60
15	White	female	02139	55-60
16	White	male	02141	55-60

Table 2.4: Dataset sample III (a)

	Sex	ZIP	Age
1	male	02138	60-65
2	female	02139	60-65
3	female	02139	60-65
4	female	02139	60-65
5	female	02139	60-65
6	female	02142	55-60
7	male	02138	55-60
8	female	02139	55-60
9	male	02138	55-60
10	male	02138	60-65
11	female	02142	55-60
12	male	02142	55-60
13	female	02142	55-60
14	male	02138	55-60
15	female	02139	55-60
16	male	02141	55-60

Table 2.5: Dataset sample III (b)

Records 12 and 16 are still unique in their own class. Now the generalization technique can be used, that is, replacing quasi-identifiers with less specific, but semantically consistent values until we obtain K identical values. In this case, omitting the last digit of the ZIP code increases the K value as it is showed at Table 2.6. At this point, the achieved level of anonymity is $K = 2$. In fact, there are: one group of 4 people, two groups of 3 and three groups of 2 people.

	Sex	ZIP	Age
1	male	02138	60-65
2	female	02139	60-65
3	female	02139	60-65
4	female	02139	60-65
5	female	02139	60-65
6	female	02142	55-60
7	male	02138	55-60
8	female	02139	55-60
9	male	02138	55-60
10	male	02138	60-65
11	female	02142	55-60
12	male	0214*	55-60
13	female	02142	55-60
14	male	02138	55-60
15	female	02139	55-60
16	male	0214*	55-60

Table 2.6: Dataset sample IV

2.1.2. Including appointment data

The selected population is, now, assigned to an appointment according to the screening program. For that purpose, there is a collection of hospitals. Each one of them offer some time slots, as well as their own capacity, that is, the maximum amount of people that can be attended in each slot.

Let us assume that there are three centres for the appointments (A, B and C) whose availabilities are the following:

- Centre A admits five people at 9am and three people at 11am.
- Centre B admits three people at 9am and two people at 11am.
- Centre C admits five people at 9am and five people at 11am.

The appointment (centre and hour) can be considered part of the quasi-identifier, so two new columns can be added to the Table 2.6. This information, not to mention the result of the screening program, can create a security risk due to a diminish of the K level of anonymity.

In this work, based on [2], we are concerned with the screening procedure and the measures for increasing anonymity when assigning people to screening rooms. Here we introduce the concept of the value of the quasi-identifiers. A value of the quasi-identifiers is a combination of quasi-identifiers values that identify a person in our dataset. In other words, a possible row of our dataset. For short, we will refer to them as quasi-identifiers. In 2.6, a possible value of the quasi-identifiers can be: **[male, 02138, 55-60]**. With this notion, let us formalize the idea of $K-$anonymity and introduce some notation.

Definition. *[$K-$anonymity] Let T be a table with N rows ($N \geq 0$) whose columns are the values of some attributes that conform quasi-identifiers. Let Q be the number of different values that the quasi-identifiers take in T, and let $v_1, \ldots, v_Q$ be these values. Let q_i be the frequency of v_i. In these conditions, we say that T verifies $K-$anonymity (or that the level of anonymity of T is K) if K is:*

$$K = \min\{q_i \mid 1 \leq i \leq Q\}$$

Remark. *Observe that* $\sum_{i=1}^{Q} q_i = N$.

Example. *Consider Table 2.6. In this case:*

- $N = 16$, *the number of rows (population's size) and* $Q = 6$, *the number of different values of* T.
- $v_1 = [male,\ 02138,\ 60-65]$ *and* $q_1 = 2$ *(see that* $v_1 = v_{10}$*).*
- $v_2 = [female,\ 02139,\ 60-65]$ *and* $q_2 = 4$.
- $v_3 = [female,\ 02142,\ 55-60]$ *and* $q_3 = 3$.
- $v_4 = [male,\ 02138,\ 55-60]$ *and* $q_4 = 3$.
- $v_5 = [female,\ 02139,\ 55-60]$ *and* $q_5 = 2$.
- $v_6 = [male,\ 0214*,\ 55-60]$ *and* $q_6 = 2$.

So, $K = \min\{q_1, \ldots, q_6\} = 2$ *is the level of anonymity of this table or, more formally, we say it verifies 2-anonymity.*

Definition (Anonymity vector)**.** *Let T, N, Q, v_i, q_i be as in the previous Definition. We define:*

- *The number of different quasi-identifiers values with j repetitions in T is denoted by $K(j)$, formally:*

$$K(j) = |\{v_i \mid q_i = j,\ 1 \leq i \leq Q\}|$$

- *The anonymity vector of T is the vector that counts the number of quasi-identifiers that are repeated once, two, three and so on, ordered in increasing order*

$$avector(T) = (K(1),\ K(2),\ \ldots)$$

Remark. *Observe that*

- *The minimum size of this vector is* 1*, the case in which every quasi-identifier is single, where we have* $avector(T) = (K(1)) = (N)$*. On the contrary, the maximum size of this vector is* N*, which is the case that all quasi-identifiers are in the same class, where we have* $avector(T) = (0, \dots, K(N)) = (\underbrace{0, \dots, 0}_{N-1}, 1)$*.*
- *The number of rows of* T *can be obtained from any* $avector(T)$ *as follows:* $\sum_i K(i) \cdot i = N$*.*
- *The* $K-$*anonymity of the population represented by* $avector(T)$ *corresponds to the position of its leftmost non-zero element.*

Example. *The anonymity vector of Table 2.6 is* $avector(T) = (0, 3, 2, 1)$*. It can be easily checked that* $3 \cdot 2 + 2 \cdot 3 + 1 \cdot 4 = 16 = N$*.*

The concept of anonymity vector allows to compare two different released databases and to establish some kind of measure of the "grade of anonymity", that is, some kind of tool to know if two tables are equally anonymous.

Definition (Generalized $K-$anonymity)**.** *Let* T_1 *and* T_2 *two tables of size* N*:*

- T_1 *has the same generalized* $K-$*anonymity as* T_2*, and we denote it* $anon_k(T_1) = anon_k(T_2)$ *if* $avector(T_1) = avector(T_2)$*.*
- T_1 *has better generalized* $K-$*anonymity than* T_2*, and we denote it* $anon_k(T_1) > anon_k(T_2)$*, if* $avector(T_1) <_{LEX} avector(T_2)$*, where* $<_{LEX}$ *is the lexicographical order.*

Example. *Let* $T_1, T_2, T_3, T_4 \dots$ *be tables with* $N = 16$ *rows and assume the following are their four anonymity vectors associated to their respective quasi-identifiers, ordered in lexicographical order:*

$$\underbrace{(0,0,1,2,1)}_{avector(T_1)} <_{LEX} \underbrace{(0,4,0,2)}_{avector(T_2)} <_{LEX} \underbrace{(0,4,1,0,1)}_{avector(T_3)} <_{LEX} \underbrace{(1,4,1,1)}_{avector(T_4)}$$

It can be seen that:

- *In the first table,* T_1*, there is one cluster with 3 people, two clusters of 4 people and a group of 5 people. The total number is* $N = 1 \cdot 3 + 2 \cdot 4 + 1 \cdot 5 = 16$*. In general, if* v *is the anonymity vector, then* $N = \sum_{i=1} v[i] \cdot i$*.*
- *We get that:*
$$anon_K(T_1) > anon_K(T_2) > anon_K(T_3) > anon_K(T_4)$$
- *About* K *anonymity, we observe that* T_1 *verifies 3 anonymity,* T_2 *and* T_3 *verifies 2 anonymity, and* T_1 *verifies 1 anonymity. So, this example illustrates that the greater the anonymity vector is (in lexicographical order), the smaller value the anonymity verifies.*

Summarizing, any anonymity vector contains not only information about the size of the population, but also the level of anonymity. Moreover, it gives a picture about the distribution of the population in the clusters. In other words, this is a refinement of the concept of anonymity. And this is how we arrive to a very important and beautiful question:

How many anonymity vectors exist for a population of quasi-identifier values of size N*?*

Obviously, there are as many vectors as ways of decomposing a natural number as a sum of positive numbers. But populations are not immutable, people change age groups as they age, they develop new symptoms, they move out of their homes and change localities... Or in the case that is most interesting to us: they get screened in a certain room. When the room in which a person is screened is appended to the database, rows with equal quasi-identifier values may differ in the column relative to the room where they were screened. This leads us to a follow-up question:

How do I keep the highest anonymity level in a population where I want to add a quasi-identifier column, with some values, to every element? In other words, how can I achieve the lowest lexicographical order in the anonymity vector of a dataset when appending one new quasi-identifier?

This is the general idea of the problem we will attempt to solve: Given some population with some quasi-identifier values and some rooms with certain capacities, how to assign each person to some screening room while ensuring the highest level of anonymity possible in the resulting population. Before we get into it, we need to introduce some concepts and definitions.

2.2. PROBLEMS AND COMPLEXITY

2.2.1. Decision Problems

Decision problems, informally, are those that are described by a statement and a question which requires a YES or NO answer (see, for instance, [7]). In other words, it is a yes-or-no question on an infinite set of inputs. It is traditional to define the decision problem as the set of possible inputs together with the set of inputs for which the answer is yes. They can be classified by their complexity, in terms of the time to solve them. In this classification there are two classes of great interest, **P** and **NP**.

Definition (Classification of problems)**.** *Decision problems can be classified as follows:*

1. **P *class***. *It is the class in which all problems are solvable in polynomial time by a deterministic Turing machine.*
2. ***Problem Reduction***. *A reduction in polynomial time is a transformation of one decision problem into another problem preserving the capability of the transformed problem to obtain the solution of the original problem in polynomial time. This transformation must be computable in polynomial time and preserve the answer to the original problem.*

   In other words, if a problem A can be reduced to another problem Q in polynomial time, then Q is at least as hard as A, and any algorithm that can solve Q in polynomial time can also solve A in polynomial time. This is important because proving that a solution for Q can be found in polynomial time, and that is also a solution for A, proves A to be polynomial.

   It can be proven through a function $f : \{0, 1\}^* \to \{0, 1\}^*$, such that for any discretized input instance $x \in \{0, 1\}^*$ of problem A, the answer to x in A is the same as the answer to $f(x)$ in Q.

   The function f must fulfil the following conditions:

   a) *There exists an algorithm that can compute $f(x)$ for any $x \in \{0, 1\}^*$ in time bounded by a polynomial in the size of the input x.*

   b) *For any $x \in \{0, 1\}^*$, x is a YES instance of A if and only if $f(x)$ is a YES instance of Q.*
3. **NP *class***. *It is the class in which all problems are solvable in polynomial time by a non-deterministic Turing machine.*

   Remark. *Belonging to* **NP***:*

   - *A decision problem is in the class* **NP** *if, given a "yes" instance of the problem, there exists a certificate (or witness) such that the correctness of this certificate can be verified in polynomial time by a deterministic Turing machine.*
4. **NP-*hard class***. *A decision problem h is* **NP***-hard if every problem $p \in$* **NP** *can be reduced in polynomial time to h.*
5. **NP-*complete class***. *A problem p is* **NP***-complete if it belongs to* **NP** *and is also* **NP***-hard.*
6. ***Inaproximability of a problem***. *The inapproximability of an NP-hard problem refers to the inherent difficulty of finding a solution that approximates the optimal solution within a certain performance ratio. In many cases, NP-hard problems are known to be difficult to approximate*

efficiently, meaning there is no polynomial-time algorithm that can provide a solution within a specified approximation ratio unless P equals NP, where

$$P_{ratio} = \frac{quality\ of\ the\ optimal\ solution}{quality\ of\ the\ returned\ solution}$$

Remark. *We observe that:*

- $\mathbf{P} \subseteq \mathbf{NP}$.
- *If a problem p_1 which does not belong to* $\mathbf{P}$ *can be reduced in polynomial time to another problem p_2, then we can conclude that p_2 does not belong to* $\mathbf{P}$ *class either. We can see that, if p_2 belonged to* $\mathbf{P}$*, then p_1 could be solved with an algorithm of polynomial time complexity by transforming it using a polynomial reduction to p_2 and then solving p_2 in polynomial time (which could be done by definition, for it belongs to* $\mathbf{P}$*).*
- *Finding a polynomial time algorithm to solve any NP-hard problem would lead a way to solve all the NP-complete problems in polynomial time too, which would imply* $\mathbf{P} = \mathbf{NP}$.

CHAPTER 3
The Problem

In this chapter we will describe formally the problem instances for $K-$Anonymity and we will study its complexity.

3.1. K-ANONYMITY PROBLEM INTRODUCTION

As we have already mentioned, the $K-$Anonymity Problem deals with the perceived anonymity of the observer of a dataset. The goal of section 3.1 is to provide a formal framework for understanding the problem and, later on, constructing and evaluating algorithms and systems that release information in a way such that the released information limits what can be revealed.

Let D be a dataset with n records, where each record $r_i \in D$ is a tuple of m attribute values. The K-anonymity problem can be seen as modifying D by adding a new quasi-identifier value (the room where each patient will be screened) while ensuring that no record can be distinguished from at least $K-1$ other records in D', the modified dataset. Formally, the new dataset D' satisfies K-anonymity if, for each record $r_i = (a_1^i, a_2^i, \ldots, a_{m+1}^i) \in D'$, there exist at least $K-1$ other records $r_{i_1}, r_{i_2}, \ldots, r_{i_{K-1}} \in D'$ such that:

$$\forall s \in \{1, \ldots, m+1\},\ \forall j \in \{1, \ldots, K-1\} :\ a_s^i = a_s^{i_j}$$

where:

- r_{i_j} are the other records that need to have identical attribute values to r_i.
- a_s^i is the s-th attribute value of the i-th record.
- $a_s^{i_j}$ is the s-th attribute value of the i_j-th record.
- a_{m+1}^i is the attribute corresponding to the new column denoting the screenign room.

In other words, for each record in the modified data set, there must be at least $K-1$ other records that have the same values for the remaining attributes. This ensures that an adversary attempting to re-identify individuals in the data set based on the remaining attributes would have at least K possible matches for each record.

3.1.1. The Partition problem

There is an NP-complete problem that deserves a special mention, because it is strongly related with the subject of this project, *the Subset sum problem.* This decision problem is posed as

> *Given a multiset S of integers and a target value T, decide if there exists a subset of S whose sum is precisely T.*

This problem has several variants that are NP-complete too, for example the case in which all the inputs are natural numbers and the target sum is exactly half of the sum of all inputs. This special case is also known as the *Partition Problem.*

3.1.2. K-Anonymity Problem Definition

We have stated the point of view of the dataset observer. Since we want to solve the problem, let us define the problem's side, in particular what its instances are made of:

1. A multiset of groups of people that possess the same quasi-identifiers:

   $\mathcal{P} = \{g_1, g_2, \ldots, g_n\}$, where g_i is a number of people with a set of quasi-identifiers which makes them indistinguishable among themselves, but distinguishable from the individuals in any other group of people. Here, n would be the number of different groups of people. The total number of people in the multiset can be obtained by $\sum_{g_i \in \mathcal{P}} g_i$. We can assume that every person has the same amount of quasi-identifiers, and they have been grouped by their equivalence beforehand, which can be done trivially in time $O(n)$ over the number of patients.

2. An array of positive integers C that defines the rooms for a day of screening, where each integer is the capacity of the corresponding room:

   $C = \{c_1, c_2, ..., c_r\}$, where r is the number of rooms, the $i-th$ room has a capacity $c_i \in C$, and of course, $\sum_{i=1}^{r} c_i \geq \sum_{g_i \in P} g_i$.

And on the solution's side:

1. An array of arrays indicating indistinguishable groups of people screened for each room:

   $S = [S_1, S_2, ..., S_r]$. In this setting, S_j is an array of the form $[a_1^j, a_2^j, \ldots, a_n^j]$, where a_i^j is the number of people from group i assigned to room r_j, and the property $g_i = \sum_{j=1}^{r} a_i^j$ holds for all $1 \leq i \leq n$. It follows that $\sum_{i=1}^{n} a_i^j$ is the amount of people screened in $j-th$ room with capacity c_j, fulfilling $\sum_{i=1}^{n} a_i^j \leq c_i$.

2. An integer K representing the level of anonymity achieved in this instance. The anonymity of each room is the least repeated combination of quasi-identifiers values in the room, that is, the least cardinal of all the indistinguishable groups assigned to that room, and K is the anonymity of the room with minimum anonymity. Therefore,

$$K\text{−level} = \min\{a_i^j \mid a_i^j > 0 \text{ and } a_i^j \in S_j, S_j \in S, \forall i = 1 \ldots n,\ \forall j = 1 \ldots r\}$$

With these elements, we have all the ingredients to instantiate the K−Anonymity Problem to further study and solve it.

Versions of the problem:

The decision version of our anonymity problem consists in, given $K \in \mathbb{N}$, finding out whether K−level can be reached, and the optimization version consists in finding a solution maximizing the anonymity level.

3.2. BEING NP-COMPLETE

As per Definition 2.2.1 (classification of problems), to prove that the decision version of the K−Anonymity problem is NP-Complete, it must be proven that it belongs to class NP and is also NP-Hard.

3.2.1. Being NP-Hard via Partition

In order to prove K-anonymity is NP-Hard, the Partition problem will be reduced to the aforementioned $K-$Anonymity problem. Firstly, we must introduce the Partition problem, which is known to be NP-Complete already.

> Partition Problem: Being S a multiset of natural numbers, find a partition of S, i.e. S_1 and S_2 with $S_1,\ S_2 \subseteq S,\ S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$, such that $\Sigma(S_1) = \Sigma(S_2)$.

For instance, $S = \{1, 2, 3, 3, 3\}$ is a positive instance of partition because we have multisets $S_1 = \{1, 2, 3\}$ and $S_2 = \{3, 3\}$. In particular, the condition $S_1 \cap S_2 = \emptyset$ is met because each element in S is seen as unique: given $S = \{1a, 2b, 3c, 3d, 3e\}$, we have $S_1 = \{1a, 2b, 3c\}$ and $S_2 = \{3d, 3e\}$. Then $S_1 \cup S_2 = S$ and $S_1 \cap S_2 = \emptyset$.

Now, adding a condition to this problem, we will create a slightly different partition problem. This will be referred to as Equipartition problem. This condition will be that the number of elements —cardinal— of each subset is the same, that is, $|S_1| = |S_2|$. Adding this constraint to this problem is a step in between proving that $K-$Anonymity is NP-Hard, as we will do it by defining a reduction from Equipartition to $K-$Anonymity. Thus, first we need to prove that Equipartition is NP-Hard, and we will do it by reducing Partition to it.

3.2.2. Reducing Partition to Equipartition:

Let S be a Partition instance with $S = \{S_1, S_2, S_3, \ldots, S_W\}$, we will translate it to a similar equivalent Equipartition instance in terms of solvability. We will transform S into S_ϕ with the introduction of a number $\phi > 0$:

$$S_\phi = \{\underbrace{S_1 + \phi, S_2 + \phi, \ldots, S_W + \phi, \phi, \ldots, \phi}_{2 \cdot W}\}$$

The conditions that need to be met in order to have a positive instance of Partition are:

$$\sum S1 = \sum S2 \quad \textbf{where} \quad S1 \cup S2 = S \quad \textbf{and} \quad S1 \cap S2 = \emptyset$$

and for Equipartition are:

$$\sum S1_\phi = \sum S2_\phi \quad \textbf{and} \quad |S1_\phi| = |S2_\phi| \quad \textbf{where} \quad S1_\phi \cup S2_\phi = S_\phi \quad \textbf{and} \quad S1_\phi \cap S2_\phi = \emptyset$$

Positiveness from Partition to Equipartition:

Let's say that we have a positive instance of our original Partition problem. This will be of the form:

$$\sum S1 = S_{11} + S_{12} + \ldots + S_{1n} = \sum S2 = S_{21} + S_{22} + \ldots + S_{2m} = \frac{\sum S}{2}$$

where: $S1,\ S2 \subset S$ and

$$S1 \cup S2 = \{S_{11}, S_{12}, \ldots, S_{1n}, S_{21}, S_{22}, S_{23}, \ldots, S_{2m}\} = S = \{S_1, S_2, S_3, \ldots, S_W\}$$

Possibly $m \neq n$. To maintain the cardinality of subsets equal, the distribution of the new introduced ϕ numbers in our transformed set S_ϕ must subside the difference. The mapping is as follows:

$$\sum S1_\phi = \underbrace{\overbrace{(S_{11}+\phi)+(S_{12}+\phi)+\ldots+(S_{1n}+\phi)}^{n}+\overbrace{\phi+\ldots+\phi}^{m}}_{W}$$

$$= \sum S2_\phi = \underbrace{\overbrace{(S_{21}+\phi)+(S_{22}+\phi)+\ldots+(S_{2m}+\phi)}^{m}+\overbrace{\phi+\ldots+\phi}^{n}}_{W}$$

$$= \frac{\sum S_\phi}{2}$$

besides, $|S1_\phi| = |S2_\phi| = n + m$.

We have now proved that if the Partition instance over a set S is positive, so it will be for a Equipartition instance on the transformed set S_ϕ.

Positiveness from Equipartition to Partition:

Trivially a positive instance of Equipartition over a set S_ϕ is a positive instance of Partition over a set S_ϕ.

Summarizing, we have proved that if an instance of Partition is positive, then its transformed Equipartition instance is positive; and if a transformed Equipartition instance is positive then the Partition instance it comes from is positive. Hence, **Equipartition is at least as hard as Partition** and, in particular, this makes it NP-Hard.

3.2.3. Reducing Equipartition to $K-$Anonymity:

Now, let's reduce the Equipartition problem to the $K-$Anonymity problem.

Let the multiset of positive integers $\mathcal{P}$ denote an instance of Equipartition and let $\ell = \min(\mathcal{P})$. Here, ℓ is of great importance. Let us write the multiset $\mathcal{P}$ in terms of ℓ, denoted $\mathcal{P}_\ell$, for which we will use the auxiliary multiset O constructed by taking all elements of $\mathcal{P}$, with the same multiplicity, and subtracting ℓ to all of them and removing all resulting 0 elements. Let $O = \{O_1, \ldots, O_r\}$. In terms of multiset O and number ℓ, multiset $\mathcal{P}_\ell$ looks like this:

$$\mathcal{P} = \mathcal{P}_\ell = \{\underbrace{O_1+\ell, O_2+\ell, \ldots, O_r+\ell, \ell, \ldots, \ell}_{|\mathcal{P}|}\}$$

Our mapping to $K-$Anonymity will imply that ℓ will be swapped by a number $\alpha = \frac{\sum \mathcal{P}}{R}$, where $R = 2$ is the particular number of rooms of our $K-$anonymity problem instance. From this notion, we create the multiset $\mathcal{K}$, where ℓ is substituted by $\alpha = \frac{\sum \mathcal{P}}{2} = \min(\mathcal{K})$. For clarity we will write $\mathcal{K}$ in terms of O and α as we did with $\mathcal{P}_\ell$:

$$\mathcal{K} = \mathcal{K}_\alpha = \{\underbrace{O_1+\alpha, O_2+\alpha, \ldots, O_r+\alpha, \alpha, \ldots, \alpha}_{|\mathcal{P}|}\}$$

In our $K-$Anonymity instance, the cardinalities of the groups of patients which are indistinguishable from each other but can be distinguished from all patients from other groups are, precisely, the numbers denoted by multiset $\mathcal{K}$. Besides, our $K-$Anonymity instance has two rooms, to mimic the division between the two sets of Equipartition, X and Y, each one with a capacity for patients of exactly:

$$C_X = C_Y = \frac{\sum \mathcal{K}_\alpha}{2} = \frac{\sum O + |\mathcal{P}| \cdot \alpha}{2}$$

Finally, in our anonymity instance, we inquire whether it is possible to achieve an anonymity level of at least α, meaning that within each room, any person should be indistinguishable from at least α other individuals also in the same room, where α is $\frac{\sum \mathcal{P}}{2}$.

Positiveness from Equipartition to $K-$Anonymity:

On the presented mapping, let us suppose that $\mathcal{P}1 \cup \mathcal{P}2 = \mathcal{P}$ is a positive instance of Equipartition, that is:

$$\mathcal{P}1 \cup \mathcal{P}2 = \mathcal{P} \text{ and } |\mathcal{P}1| = |\mathcal{P}2| \text{ with } \sum \mathcal{P}1 = \sum \mathcal{P}2 = \frac{\sum \mathcal{P}}{2}$$

Let us introduce a mapping to our auxiliary sets $\mathcal{O}1 \cup \mathcal{O}2 = \mathcal{O}$, where $\mathcal{O}1$ is constructed from $\mathcal{P}1$ and $\mathcal{O}2$ is constructed from $\mathcal{P}2$ exactly as we constructed $\mathcal{O}$ from $\mathcal{P}$ before (i.e. by taking the same elements while respecting their multiplicity, subtracting $\ell = \min(\mathcal{P})$ to all of them, and removing all resulting 0 elements). Let the resulting multisets $\mathcal{O}1$ and $\mathcal{O}2$ be:

$$\mathcal{O}1 = \{\mathcal{O}1_1, \ldots, \mathcal{O}1_n\}$$

$$\mathcal{O}2 = \{\mathcal{O}2_1, \ldots, \mathcal{O}2_m\}$$

Now, we can write $\mathcal{P}_\ell$ and $\mathcal{K}_\alpha$ in terms of $\mathcal{O}1$, $\mathcal{O}2$ and ℓ, α, respectively:

$$\mathcal{P}_\ell = \underbrace{\left\{ \begin{array}{l} \mathcal{O}1_1 + \ell, \mathcal{O}1_2 + \ell, \ldots, \mathcal{O}1_n + \ell, \\ \mathcal{O}2_1 + \ell, \mathcal{O}2_2 + \ell, \ldots, \mathcal{O}2_m + \ell, \\ \underbrace{\ell, \ell, \ldots, \ell}_{|\mathcal{P}|-(m+n)} \end{array} \right\}}_{|\mathcal{P}|} \qquad \mathcal{K}_\alpha = \underbrace{\left\{ \begin{array}{l} \mathcal{O}1_1 + \alpha, \mathcal{O}1_2 + \alpha, \ldots, \mathcal{O}1_n + \alpha, \\ \mathcal{O}2_1 + \alpha, \mathcal{O}2_2 + \alpha, \ldots, \mathcal{O}2_m + \alpha, \\ \underbrace{\alpha, \alpha, \ldots, \alpha}_{|\mathcal{P}|-(m+n)} \end{array} \right\}}_{|\mathcal{P}|}$$

Besides, we can write $\mathcal{P}1$ and $\mathcal{P}2$ as follows:

$$\mathcal{P}1 = \underbrace{\left\{ \begin{array}{l} \mathcal{O}1_1 + \ell, \mathcal{O}1_2 + \ell, \ldots, \mathcal{O}1_n + \ell, \\ \underbrace{\ell, \ell, \ldots, \ell}_{\frac{|\mathcal{P}|}{2}-n} \end{array} \right\}}_{|\mathcal{P}1|} \qquad \mathcal{P}2 = \underbrace{\left\{ \begin{array}{l} \mathcal{O}2_1 + \ell, \mathcal{O}2_2 + \ell, \ldots, \mathcal{O}2_m + \ell, \\ \underbrace{\ell, \ell, \ldots, \ell}_{\frac{|\mathcal{P}|}{2}-m} \end{array} \right\}}_{|\mathcal{P}2|}$$

We can observe that any number in $\mathcal{O}$ is upper bounded by α. Indeed:

$$\mathcal{O}1_i \leq \sum \mathcal{O}1 = \alpha - \ell\,|\mathcal{P}1| < \alpha$$
$$\mathcal{O}2_i \leq \sum \mathcal{O}2 = \alpha - \ell\,|\mathcal{P}2| < \alpha$$

Notably, if we take a group of g_i individuals such that one individual is placed in one room, and the remaining individuals are distributed in any manner, then the $K-$Anonymity level of the solution will drop to 1, since our $K-$level is always the less anonymous group. This represents the lowest possible level of $K-$Anonymity of any solution.

Of course, the best split for a group with g_i individuals, in general, is $\frac{g_i}{1}$, where we place the whole group in 1 room with capacity $c_i \geq g_i$, since if g_i is the smallest K-Anonymous group, then the K-level of the solution becomes g_i, which is the maximum achievable with a group $g_i \in G$. Let us consider a group of g_i indistinguishable people which cannot fit in any given room. Intuitively, the best split between R rooms that can hold some capacity c is always $\frac{g_i}{R}$, provided that $\frac{g_i}{R} < c$.

On the other hand, if we want to achieve a $K-$level $\geq \alpha$, $\alpha = \min(\mathcal{K})$, we observe that any group that contains fewer than $2 \cdot \alpha$ individuals cannot be split between the two available rooms while maintaining the anonymity level K above α. They must go undivided to one of the two rooms. A group that cannot be split will be called $\alpha-$group.

Since we have observed that every element in the set O is smaller than α, and the elements of $\mathcal{K}$ are either of the form $O_i + \alpha$ or equal to α, it follows that each element of $\mathcal{K}$ is also smaller than $2 \cdot \alpha$. Consequently, we know that all groups of individuals in $\mathcal{K}$ are α-groups.

With these $\alpha-$groups, our multiset $\mathcal{K}$ and our rooms C_X, C_Y, we now have the following properties:

(a) Each α-group must go entirely to one room or the other to ensure a $K-$level $\geq \alpha$ in our solution.

(b) All groups in K are α-groups.

(c) Each room can accommodate $\frac{\sum O + |\mathcal{P}| \cdot \alpha}{2}$ individuals.

(d) By construction, in no room can we place people from more than $\frac{|\mathcal{P}|}{2}$ different groups if groups go undivided, as they would not fit.

Since we know that the α-groups can never be split between the two rooms, any two multisets $\mathcal{K}1$ and $\mathcal{K}2$ constituting a solution of K-Anonymity will be such that all elements of a multiset (say $\mathcal{K}1$) are α-groups screened in room 1, and all elements of the other one (say $\mathcal{K}2$) are α-groups screened in room 2. It corresponds to the solution definition in 3.1.2 with $S = [\mathcal{K}1, \mathcal{K}2]$, and K-level $= \min\{a \mid a > 0 \text{ and } a \in \mathcal{K}1 \cup \mathcal{K}2\}$. Indeed, let us define $\mathcal{K}1$ and $\mathcal{K}2$ as we defined $\mathcal{P}1$ and $\mathcal{P}2$ before, replacing all appearances of ℓ by α:

$$\mathcal{K}1 = \underbrace{\left\{ O1_1 + \alpha, O1_2 + \alpha, \ldots, O1_n + \alpha, \underbrace{\alpha, \alpha, \ldots, \alpha}_{\frac{|\mathcal{K}|}{2} - n} \right\}}_{|\mathcal{K}1|}$$

$$\mathcal{K}2 = \underbrace{\left\{ O2_1 + \alpha, O2_2 + \alpha, \ldots, O2_m + \alpha, \underbrace{\alpha, \alpha, \ldots, \alpha}_{\frac{|\mathcal{K}|}{2} - m} \right\}}_{|\mathcal{K}2|}$$

By construction, the capacity of each room is $C_X = C_Y = \frac{\sum O + |\mathcal{P}| \cdot \alpha}{2}$. Since we know that $\frac{\sum O}{2} < \frac{\sum \mathcal{P}}{2} = \alpha$, it follows that $\frac{\sum O + |\mathcal{P}| \cdot \alpha}{2} = C_X < \frac{\sum \mathcal{P} + |\mathcal{P}| \cdot \alpha}{2} = \frac{2\alpha + |\mathcal{P}| \cdot \alpha}{2} = \frac{\alpha(|\mathcal{P}|+2)}{2}$. We will introduce the difference between these numbers as:

$$\delta = \frac{\alpha(|\mathcal{P}| + 2)}{2} - C_x = \frac{\alpha(|\mathcal{P}| + 2)}{2} - \frac{\sum O + |\mathcal{P}| \cdot \alpha}{2} = \alpha - \frac{\sum O}{2}$$

If we place $\frac{|\mathcal{P}|}{2}$ complete groups in one room, then we already have a minimum of $\frac{|\mathcal{P}|}{2} \cdot \alpha$ individuals in that room. The remaining amount of $\delta < \alpha$ individuals to reach C_X must come from the excesses over α on all the numbers of individuals in those groups, which are, in fact, the original numbers from our original auxiliary set, meaning the numbers in $O1$, or the numbers in $O2$, added to the numbers in the multisets $\mathcal{K}1$ and $\mathcal{K}2$, respectively.

As per (d), if one room can contain a maximum of $\frac{|\mathcal{K}|}{2} = \frac{|\mathcal{P}|}{2}$ different α-groups, then the other room must contain a minimum of $\frac{|\mathcal{P}|}{2}$ different α-groups. Thus, we conclude on the fact that both rooms must contain *exactly* $\frac{|\mathcal{P}|}{2}$ non-split α-groups, meaning $|\mathcal{K}1| = |\mathcal{K}2|$.

With this information, going back to multiset $\mathcal{P}_\ell$, in our multisets $\mathcal{K}1$ and $\mathcal{K}2$ we have the following:

$$\sum \mathcal{K}1 = \sum \mathcal{K}2 = \sum \mathcal{P}1 + (\alpha - \ell) \cdot |\mathcal{K}1| = \sum \mathcal{P}2 + (\alpha - \ell) \cdot |\mathcal{K}2| = \frac{\sum \mathcal{P} + (\alpha - \ell) \cdot |\mathcal{K}|}{2} = \frac{\sum O + \alpha \cdot |\mathcal{K}|}{2}$$

Therefore, this $K-$Anonymity instance requires placing in each room $\frac{|\mathcal{P}|}{2}$ original numbers from the Equipartition instance over multiset $\mathcal{P}$ that sum to exactly the remaining amount δ.

We conclude that, if an instance of Equipartition is positive, then the $K-$Anonymity instance derived from from it is positive as well.

From K-Anonymity to Equipartition:

Let's say that our $K-$Anonymity instance $\mathcal{K}$ produced by our reduction is positive. Let us recall its definition:

$$\mathcal{K} = \mathcal{K}_\alpha = \underbrace{\left\{ \begin{array}{l} O_1 + \alpha, O_2 + \alpha, \ldots, O_r + \alpha, \\ \underbrace{\alpha, \alpha, \ldots, \alpha}_{|\mathcal{K}|-(r)} \end{array} \right\}}_{|\mathcal{K}|}$$

Where O and α are defined as before.

We know that this reduction comes from the multiset $\mathcal{P}_\ell$ (the instance of Equipartition), where $\ell < \alpha$. Remembering that no number in O can be bigger than α in our created instance, by construction no number in $\mathcal{K}$ is bigger than $2 \cdot \alpha$, so all groups in $\mathcal{K}$ are $\alpha-$groups. Also, the rooms can only fit, at most, $\frac{|\mathcal{K}|}{2}$ full $\alpha-$groups each due to their capacity.

Because of this, the multiplicity of all original numbers in O and all original α number is kept in any solution consisting of two multisets $\mathcal{K}1$ and $\mathcal{K}2$, where these multisets denote the assignation of (full) groups to rooms 1 and 2, respectively. Hence, $|\mathcal{K}1| = |\mathcal{K}2|$. Since no groups are split, we have $\mathcal{K}1 \cup \mathcal{K}2 = \mathcal{K}$. We conclude that $\mathcal{K}1$ and $\mathcal{K}2$ can be defined like this in terms of α and two multisets $O1 = \{O1_1, ..., O1_n\}$ and $O2 = \{O2_1, ..., O2_m\}$ with $O = O1 \cup O2$ and $O1 \cap O2 = \emptyset$:

$$\mathcal{K}1 = \underbrace{\left\{ \begin{array}{l} O1_1 + \alpha, O1_2 + \alpha, \ldots, O1_n + \alpha, \underbrace{\alpha, \alpha, \ldots, \alpha}_{\frac{|\mathcal{K}|}{2}-n} \end{array} \right\}}_{|\mathcal{K}1|}$$

$$\mathcal{K}2 = \underbrace{\left\{ \begin{array}{l} O2_1 + \alpha, O2_2 + \alpha, \ldots, O2_m + \alpha, \underbrace{\alpha, \alpha, \ldots, \alpha}_{\frac{|\mathcal{K}|}{2}-m} \end{array} \right\}}_{|\mathcal{K}2|}$$

Where $O2 \cup O1 = O$, denoting the elements of O that must be in $\mathcal{K}1$ and $\mathcal{K}2$ respectively.

Let us remember that the capacities for each room in our positive $K-$Anonymity instance are of the same size, and both must be full to the brim since both capacities are of $\frac{\sum \mathcal{K}}{2}$ exactly, meaning $\sum \mathcal{K}1 = \sum \mathcal{K}2 = \frac{\sum \mathcal{K}}{2}$.

Regressing our mapping over $\mathcal{K}1$ and $\mathcal{K}2$, we can arrive at some multisets $\mathcal{P}1$ and $\mathcal{P}2$ by replacing all appearances of α in the definitions of $\mathcal{K}1$ and $\mathcal{K}2$ with $\ell = \min(\mathcal{P})$, respectively. Since all groups of $\mathcal{K}1$ and $\mathcal{K}2$ are α-groups, we have that:

$$\sum \mathcal{P}1 = \sum \mathcal{P}2 = \sum \mathcal{K}1 - |\mathcal{K}1| \cdot (\alpha - \ell) \textbf{ and } |\mathcal{P}1| = |\mathcal{P}2| \textbf{ and } \mathcal{P}1 \cup \mathcal{P}2 = \mathcal{P} \textbf{ and } \mathcal{P}1 \cap \mathcal{P}2 = \emptyset$$

Thus, $\mathcal{P}1$ and $\mathcal{P}2$ constitute a solution to our original Equipartition instance, that is:

$$\mathcal{P} = \mathcal{P}_\ell = \underbrace{\left\{ \begin{array}{c} O_1 + \ell, O_2 + \ell, \ldots, O_n + \ell, \underbrace{\ell, \ell, \ldots, \ell}_{|\mathcal{K}|-(n)} \end{array} \right\}}_{|\mathcal{K}|}$$

We conclude that if the derived $K-$Anonymity instance is positive, then the original Equipartition instance it comes from is positive as well. We conclude that $K-$Anonymity is NP-Hard.

Since checking the validity of a possible assignment of patients to rooms involves very simple operations, such as checking that the room capacities are respected and that the original group sizes are kept, we can say that $K-$Anonymity is in NP class by the remark in the Definition of the NP class, so by the Definition of the NP-Complete class $K-$Anonymity must be NP-Complete.

CHAPTER 4

Solving via Exhaustive Algorithm

4.1. BRANCH AND BOUND ALGORITHM

The Branch and Bound (BnB) algorithm is a general method for solving combinatorial optimization problems. It systematically explores branches of a solution space tree by dividing the problem into smaller subproblems and using bounds to eliminate branches that cannot yield an optimal solution.

We will consider that all possible solutions to some problem P can be represented as a tree where:

- The **root** of the tree represents the initial state of the problem solution, where no decisions have been made and all possible solutions remain to be explored.
- The **leaves** of the tree represent complete solutions to the problem, where all decisions have been finalized.
- The **intermediate nodes** of the tree correspond to partial solutions where some decisions have been made, and not all possible leaves can been reached. These nodes represent branches in the decision-making process.

Vulgarly, the BnB algorithm is an algorithm that generates this tree and ignores the ugly branches. It consists of the following steps:

1. **Branching**: Divide the problem into smaller subproblems. Each result of a smaller subproblem defines a node in the exploration space.
2. **Bounding**: Calculate an upper bound of the value of the solution under construction, i.e. each node.
3. **Pruning**: When a solution is found, the value of the solution becomes the best known value. When considering nodes after some solution is found, we will eliminate nodes that cannot improve upon the best-known solution, thus pruning the tree.
4. **Searching**: Explore the branches of the solution space until the best leaves are found.
5. **Termination**: End when all branches have been explored or pruned, or when a certain condition is met.

Algorithm 4.1 shows the structure of the branch and bound algorithm with backtracking through recursion and Depth-First search in pseudocode:

Algorithm 4.1: Branch and Bound with Recursion and Backtracking in Depth-First order

```
Input: A problem instance P, a current node current_node, the current best solution
       best_sol, a maximum reachable quality of the solution max_bound
Output: The best solution found
Function BranchAndBoundRecursive(P, current_node, & best_sol, max_bound):
    if Objective(best_sol) ≥ max_bound then
        return best_sol;
    end
    if IsFeasible(current_node) then
        bound ← ComputeBound(current_node);
        if bound < Objective(best_sol) and bound < max_bound then
            if IsSolution(current_node) then
                best_sol ← current_node;
            end
            else
                foreach promisingChild in Branch(current_node) do
                    best_sol ← BranchAndBoundRecursive(P, promisingChild,
                     best_sol, max_bound);
                end
            end
        end
    end
    return best_sol
```

We will now define these steps in our $K-$Anonymity problem.

4.2. SOLVING THE K-ANONYMITY PROBLEM VIA BRANCH AND BOUND

4.2.1. Branching

In the K-Anonymity problem, branching is accomplished by dividing the problem of distributing patient groups into different rooms into smaller subproblems.

We branch by selecting a patient group and finding all possible promising combinations of assignations into the remaining room capacities. Each branch represents a possible assignment of a patient group to the set of remaining room capacities.

This is the subproblem, essentially an Integer Partition problem with extra restrictions, namely, the result will be an array in which numbers can be repeated, but are bounded by the array of remaining room capacities.

It is important that only solutions that can have a higher $K-$Level than the current one are explored, and that we only calculate the first one that does this, to explore in-depth the tree of possible solutions.

The algorithm to find out the remaining room capacities is trivial: when a group distribution for the set of rooms is decided into our solution, i.e. we move from one node to another, the capacity of each room is decreased by the amount of patients assigned to it.

Solving the Integer Partition in this context implies the operation described next.

Integer Partition For Patient Distributions

`getNextUsefulPatientDistribution` generates one valid and promising distribution of patients across multiple rooms, adhering to each room's capacity constraints and avoiding previously explored distributions. It uses a recursive helper function `findNextUsefulDistribution` to explore and find the next useful distribution.

Inputs

- `currentSolution`: A list of lists representing the current partial solution of patient distributions across the rooms.
- `exploredDistributions`: A set of lists of lists, where each list represents a distribution of patients that has already been explored.
- `kLevelToBeat`: An integer value representing the minimum k-level that the next distribution must exceed to be considered useful.
- `totalPatients`: The total number of patients that need to be distributed.
- `roomCapacities`: A list where each item represents the maximum number of patients that can be accommodated in each room.

Outputs

- A pair consisting of:
  - `nextDistribution`: A list representing the next useful distribution of patients across the rooms.
  - `foundNewDistribution`: A boolean indicating whether a valid and new distribution was found.

Process of the auxiliary caller The auxiliary caller (i.e. Algorithm 4.3) carries out the following process:

1. First, the function checks if the current solution is promising by comparing its k-level with the `kLevelToBeat`. If it is not promising, it returns an empty distribution and `false`.
2. Initialize an empty list `nextDistribution` to store the next valid distribution.
3. Invoke the recursive function `findNextUsefulDistribution` (i.e. Algorithm 4.2) with the following parameters: the remaining number of patients, the current solution, the room capacities, the set of explored distributions, the $K-$level to beat, and the empty `nextDistribution`.
4. Inside `findNextUsefulDistribution`:
   a) If the remaining patients is zero and the size of `nextDistribution` does not exceed the number of rooms, check if the distribution is promising (i.e., its $K-$level exceeds `kLevelToBeat`) and hasn't been explored before. If so, it is a **valid and promising** distribution of the next group, return it immediately and stop the search.
   b) If we have used all rooms and have not found a valid and unexplored distribution, backtrack.
   c) Otherwise, for the next room, try placing a number of patients from the minimum of: the remaining patients to assign of this group and the room's capacity, down to zero. We discard distributions that are not promising (i.e., where the number of patients placed is between zero and `kLevelToBeat`, including `kLevelToBeat`). This enforces pruning.
   d) If we successfully assign some patients to the room, we recursively call the function, with the remaining patients and the partial distribution, to then fill out the next room with its corresponding remaining capacity. If we cannot, the algorithm will return false. Returning false will create a backtracking step by removing the last split of patients to the previous room. The algorithm can resume the exploration after undoing this last assignation.
5. If a valid distribution is found, return it along with `true`. Otherwise, return an empty distribution and `false`, which will indicate to our BnB algorithm that all possible branches from that node are pruned. Here BnB will be forced to backtrack and undo the assignments of the previous group.

Algorithm 4.2: Patient Distribution IP Algorithm

Input: remainingPatients: Number of patients of the group not yet assigned.
currentSolution: Current partial solution of patient distributions across rooms.
roomCapacities: List of room capacities available.
exploredDistributions: Set of previously explored distributions.
kLevelToBeat: The minimum k-level that the next distribution must exceed.
nextDistribution: The current distribution being evaluated.
Output: A boolean indicating whether a new, valid, and useful distribution has been found.
Function `findNextUsefulDistribution(`*remainingPatients, currentSolution, roomCapacities, exploredDistributions, kLevelToBeat, nextDistribution*`)`**:**

```
if remainingPatients ≠ 0 and remainingPatients ≤ kLevelToBeat then
    return false; // Optimization: Useless to explore
end
if remainingPatients == 0 and
 nextDistribution.size() ≤ roomCapacities.size() and
 findKLevelOfNextBranch(nextDistribution) > kLevelToBeat then
    currentDistWrapper ← currentSolution::nextDistribution;
    if not exploredDistributions.contains(currentDistWrapper) then
        return true; // Return true to indicate a valid distribution has been found
    end
    return false; // Already explored, must backtrack
end
if nextDistribution.size() ≥ roomCapacities.size() then
    return false; // Return if the number of rooms used exceeds the available rooms
end
for i = min(remainingPatients, roomCapacities[nextDistribution.size()])
 down to 0 do
    if i == 0 or i > kLevelToBeat then
        nextDistribution.push_back(i);
        if findNextUsefulDistribution(remainingPatients − i,
         currentSolution, roomCapacities, exploredDistributions,
         kLevelToBeat, nextDistribution) then
            return true; // Return immediately if a valid distribution is found
        end
        nextDistribution.pop_back();
        // Backtrack: remove the last element and continue
    end
end
return false; // No valid distribution was found in this path
```

Algorithm 4.3: Auxiliary Caller Function

Input: totalPatients: Total number of patients to assign.
currentSolution: Current partial solution of patient distributions across rooms.
roomCapacities: List of remaining room capacities.
exploredDistributions: Set of previously explored distributions.
kLevelToBeat: The minimum k-level that the next distribution must exceed.
Output: nextDistribution: The next useful and valid distribution of patients across rooms.
foundNewDistribution: Boolean indicating if a new valid distribution was found.
Function `getNextUsefulPatientDistribution(`*currentSolution, exploredDistributions, kLevelToBeat, totalPatients, roomCapacities*`):`

```
if isNotRoot(currentSolution) and findKLevelOfBranch(currentSolution)
 ≤ kLevelToBeat then
    return {{}, false}; // Prune branch if useless to explore
end
Initialize an empty list nextDistribution;
// Start the recursive distribution generation
 foundNewDistribution ← findNextUsefulDistribution(totalPatients,
 currentSolution, roomCapacities, exploredDistributions, kLevelToBeat,
 nextDistribution);
return {nextDistribution, foundNewDistribution};
```

4.2.2. Bounding

Bounding is used to estimate the potential of each subproblem. In our case, this would trivially be the maximum achievable $K-$Level given by the groups that are already assigned. We retrieve the minimum non-zero value in the current distribution, which serves as an upper bound on the quality of the solution. The bound of a branch created from a node will be given by both the minimum, non-zero number of the next group distribution in the rooms and the minimum non-zero in the already assigned ones. This bound will be used in pruning decisions.

4.2.3. Pruning

Pruning involves eliminating subproblems that cannot lead to a better solution than the current best-known solution. Using our aforementioned bounding method, if the minimum non-zero value of our current group assignations is less than or equal to the best-known solution's minimum, it indicates that this branch cannot become better than it, and should be pruned.

This will reduce vastly the exploration space since, for example, after achieving some solution of a $K-$Level of 2, all solutions that are being generated with groups of K-Level ≤ 2 will be pruned.

4.2.4. Searching

The searching process in BnB is carried out by recursively exploring all branches generated through patient group assignments. The search continues until all possible distributions have been considered or pruned.

In the context of the K-Anonymity problem, the decision to use Depth-First Search (DFS) order of exploration is particularly effective for improving the pruning process within the BnB framework. The key reasons for this choice are the following ones:

1. **Efficient Bounding:** DFS allows the algorithm to explore a complete path from the root to a leaf (a full solution) as quickly as possible. In BnB once a full solution is found, it immediately provides a concrete bound. Subsequent branches can be pruned as soon as their partial solutions fail to exceed this bound. Since DFS explores one branch entirely before moving to another,

it continuously updates and tightens the bound with each complete solution found, thus improving the effectiveness of pruning.

2. **Memory Efficiency:** DFS inherently uses less memory compared to breadth-first search (BFS), as it only needs to store the current path from the root to the current node (i.e., the current partial solution). This is particularly beneficial in combinatorial problems like $k-$Anonymity, where the solution space can be large. The reduced memory footprint ensures that more resources are available, which is crucial when solving NP-Hard problems.
3. **Targeted Exploration:** The nature of DFS ensures that the search dives into specific combinations of patient group distributions before considering alternatives. This exploration aligns with the goal of finding and verifying potential optimal solutions quickly, focusing computational efforts on promising regions of the search space. We also help achieve this by traversing the next possible assignations in reverse order, as it can be seen in the decreasing order of the `for` loop in Algorithm 4.2.

   For example, let us consider some *Rooms* with some remaining capacity and some *PatientGroups* that need to be assigned in those rooms. In Figure 4.1 we show the assignations that Algorithm 4.2 returns.

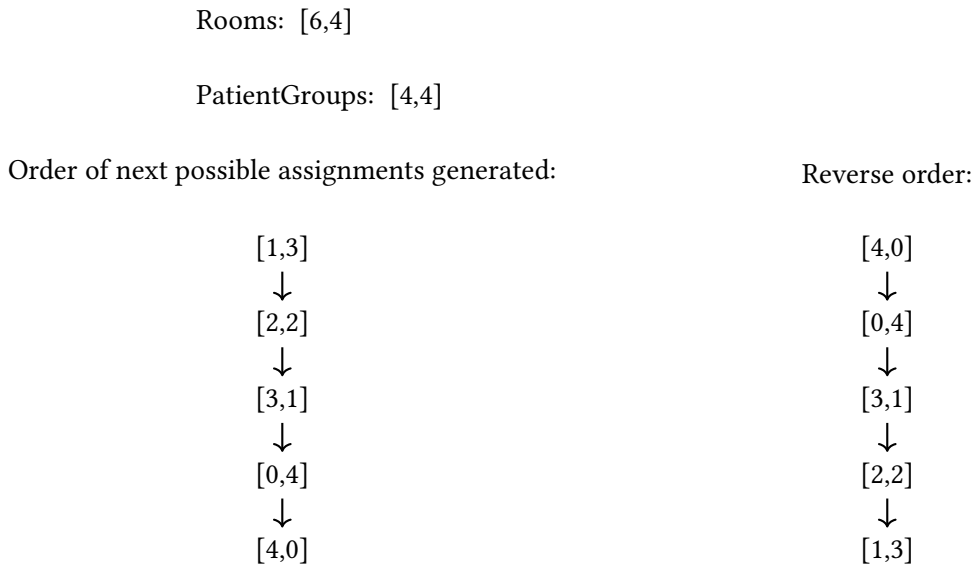


Figure 4.1: Regular and Reverse Order of Arrays of Size 2 Summing to 4

As we can see, both $[4, 0]$ and $[0, 4]$ offer the maximum $K-$Level of 4 over that branch decision. They will be better to explore in-depth than the others. Let's say that by exploring the $[0, 4]$ branch we can reach a leaf that has a $K-$Level of 2, after that we will not explore anything else in the current branch, because we already have one of the best leaves that could be possibly generated.

This idea, in conjunction with DFS, can bypass large portions of the solution space that are unlikely to yield better solutions, directly contributing to more effective pruning.

4.2.5. Termination

A maximum, not always achievable, $K-$Level is trivially given by the minimum group of patients in the groups to assign, or in some cases, the size of the smallest room we will need to use.

Upper bound analysis

Let $P = \{p_1, p_2, \ldots, p_m\}$ represent the set of patient groups, where p_i denotes the size of the i-th group, and let $R = \{r_1, r_2, \ldots, r_n\}$ represent the set of room capacities, where r_j denotes the capacity

of the j-th room. We seek an assignment matrix $A = [a_{ij}]$, where:

- The sum of each row i in A is equal to p_i:

$$\sum_{j=1}^{n} a_{ij} = p_i \quad \text{for all } i \in \{1, 2, \ldots, m\}.$$

- The sum of each column j in A is less than or equal to r_j:

$$\sum_{i=1}^{m} a_{ij} \leq r_j \quad \text{for all } j \in \{1, 2, \ldots, n\}.$$

Our objective is to maximize the minimum element of A, i.e., to find:

$$\max_{A} \textit{min-non-zero}_{i,j} a_{ij}.$$

An upper bound Υ for $\min\limits_{i,j} \ a_{ij}$ can be established as:

$$\Upsilon = \max\{\textit{min-non-zero}\{s_1, s_2, \ldots, s_p\}, \ \min\{u_1, u_2, \ldots, u_r\}\}.$$

Where $u_i \in U \subseteq R$ is some possible subset of the rooms we need to use in order to assign every patient, and $s_i \in S$ is some possible split of each group $p_i \in P$ inside A. This bound reflects the fact that no assignment can exceed the smallest group size or smallest usable room capacity. Calculating it presents problems from the computational point of view, since establishing this upper bound is similar to solving the original problem, because we must care about the rooms that will be used and the possible splits for the patient groups. Therefore, computing the upper bound Υ involves verifying if the assignment matrix A can satisfy both the row and column constraints with $\textit{min-non-zero}_{i,j} \ a_{ij} = \Upsilon$. Making Υ the best $K-$Level achievable in our instance.

Because of this, ignoring the concept of the usability of the rooms, we will create a different upper bound as follows:

- The upper-bound calculation algorithm terminates if we can assign the smallest patient group, to a room.
- If the lesser group cannot fit undivided into any room, this means no groups fit into any room undivided. The lesser group is split into two different groups resulting in a new group list of two members.
- We repeat the process until the smallest patient group fits in some room.

Algorithm 4.4: Find Smallest Group Split That Fits

```
Input: patientGroups: List of patient group sizes, roomCapacities: List of room capacities
Output: Relaxed upper bound for K−Level
while true do
    Sort patientGroups in ascending order;
    a ← Lowest element of patientGroups;
    for each c ∈ roomCapacities do
        if a ≤ c then
            return a;
        end
    end
    Initialize an empty list splitGroups;
    h1 ← ⌊a/2⌋;
    h2 ← a − h1;
    Append h1 and h2 to splitGroups;
    Replace patientGroups with splitGroups;
end
```

While this upper bound may be relaxed and often not achievable in practice, it provides a fast and computationally efficient method to determine a preliminary bound. When correct, it means that if a solution of this quality is found, the rest of the tree can be pruned. In the best case, only one branch, from root to leaf, will be traversed to solve the problem.

For this, the termination condition of our BnB algorithm is met if we achieve a solution of a $K-$Level of the amount retrieved by the aforementioned algorithm. Alternatively, the algorithm will stop when all branches of the solution space have been explored or pruned, thus returning the best possible solution achievable in that context.

4.3. RESULTS OF THE BRANCH AND BOUND ALGORITHM

The BnB algorithm has extremely fast results in cases where the upper bound is tight and the termination condition can be met. This is because once the $K-$Level reaches the termination condition, the rest of the tree does not need to be explored. In other words, if the upper bound is relaxed, the computer will need to try, and backtrack from, a factorial number of combinations once the maximal reachable $K-$Level is reached. It will do this only to ultimately give back a combination that had been known before attempting to reach a better one.

Because of this, we will divide our results into the two following scenarios:

- **Cases where the upper bound is reachable**: In these cases, Algorithm 4.4 finds the reachable maximum $K-$Level. This means that the smallest group can fit undivided in some room, and all the assignments of other groups, split or not, result in numbers higher or equal than the size of that smallest group.
- **Cases where the upper bound is unreachable**: In these cases, Algorithm 4.4 gives a relaxed $K-$Level, impossible to reach. This means that there is some group that must be split in a way that some part of the split is smaller than the size of that smallest group.

In Annex A we can find the configuration used to run the algorithm and generate the plots.

4.3.1. Cases where the upper bound is reachable

In this section, we demonstrate cases where the Branch and Bound (BnB) algorithm reaches the upper bound and prunes the remaining branches of the search tree. These scenarios result in fast computations, as the upper bound termination condition of the Depth-First Search (DFS) effectively limits the computational cost, regardless of the growth in input size.

1. **Case 1:**
   Consider 5 groups and 5 rooms, each with size X. The room configuration is as follows:

   $$\text{Rooms} = \text{Groups} = [X, X, X, X, X]$$

   In this scenario, X grows from 1 to 100,000 in increments of 1. In Figure 4.2 we can see how, because of DFS and the termination condition, the number of steps taken by the algorithm is always the same, regardless of the growth of X. Of course since X people can go to a room with a size of X, the highest K-Level of the solution is exactly X, indeed our X-Axis will be the $K-$Level of the solution.

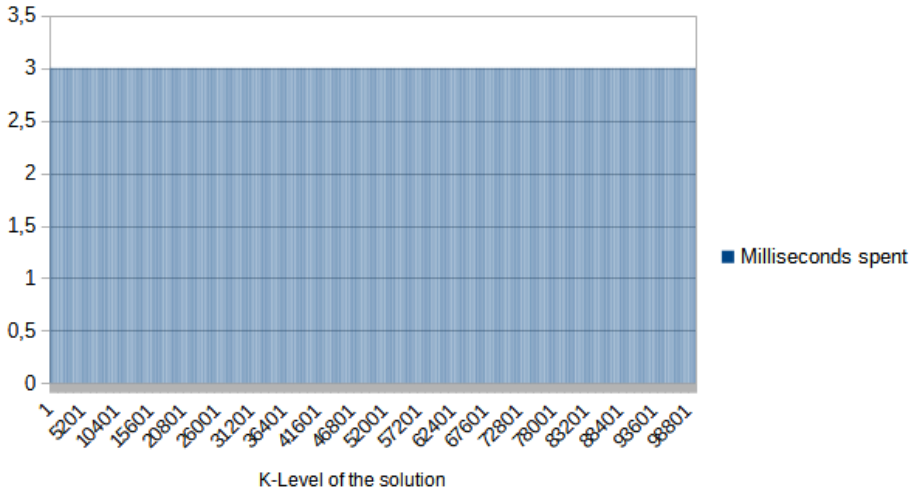


Figure 4.2: BnB algorithm performance for 5 groups and 5 rooms, each of size X.

2. **Case 2:**
   Consider 5 groups and 10 rooms, with the following sizes:

$$\text{Rooms} = \left[\frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}\right]$$

$$\text{Groups} = [X, X, X, X, X]$$

   Here, X grows from 1 to 200,000 in increments of 1. In this case, the $K-$Level of the solution grows from 1 to 100,000, since the optimal solution is constructed by splitting groups of X and putting them into rooms of capacity $\frac{X}{2}$. The BnB algorithm reaches the upper bound in the first depth-first traversal, although it takes 13 ms instead of 3, since there are more rooms, so there are more levels to traverse. This can be seen in Figure 4.3.

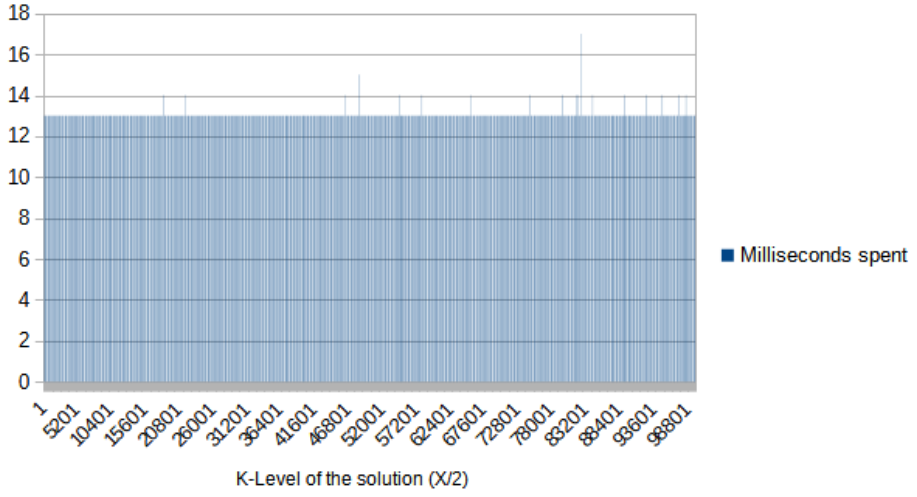


Figure 4.3: BnB algorithm performance for 5 groups and 10 rooms, with room sizes $\frac{X}{2}$.

3. **Case 3:**
   Finally, consider 10 groups and 5 rooms, with the room sizes configured as:

$$\text{Rooms} = [X, X, X, X, X]$$

$$\text{Groups} = \left[\frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}\right]$$

In this case, X grows from 1 to 200,000 in increments of 2. In this case, the $K-$Level of the solution grows from 1 to 100,000, since the optimal solution is constructed by assigning groups of $\frac{X}{2}$. Similar to the previous cases, the BnB algorithm reaches the upper bound in the first depth-first traversal, as can be seen in Figure 4.4.

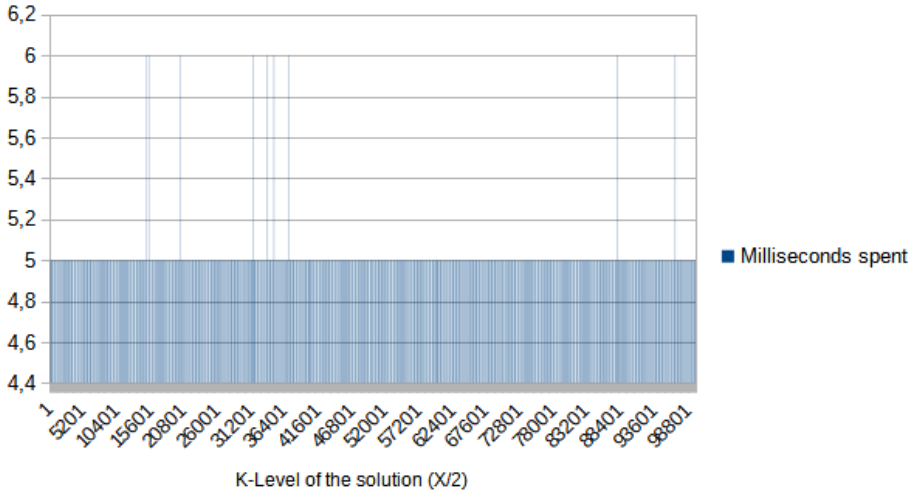


Figure 4.4: BnB algorithm performance for 10 groups and 5 rooms, each of size X.

These examples illustrate how the upper bound termination mechanism in BnB allows the algorithm to handle large inputs efficiently by pruning non-promising branches early in the computation.

4.3.2. Cases where the upper bound is unreachable

To explore a case where the upper bound is unreachable as input size increases, we use the following configuration:

$$\text{Rooms} = [X, X, X, X, 2X]$$

$$\text{Groups} = [X + 2, 2X, 3X - 2]$$

Here, the smallest group, $X + 2$, fits in the last room, so the upper bound is set at $X + 2$. However, the highest achievable K-Level is always $X - 2$, since $\sum \text{Groups} = \sum \text{Rooms} = 6 \cdot X$. Therefore, after placing all groups into rooms, there will be no space left. To reach the highest K-Level, we must also avoid splitting the smallest group. The distribution yielding the highest K-Level will be:

$$\text{Assignments} = \begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ X & X & 0 & 0 & 0 \\ 0 & 0 & X & X & X-2 \end{bmatrix}$$

Where the X values of groups 2 and 3 can be rearranged freely. The plot results of this configuration can be seen in Figure 4.5.

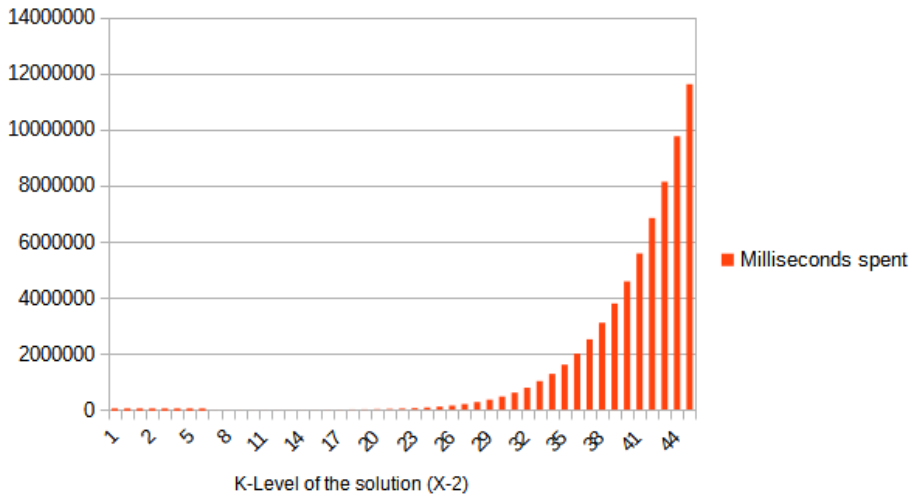


Figure 4.5: BnB algorithm performance for 3 groups and 5 rooms, with varying sizes.

Here, X ranges from 3 to 47 in increments of 1. As we have already mentioned, the $K-$Level of the solution will be $X - 2$.

In this case, the cost arises from having an unreachable termination condition. Keeping in mind the mechanisms of the BnB algorithm, after finding a solution of $X - 2$ quality, the algorithm will need to explore and discard every partial assignation of groups of $K-$Level $\geq X - 1$. The plot of nodes explored in this same setting is shown in Figure 4.6.

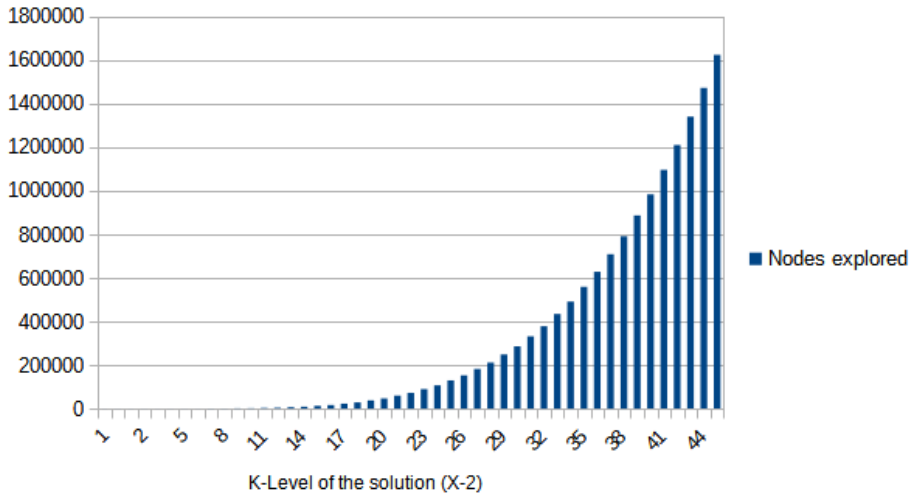


Figure 4.6: Exponential growth of the search tree due to increasing X, highlighting the difficulty of pruning.

This illustrates that even with pruning, the computational time remains high due to the hardness of the problem and the exhaustive approach. Therefore, we turn to polynomial, non-exhaustive solutions, such as *Genetic Algorithms*.

CHAPTER 5

Solving via Genetic Algorithm

5.1. INTRODUCTION TO GENETIC ALGORITHMS

Genetic algorithms (GAs) are a class of optimization algorithms inspired by the principles of natural selection and genetics, as formulated by Charles Darwin. These algorithms operate by evolving a population of potential solutions through iterative processes that mimic biological evolution, such as selection, crossover, and mutation.

The core idea behind GAs is to evolve solutions to optimization problems by favoring the survival and reproduction of the fittest individuals, thereby gradually improving the quality of solutions over successive generations. This evolutionary approach is particularly well-suited for complex problems where traditional optimization techniques may fall short and exhaustive algorithms take too long.

5.2. MECHANISMS OF GENETIC ALGORITHMS

GAs begin with an initial population of potential solutions, known as chromosomes or individuals. Each individual is evaluated using a fitness function, which quantifies how well it solves the problem at hand. The fittest individuals are more likely to be selected for reproduction, where they generate offspring through crossover and mutation. Additionally, mechanisms like gene repair can be added.

A Steady-State GA [8] is a GA where only one offspring may be generated, as opposed to a Generational GA, where as many as the original population size may be generated. After it is generated through crossover, mutation and gene repair; the offspring is compared against the weakest solution, if it is fitter, it will take its place in the population. Steady-state GAs are good at maintain feasibility in problems with hard constraints, while still exploring thoroughly the exploration space, we will discuss this in Section 5.3.

Essentially, Algorithm 5.1 is the pseudo-code skeleton of a Steady-State Genetic Algorithm where only one offspring is generated per generation. It resembles the general scheme in [9], but we adapt it to our needs by being ambiguous over the method of parent selection and by only creating one child per generation. We also add gene repair after the mutation:

Algorithm 5.1: Steady-State Genetic Algorithm

```
Input: A vector of solutions population
Output: The best solution found
Function GeneticAlgorithm(population):
    for generation ← 0 to MAX_GENERATIONS do
        parent1 ← SelectParent(population);
        parent2 ← SelectParent(population);
        child ← Crossover(parent1, parent2);
        Mutate(child);
        Repair(child);
        ReplaceWeakest(population, child);
    end
    return GetBestSolution(population)
```

In a case where there is no gene repair, the call to function `Repair(solution)` is a *Noop*.

5.2.1. Encoding Of The Solution Individuals

In our approach to solving the $K-$Anonymity problem, we utilize a structured representation of the chromosomes (candidate solutions) as a 2D Matrix. This matrix provides a clear and organized way to represent the assignment of individuals from various groups to different rooms, adhering to the problem's constraints and requirements.

Components of the Solution Matrix

In our problem instance we receive:

- G, the set of K-Anonymous groups, where each group in G has a specific number of individuals, denoted as $g_i \in \mathbb{N}$.
- R, the set of rooms, where each room in R has a defined capacity, denoted as $r_j \in \mathbb{N}$.

The matrix M representing a possible solution (chromosome) has dimensions $n \times m$, where:

- n represents the total number of groups, that is $|G|$.
- m represents the total number of rooms, that is $|R|$.

Each element of the matrix M is denoted as c_{ij}, where:

- i indexes the groups ($g_i \in G$).
- j indexes the rooms ($r_j \in R$).

Thus, c_{ij} denotes the number of individuals from group g_i assigned to room r_j, with $c_{ij} \in \mathbb{N}$. All values r_j, g_i, and c_{ij} are natural numbers.

The solution matrix provides a structured framework for managing and evaluating group-room assignments throughout the algorithm's execution. Over this matrix, it is simple to ensure adherence to constraints and facilitating evaluating the assignments based on a fitness function.

5.2.2. Selection

Selection is the process of choosing individuals from the population to create offspring. In our algorithm, we use a **binary tournament selection method**, where two individuals are randomly chosen, and the one with the better fitness score is selected as a **parent individual**. This process is repeated to select another parent. The binary fitness score comparison ensures that fitter individuals have a higher probability of being chosen and reproduce.

5.2.3. Crossover

Crossover, also known as recombination, combines the genetic material of two **parent individuals** to create an offspring. Our algorithm employs a two-point crossover method. Two random points within the chromosome are selected, and the rows between these points are swapped between the parents to produce a new individual. This method respects the integrity of group assignments, while it may not ensure respecting the room capacities in our solution. By crossing over group assignments, we preserve the total number of people per group, reducing the likelihood of violating group constraints compared to crossing over room assignments in our resulting populations.

Crossover Example

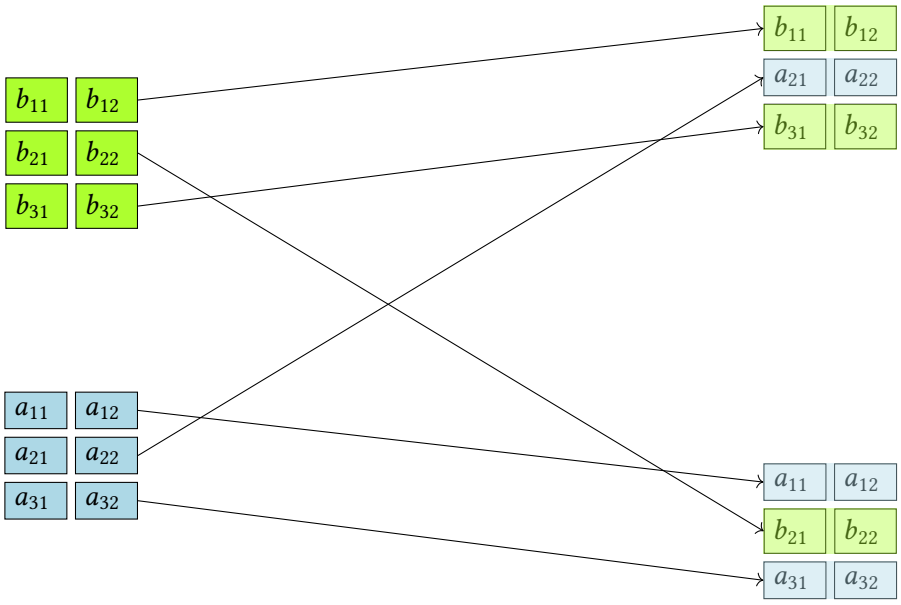
The two-point crossover method involves choosing two random crossover points and swapping the rows between these points to produce an offspring. Consider two parent matrices, M_1 and M_2, each with dimensions 3 *Groups* $\times$ 2 *Rooms*. We perform a two-point crossover where the row in the middle is selected for swapping between the two parent matrices, meaning the offsprings will have the assignations of groups g_1 and g_3 from one parent, and the assignations of g_2 from the other. In this case, our two points for crossover are 1 and 2.

Diagram of Two-Point Crossover

Let M_1 and M_2 be the two parent matrices defined as follows:

$$M_1 = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{bmatrix} \quad M_2 = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Let us say that the points of crossover are 1 and 2. For two 3×2 matrices, this means that the only row involved in the crossover is column 2. This can be illustrated as follows:



The resulting offspring matrices after crossover are:

Offspring 1 Offspring 2

$$\begin{bmatrix} a_{11} & a_{12} \\ b_{21} & b_{22} \\ a_{31} & a_{32} \end{bmatrix} \quad \begin{bmatrix} b_{11} & b_{12} \\ a_{21} & a_{22} \\ b_{31} & b_{32} \end{bmatrix}$$

Since we only create one offspring individual, we can say that, depending of the random order in which $M1$ and $M2$ are picked from our population, we will end up with the first or second offspring for this two-point crossover.

5.2.4. Mutation

Mutation introduces random changes to individuals, promoting genetic diversity and enabling the algorithm to explore new regions of the solution space. In our implementation, we introduce two types of mutations that we will compare against each other: random mutation and heuristic-guided mutation. Mutation itself is a random occurrence as well, a gene typically has a certain probability of exhibiting a mutation. The probability chance will be decided and justified further ahead.

Random Mutation

The random mutation assigns a random number of people from a random group to a random room. This approach is a usual method in genetic algorithms, where some cell is given a random value with correct encoding. In this case we will consider a value going from 0 until the size of the group from which patients are, or the total capacity of the room, whichever is smaller; as the correct encoding. The randomness introduces genetic diversity and allows the algorithm to explore a wide range of potential solutions.

Heuristic-guided Mutation

The heuristic-guided mutation, however, uses a more informed approach, similar to how spiders design their webs based on the knowledge of their environment. Spiders know that certain insects fly at specific heights and build their webs to maximize catch rates. For instance, the golden silk orb-weaver's large, durable webs can trap a wider range of prey, like moths, giving them an advantage in habitats with plentiful but large insect populations. These evolutionary web-building strategies demonstrate how species can out-compete others by leveraging specialized, informed approaches to survival.

Similarly, our mutation uses probabilities to mutate the people assigned from a random group g_i to a random room r_j that reflect knowledge of how a good solution of the problem looks like:

- 25% probability of assigning 0 people to cell c_{ij}.
- 50% probability of assigning the maximum possible group count to cell c_{ij}.
- 25% probability of assigning a random number of people to cell c_{ij}.

This informed mutation leverages domain knowledge to produce more effective solutions than purely random assignments. This is because, generally, the less split the groups are, the higher the $K-$Level will be.

5.2.5. Gene Repair

After applying mutation or crossover, the resulting gene (or chromosome) may violate the constraints over the result of the problem. To address this, gene repair mechanisms can be employed to ensure that the solutions remain valid. Gene repair is a post-processing step that modifies the gene to correct any infeasibilities introduced by the genetic operators.

The effectiveness of these repair strategies has been demonstrated in various studies. For instance, Mitchell et al. proposed the GeneRepair operator for the TSP problem.[10].

Gene Repair in the TSP

The Traveling Salesman Problem (TSP) is a classic optimization problem that involves finding the shortest possible Hamiltonian cycle in a graph, where the cycle visits each vertex (city) exactly once and returns to the starting vertex.

In the context of the TSP, a valid solution is a permutation of cities where each city is visited exactly once. When mutation or crossover disrupts this permutation, gene repair can be used to restore validity, instead of hoping that the evolutionary process will result in better solutions.

The process of gene repair for the TSP involves the following steps, as described by Mitchell et al: "In practice, GeneRepair examines each tour in turn, enforcing the following:

1. Correct number of cities in the tour,
2. No duplicate cities,
3. No missing cities" [10].

This creates better solutions without hardly sacrificing relevant exploration space, although the cost of repairing these genes is in $O(n)$, where n is the size of the hamiltonian cycle.

Gene Repair in $K-$Anonymity

In our case, after mutation and crossover, we fix a problematic group. This will help converge into better solutions. A problematic group is a row of our solution matrix that is overassigning or underassigning patients from a group to some rooms.

To fix up a problematic group in our solution, we:

1. **Identifying a Random Problematic Group**: Iterate through all patient groups to identify those whose current size does not match the required group size. Store these groups in a list, then pull one at random.
2. **Underassignment Handling**:
   - If a group is underassigned, collect a list of rooms with available capacity.
   - Randomly select a room from this list and move as many patients as possible from the group to the room, updating the relevant data structures.
   - If a room becomes full after the adjustment, remove it from the list of available rooms.
   - Repeat this process until the group's size matches the required size or no more rooms are available.
3. **Overassignment Handling**:
   - If a group is overassigned, collect a list of rooms that currently have patients.
   - Randomly select a room from this list and reduce the number of patients from the group in that room, updating the relevant data structures.
   - If a room becomes empty of patients from the group, remove it from the list of occupied rooms.
   - Repeat this process until the group's size matches the required size or no more rooms are available.

This scheme is applied in Algorithm 5.2.5:

Gene Repair Algorithm:

Algorithm 5.2: Repair Group Assignments in Gene

Input: patientsPerGroup: Required number of patients per group,
roomCapacities: Capacity of each room,
assignments: Assignment matrix of the solution individual
Output: Individual with one wrong group now sticking to invariants of a valid solution

badGroups $\leftarrow \{i \mid i \in [0, \textbf{PATIENTS\_GROUPS} - 1]$ and **getPatientGroupCount**$(i) \neq$ **patientsPerGroup**$[i]\}$
if ***badGroups*** *is empty* **then**
  return;
end
Select a random group *wrongGroup* from **badGroups**;
if ***getPatientGroupCount***$(wrongGroup) >$ ***patientsPerGroup***$[wrongGroup]$ **then**
  availableRooms $\leftarrow \{r \mid r \in$ $[0, \textbf{NUMBER\_ROOMS} - 1]$ and **getRoomCapacity**$(r) >$ **peoplePerRoom**$[r]\}$
  while ***getPatientGroupCount***$(wrongGroup) \neq$ ***patientsPerGroup***$[wrongGroup]$ **do**
    if ***availableRooms*** *is empty* **then**
      break while;
    end
    Select a random room *targetRoom* from **availableRooms**;
    availableCapacity $\leftarrow$ **getRoomCapacity**$(targetRoom)$ $-$ **peoplePerRoom**$[targetRoom]$;
    patientsToMove $\leftarrow \min($**getPatientGroupCount**$(wrongGroup) -$ **patientsPerGroup**$[wrongGroup], availableCapacity)$;
    updateDataStructures(wrongGroup, targetRoom, **assignments**[wrongGroup][targetRoom] + patientsToMove);
    if ***peoplePerRoom***$[targetRoom] \geq$ ***getRoomCapacity***$(targetRoom)$ **then**
      Remove *targetRoom* from **availableRooms**;
    end
  end
end
else if ***getPatientGroupCount***$(wrongGroup) <$ ***patientsPerGroup***$[wrongGroup]$ **then**
  roomsOccupied $\leftarrow \{r \mid r \in [0, \textbf{NUMBER\_ROOMS} - 1]$ and $\exists g$ where **assignments**$[g][r] > 0\}$
  while ***getPatientGroupCount***$(wrongGroup) \neq$ ***patientsPerGroup***$[wrongGroup]$ **do**
    if ***roomsOccupied*** *is empty* **then**
      break while;
    end
    Select a random room *targetRoom* from **roomsOccupied**;
    newAmount $\leftarrow \max($**assignments**$[wrongGroup][targetRoom] -$ (**patientsPerGroup**$[wrongGroup] -$ **getPatientGroupCount**$(wrongGroup)), 0)$;
    updateDataStructures(wrongGroup, targetRoom, newAmount);
    if ***assignments***$[wrongGroup][targetRoom] == 0$ **then**
      Remove *targetRoom* from **roomsOccupied**;
    end
  end
end

While gene repair may potentially result in loss of quality in the solution, since repaired groups accounting for room space tend to be split more, it will result in solutions that violate less constraints,

which mean they can be implemented.

In Appendix D some discarded alternative, very costly algorithms for the gene reparation are presented.

5.2.6. Fitness Score

The fitness score is a crucial component of the GA, guiding the selection, crossover, and mutation processes. It evaluates how well an individual meets the problem's constraints and objectives. Our fitness score must consider three factors:

- Number of rooms over capacity.
- Number of groups with incorrect patient counts.
- Minimum number of people assigned from any group to any room ($K-$level).

Usually fitness scores are calculated as an arithmetic number obtained from normalizing an objective function and using mechanisms as penalty functions over the solution. In our case, since we are using a binary tournament selection process, we can make our fitness score a deterministic operator which, given two solutions, determines which one is the fitter.

We define our fitness score as a less-than operator ('operator<'), enabling straightforward comparison between **two** individuals in the following manner:

- **Room constraint:** Solutions are first compared based on *rooms exceeding max capacity*. Fewer rooms exceeding capacity is better, so a solution with fewer rooms exceeding capacity is considered fitter.
- **Group constraint:** If the number of rooms exceeding capacity is the same, the solutions are compared based on wrong $K-$group counts. Similarly, fewer $K-$groups that do not add up to the original group counts signal a better solution. A solution with fewer groups with wrong count is preferred.
- **K−Level:** If both previous elements are equal, then the solutions are compared based on $K-$level, meaning solutions achieving a higher level of K-anonymity are considered fitter.

5.3. JUSTIFICATION OF GENETIC ALGORITHM DESIGN CHOICES

The design of our GA incorporates several key decisions to address the specific constraints and objectives of the problem:

5.3.1. Steady-State GA vs. Generational GA

$K-$Anonymity presents particularly challenging constraints, such as group constraints that can be violated during mutation and room constraints that can be violated during crossover. These constraints become even more difficult to adhere to when the number of patients matches the room capacities.

We choose a Steady-state GA in particular because its incremental approach improves gradually a population of solutions while minimizing the computational overhead associated with constraint checks. By replacing only the weakest individual with a newly generated offspring, the steady-state GA improves gradually the quality of the population and helps avoid premature convergence. This method results in a more careful exploration of the search space and adapts well to the tight constraints of the $K-$Anonymity problem.

In contrast, generational GAs replace the entire population with new offspring in each generation, which can disrupt the solution space and result in the loss of feasible solutions. This approach tends to be more computationally expensive due to the need to evaluate and manage the feasibility of a complete new population. Moreover, generational GAs are more prone to premature convergence because the abrupt replacement of the entire population can lead to reduced diversity and early

stagnation. For $K-$Anonymity, where feasibility is critical, generational GAs might not be as effective since replacing the whole population can be detrimental when constraints are not fully met. This method risks converging on suboptimal solutions quickly, leading to inbreeding and stagnation.

The focus of the GA must be on achieving viable solutions in $K-$Anonymity. Because of this we use a steady-state GA with a fitness operator that prioritizes constraint adherence. By steadily evolving solutions and repairing genes through generations, this approach effectively maintains and improves solution viability, making it a better fit for the problem's demanding constraints than a Generational GA.

5.3.2. Group vs. Room Crossover

We chose to perform crossover operations based on group assignments rather than room assignments. This decision is driven by the need to maintain the integrity of the total number of people in each group. Violating the group constraint is more detrimental than room constraint violations, as groups must sum exactly to their initial totals, whereas rooms must only avoid exceeding capacity.

5.3.3. Informed Mutation Strategy

The heuristic-guided mutation strategy is designed to leverage knowledge about the problem. By favoring full group assignments and zero assignments, and reducing the occurrence of partial group assignments, we produce more feasible and effective solutions that converge to higher $K-$levels. Later, we will compare both versions of the mutation to illustrate the difference in performance achieved.

5.3.4. Fitness Score and Selection Process

A fitness score expressed as a simple operator that holds the three aforementioned pieces of information is a simple and effective way to guide the search that reflects the problem's constraints and objectives. We consider that any solution that cannot be implemented is not as good as any other which can. This attempts to ensure the constraints are met by prioritizing them over the $K-$Level when comparing two different individuals.

It facilitates the binary tournament selection, replacement of the weakest individuals, and determination of the best solution, since there is no need to compare one solution to all others, or do any fitness, roulette-based probabilities for selection. All needs of our algorithm can be met with an operator that is able to face one individual against another.

Also it is an illustrative design choice, since we can always know the number of constraint violations and the $K-$Level in our solution, and thus, the average constraint violations and $K-$Level of our population, instead of having to derive it from a numerical calculation. This means that data plots derived from our algorithm are easier to read and accurate to the evolution of our population throughout generations.

5.3.5. Gene reparation

While unrepaired approaches can result in solutions with higher $K-$Levels, we will observe that, for more complex cases, they usually result in solutions that incur in constraint violation.

By our definition of the Fitness score, a solution that has any constraint violation will be less fit than any one that does. Therefore, while gene repair results in longer computation times, it helps find solutions that are feasible and satisfy all constraints, and fish for higher $K-$Levels.

It is a mechanism that, in solution instances where there total capacity is equal to the amount of patients, it may create a population with zero constraint violations that has a lot of trouble to reach higher $K-$Levels due to the nature of the mechanism (i.e. it splits some group while adhering to room capacities) and the nature of the problem (i.e. only the smallest assignation matters). But for cases where the total capacity is greater than the amount of patients to assign, it is an effective

mechanism to reach zero group constraint violations quickly and start finding better solutions from there.

5.4. GENETIC ALGORITHM CONFIGURATIONS AND RESULTS

In this section we will discuss the three main parameters of our GA: Population size, mutation chance and number of generations. The population size is the number of individuals in our population, simply put. The number of generations is the amount of times we will repeat the GA's cycle, as shown in Algorithm 5.1. The mutation chance is a probability, given as a percentage probability of executing our mutation mechanism in the child gene.

5.4.1. Configuration choice: Generations, mutation chance and population size

There are three significant configuration parameters for our Steady-State GA to decide upon: number of generations, mutation chance of the child individual and size of the population throughout the execution of the GA.

To decide these parameters we will introduce two types of instances of the $K-$Anonymity problem:

1. **Restrictive instances**: In these instances $\sum \text{Groups} = \sum \text{Rooms}$. They are called restrictive since a mutation on a gene that has 0 constraint violations will create at least 1 constraint violation.
2. **Non-restrictive instances**: In these instances $\sum \text{Groups} < \sum \text{Rooms}$. They are non-restrictive since a mutation on a gene that has 0 constraint violations may create another gene with 0 constraint violations.

The difference between these two is of high importance to our GA, since we have not created a numerical fitness function. Our fitness function will prioritize solutions without constraints, so it has a harder time evolving in restrictive instances, since there is no trade-off between, on the one hand, a solution with a high $K-$Level and a slightly higher number of violations, and on the other hand a solution that has a low $K-$Level and no constraint violations, the latter will always be considered stronger.

Because of this, to decide the values of these three parameters we will use *non-restrictive instances*. Again, we will extract the results using the configuration listed in Annex A and the GA implementation listed in Annex C.

1: Mutation chance

In this section, we choose 400 as population size and 2,000,000 generations in order to see which mutation chance is best for such a configuration. We will refine the population size and the number of generations later. We choose 400 since throughout development we have seen this is one that works well, and 2,000,000 generations since it is a large number in order to make sure we let the algorithm evolve as much as it needs, with the goal of reducing that number when we refine the other two parameters.

Let us consider the following non-restrictive case:

$$\begin{aligned} \text{Rooms} &= [100, 100, 100, 100, 100, 100] \\ \text{Groups} &= [28, 72, 44, 56, 48, 52, 44, 56, 36, 64] \\ \text{where } \sum_{i=1}^{6} \text{Rooms}_i &= \sum_{j=1}^{10} \text{Groups}_j + 100 \end{aligned}$$

Where we have extra room for 100 patients, since all groups can be placed, assigned in consecutive pairs, in each room while leaving one room free. This case has a maximum $K-$Anonymity level of 26, the smallest group size. In Figure 5.1 we can observe the results of different mutation probabilities.

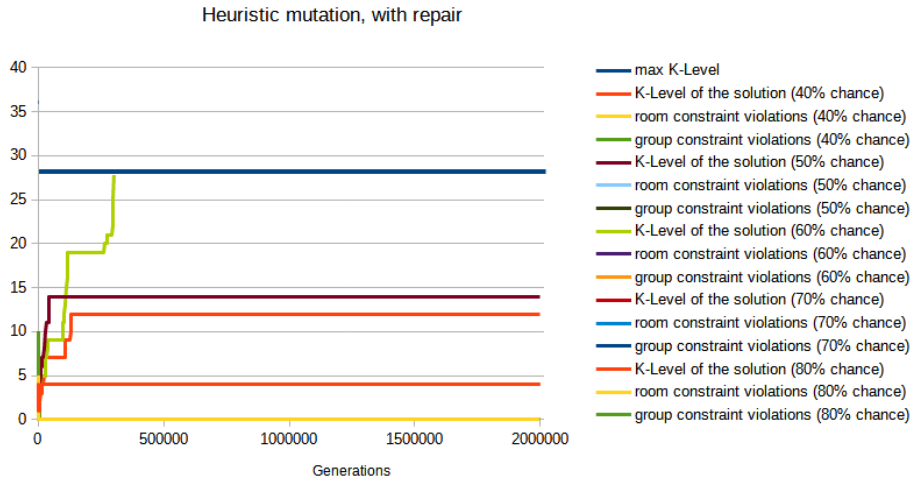


Figure 5.1: Metrics of evolution for different mutation probabilities of a Steady-State GA with gene repair and heuristic-guided mutation in a case with big groups and small rooms

This case favors the heuristic mechanism because we can always fit the total size of the group in any given room. We observe that the best-performing version exhibits a 60% probability of mutation. Although this is a high mutation rate, it can be attributed to the steady nature of the algorithm.

The algorithm ensures that only one individual is replaced per generation, allowing for gradual but consistent exploration of the solution space. This gradual replacement reduces the risk of losing valuable genetic material, trying to ensure constraints are kept, thereby justifying the elevated mutation rate. The high mutation probability introduces variability into the population, which is crucial to increase the $K-$Level.

Moreover, the ability to consistently fit the total group size into any room indicates that the heuristic is effectively managing the constraints of the problem space in mutation. This efficiency highlights the robustness of the algorithm in maintaining feasible solutions while exploring new possibilities. It seems 60% probability of mutation offers a good balance between exploration (through mutation) and exploitation (through steady state replacement). It seems that higher mutation rates are detrimental to the search, exhibiting more trouble in finding a better solution.

We can see all of them also drop the constraint violations thanks to the gene repair mechanism. This is crucial since our focus is on the viability of our solutions. It is less detrimental to have solutions with lower $K-$Levels than solutions that violate the constraints of our instance.

Now we will consider the following non-restrictive case:

$$\begin{aligned} \text{Rooms} &= [85, 100, 100, 100, 100, 100] \\ \text{Groups} &= [100, 100, 100, 100, 100] \\ \text{where } \sum_{i=1}^{6} \text{Rooms}_i &= \sum_{j=1}^{5} \text{Groups}_j + 85 \end{aligned}$$

In this case, the algorithm faces a similar situation to the previous case, where one room is useless to the solution, only in this case using it is also detrimental. The first room, with a capacity of 85, presents local optima with which the algorithm might prematurely settle, attempting to split a group

of 100. Our GA must overcome these local optima to achieve an optimal solution. In Figure 5.2 we can observe the results achieved by different mutation rates.

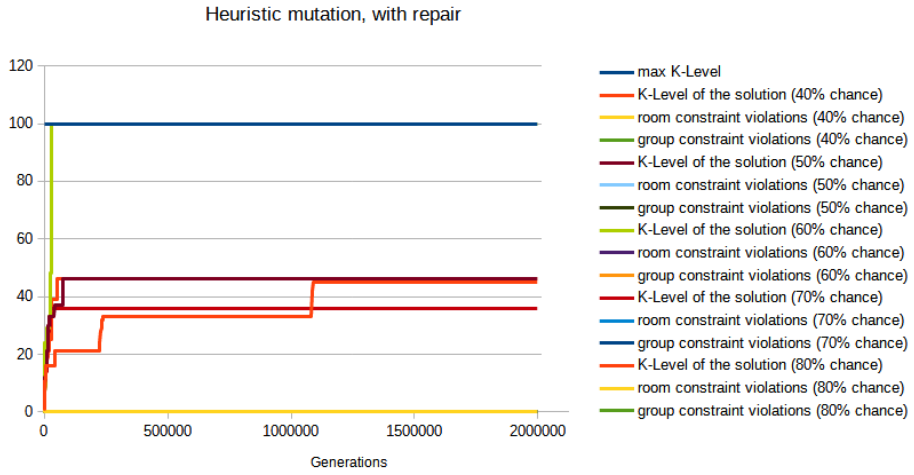


Figure 5.2: Metrics of evolution for different mutation probabilities of a Steady-State GA with gene repair and heuristic-guided mutation in a case with similar groups and rooms

For most mutation probabilities, the algorithm struggles to escape these local optima. However, when the mutation probability is set at 60%, the algorithm overcomes them. This exact mutation rate introduces enough variability to the population, allowing the algorithm to explore solutions that bypass the difficulties posed by the room with 85 capacity and correctly place the groups into the 100 capacity rooms.

Somehow, the 60% mutation probability proves to be particularly effective in this context, as it provides the necessary diversity to find the global optima where all groups are placed perfectly in the 100 capacity rooms. However, it seems that higher probabilities than that become stuck at zero constraints broken and a $K-$Level of 45 too quickly and unable to overcome it. In these solutions, the room with 85 capacity consists of a split of 50 of some group and 35 empty spaces. For the solutions stuck at 35, the solution uses the full capacity of the room with 85 capacity.

Let us now consider the last non-restrictive case:

$$\begin{aligned} \text{Rooms} &= [28, 72, 44, 56, 48, 52, 44, 56, 36, 64, 50, 50] \\ \text{Groups} &= [100, 100, 100, 100, 100] \\ \text{where } \sum_{i=1}^{12} \text{Rooms}_i &= \sum_{j=1}^{5} \text{Groups}_j + 100 \end{aligned}$$

Where we have extra room for 100 patients, since all groups can be placed, split in consecutive pairs, in each pair of rooms without using the first two rooms. This case has a maximum $K-$Anonymity level of 36. In Figure 5.3 we can observe the results of different mutation probabilities.

The heuristic-guided mutation in our GA will try to place as many people as there are in some group, when mutating a random cell. Because no group fits entirely in any given room, in this case the heuristic-guided mutation is providing only incorrect values for our search. The probabilities of mutation chance vary, but there is no probability for gene repair, and gene repair will be applied whenever there is a group constraint violation in our generated child. Because of this, our algorithm can't seem to make great improvements, since most mutations make the solution incur in a group violation, which then is split by the gene repair mechanism. The mechanism for gene repair, when

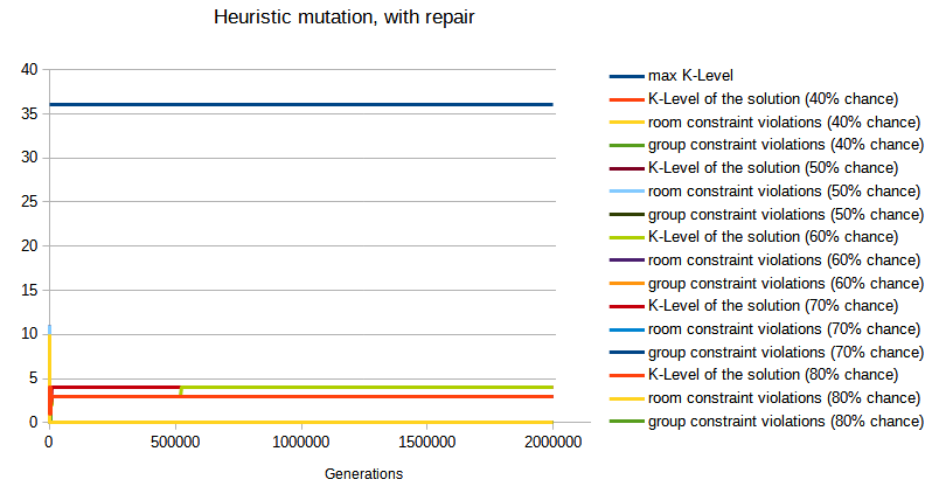


Figure 5.3: Metrics of evolution for different mutation probabilities of a Steady-State GA with gene repair and heuristic-guided mutation in a case with big groups and small rooms

repairing a group, splits it depending on the remaining capacities of the rooms, making it extremely difficult to reach higher $K-$Levels in cases like these.

We can observe that, even in this case, it seems that the only versions to manage some kind of improvement, from 3 to 4, are the probabilities of 60% and 70%. From these three cases we conclude that the probability which strikes the best balance between exploration and exploitation for a configuration with 400 individuals and 2,000,000 generations is 60%.

2: Population size

Now, on the aforementioned non-restrictive cases (that is, one with big rooms and small groups, one with similar groups and rooms and one with small rooms and big groups), we will plot out the difference between 50, 300 and 500 as population sizes. Let the non-restrictive case with small groups and big rooms be:

$$\text{Rooms} = [100, 100, 100, 100, 100, 100]$$
$$\text{Groups} = [28, 72, 44, 56, 48, 52, 44, 56, 36, 64]$$
$$\text{where } \sum_{i=1}^{6} \text{Rooms}_i = \sum_{j=1}^{10} \text{Groups}_j + 100$$

In Figure 5.4 we can observe the results on the case of big rooms and small groups. In this one, the heuristic is guiding the search the right direction, since all groups can be placed entirely in every room.

Again, when the population size is 200 individuals, we can see how it cannot find good solutions due to inbreeding, and higher population sizes always seem to benefit the algorithm. Only in 800 we see some detriment, reaching a solution of a $K-$Level of 27, instead of 28. Because of this we will choose to measure higher population sizes and we will also stop plotting the constraint violations since we know they will be zero. In Figure 5.5 we can observe the results on this case with populations of 400 through 700 individuals.

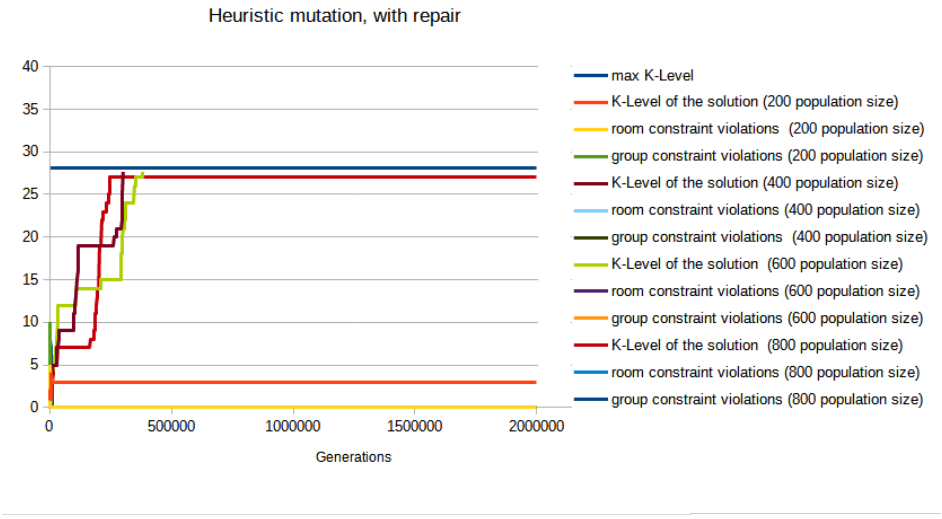


Figure 5.4: Metrics per populations sizes of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with big rooms and small groups

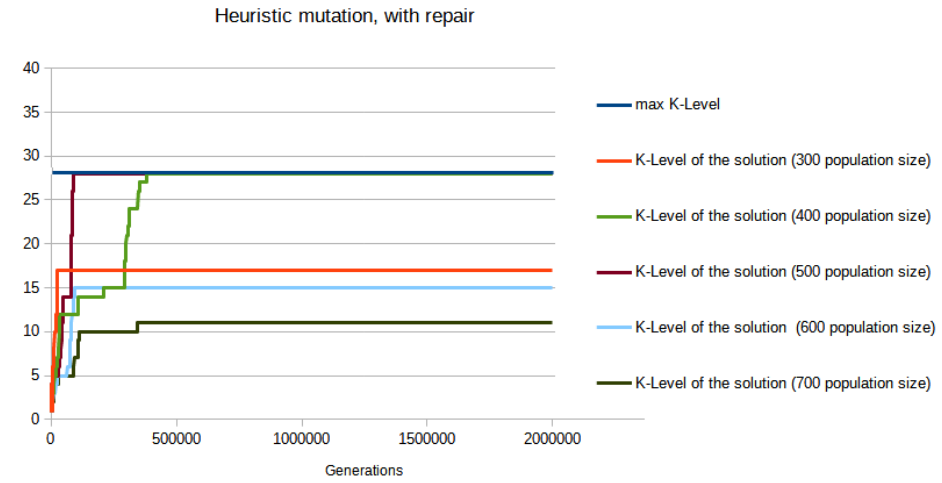


Figure 5.5: Metrics per populations sizes of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with big rooms and small groups

It seems the best results are achieved by a population of 500, which finds global optima faster than 400. The rest of the configurations seems like they are not as good as the one with 400 and 500 individuals. Interestingly, a population size of 500 individuals outperforms a size of 700 in our specific GA scheme. Given the same number of generations, a larger population results in fewer evolutionary steps per individual on average. This suggests that the better performance of the smaller population may come from a more focused exploitation of promising areas within the solution space. In contrast, the larger population size, while enhancing exploration, may dilute the algorithm's ability to intensively refine better solutions.

Let us remember the following non-restrictive case where the rooms are similar to the groups:

$$\text{Rooms} = [85, 100, 100, 100, 100, 100]$$
$$\text{Groups} = [100, 100, 100, 100, 100]$$
$$\text{where } \sum_{i=1}^{6} \text{Rooms}_i = \sum_{j=1}^{5} \text{Groups}_j + 85$$

In Figure 5.6 we can observe the results on the case with similar rooms and groups.

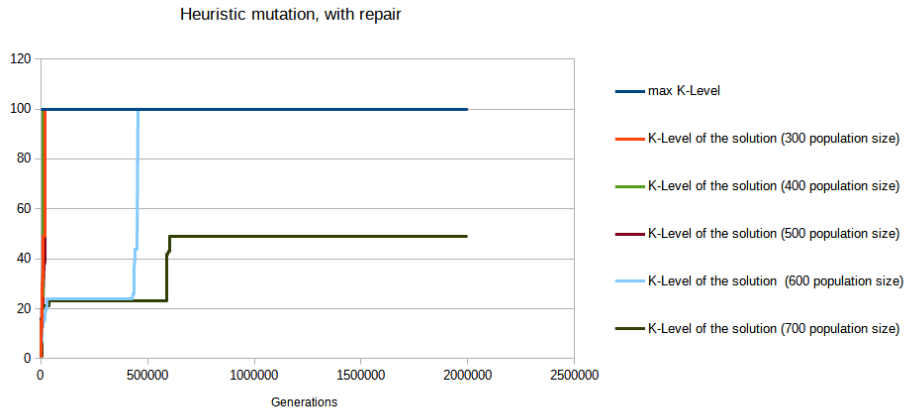


Figure 5.6: Metrics per populations sizes of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with similar rooms and groups

As we can observe, 600 converges way later than the others into the right solution and 700 does not. In Figure 5.7 we have extricated the results of the other population sizes to see which converges first.

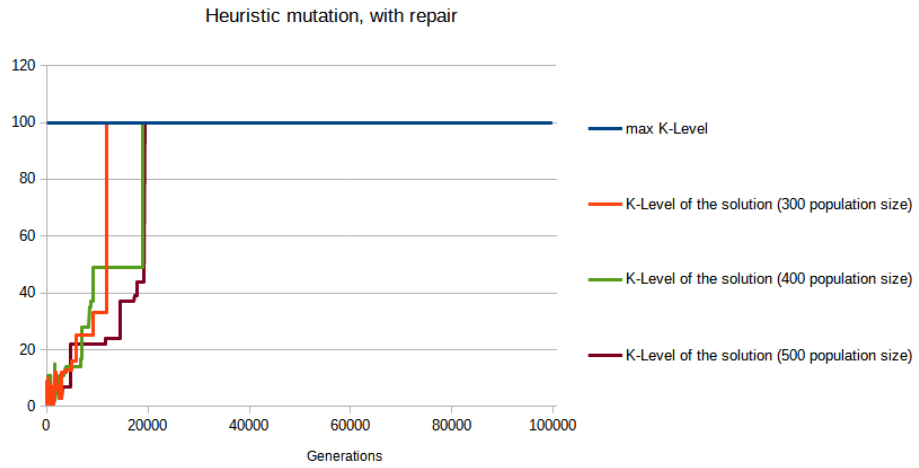


Figure 5.7: Cut off metrics per populations sizes of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with similar rooms and groups

In our experiments, we noticed that when we use a smaller population size, the algorithm finds the best solution faster. This is understandable since the problem is not particularly difficult for our algorithm to solve. However, the results also supports the idea that the ideal population size for more challenging cases should likely be somewhere between 400 and 500.

Now let us now consider the last case:

$$\begin{aligned} \text{Rooms} &= [28, 72, 44, 56, 48, 52, 44, 56, 36, 64, 50, 50] \\ \text{Groups} &= [100, 100, 100, 100, 100] \\ \text{where } \sum_{i=1}^{12} \text{Rooms}_i &= \sum_{j=1}^{5} \text{Groups}_j + 100 \end{aligned}$$

As already mentioned, this is a case that goes against the heuristic, since no group can fit entirely in any given room. The maximum $K-$Level of this case is 36. In Figure 5.8 we can observe the results with population sizes from 300 to 700.

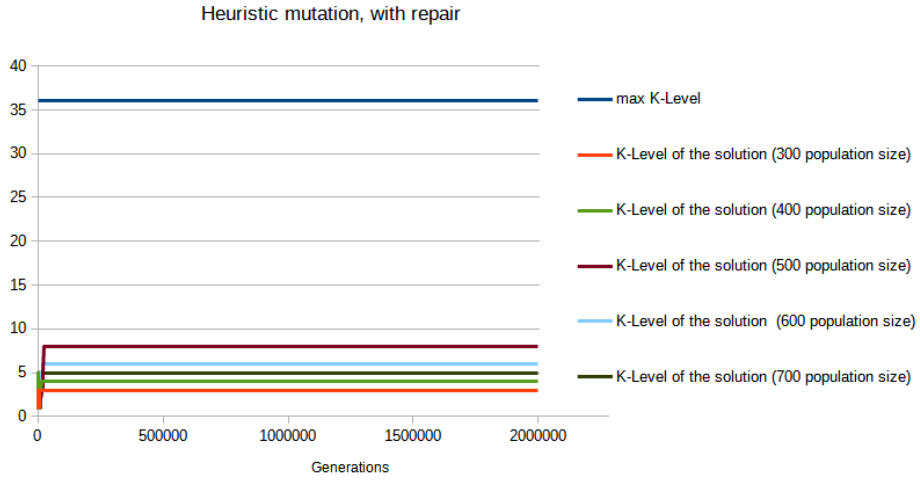


Figure 5.8: Metrics per populations sizes of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with small rooms and big groups

As we have already observed, the heuristic is again hindering the results of our GA in this case. However, we can observe that, even in this case, it seems that the version that manages the best results has 500 individuals. A population bigger than 500 individuals, while offering greater diversity, tends to slow down convergence. This is because larger populations require more generations to effectively reduce constraints and find optimal solutions. The slow convergence can be attributed to the increased number of potential solutions the algorithm must evaluate, which spreads the evolutionary pressure thinner across the population. As we have already mentioned, a smaller population size results in a more focused exploitation of promising areas within the solution space. The larger population size dilutes the algorithm's ability to intensively refine better solutions.

Conversely, a population size smaller than 400 individuals allows for quicker convergence due to the limited genetic diversity, which leads to faster identification of good solutions. However, this rapid convergence often comes at the cost of premature stagnation, as the reduced diversity can cause the population to become homogeneous too quickly, leading to inbreeding and an early plateau at suboptimal K-Levels. This inbreeding effect limits the algorithm's ability to explore new and potentially better solutions, trapping it in local optima.

Although this particular case is not the most representative, as it goes against our GA mechanisms, our analysis across the three scenarios suggests that a population size of 500 strikes the best balance between exploration and exploitation. It provides sufficient diversity to avoid premature convergence while maintaining a reasonable pace of evolution. This balance allows the algorithm to explore a broader solution space without excessively prolonging the convergence process. And while it may

converge more slowly than a population size of 400, but this is not an issue since the computational cost of a GA is polynomial.

We conclude on a 60% mutation chance and a population of 500 individuals for our GA.

3: Number of generations

The first scenario features large rooms and smaller groups:

$$\text{Rooms} = [100, 100, 100, 100, 100, 100]$$
$$\text{Groups} = [28, 72, 44, 56, 48, 52, 44, 56, 36, 64]$$

The results on this case are illustrated in Figure 5.9, which shows the evolution metrics. The second scenario has rooms and groups of similar sizes:

$$\text{Rooms} = [85, 100, 100, 100, 100, 100]$$
$$\text{Groups} = [100, 100, 100, 100, 100]$$

The metrics for this case are shown in Figure 5.10.

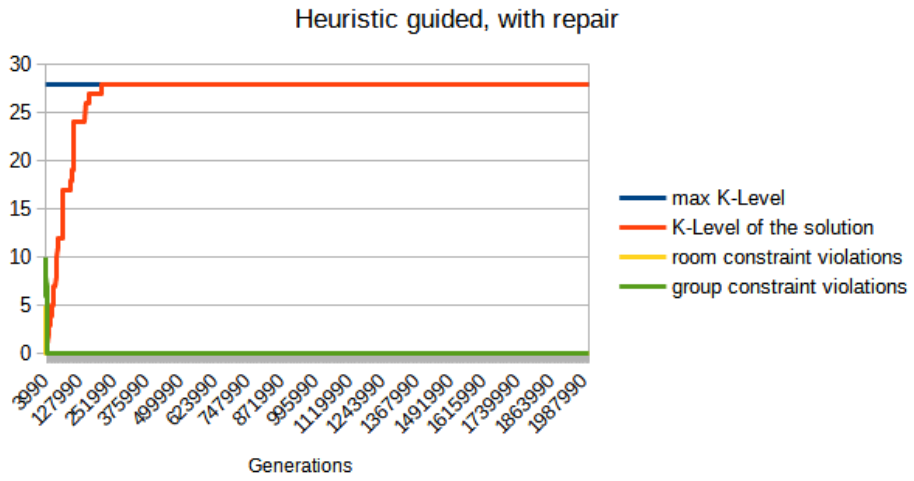


Figure 5.9: Metrics of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with big rooms and small groups

In both these cases we can see that we do not need 2,000,000 generations, in fact, in all of the previous plots in this chapter we have seen that, after 500,000 generations, there are no improvements in population sizes of 400 or 500.

Finally, to not be conservative and give some extra room to our algorithm, we will settle for three quarters of a million generations. In the end, we conclude that our parameters will be:

- **Mutation chance**: 60%
- **Population size**: 500
- **Number of generations**: 750,000

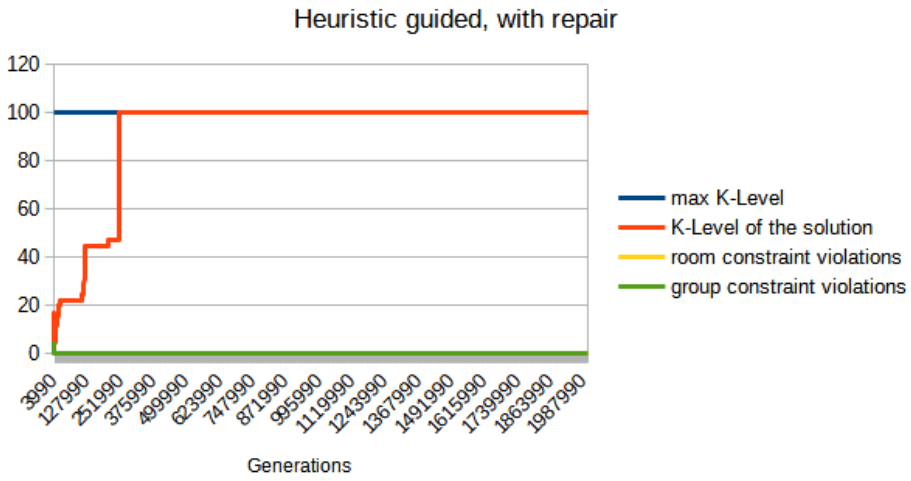


Figure 5.10: Metrics of evolution of a Steady-State GA with gene repair and heuristic-guided mutation in a case with similar rooms and groups

5.4.2. Results for our chosen configuration on growing cases

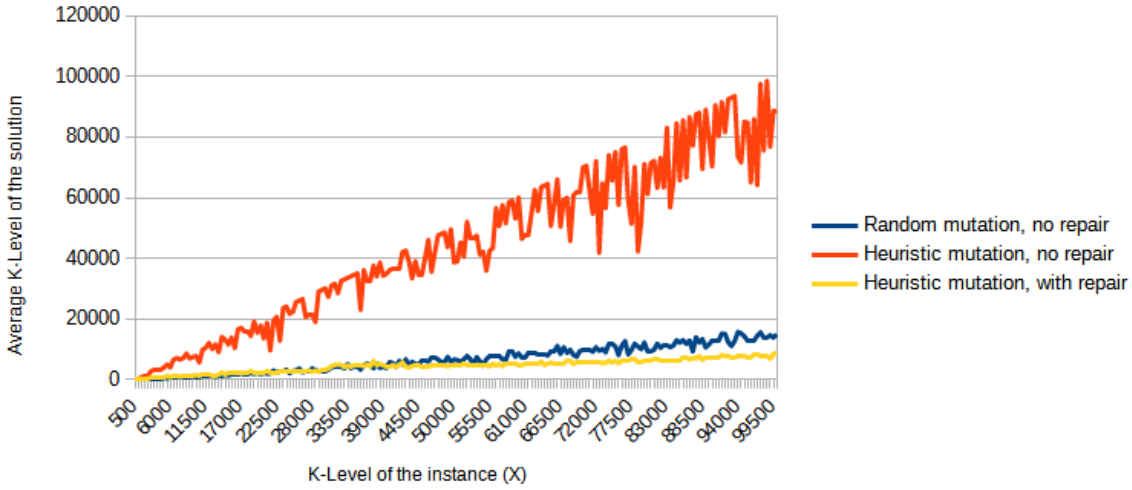
Recalling the different instances, we will create tests for three cases. Each case will have a restrictive instance and a non-restrictive instance. The non-restrictive instance will be generated from the restrictive instance by adding some extra room for a certain amount of patients that will be greater than the smallest patient group.

Recalling the different design choices, we chose some heuristic-guided mutation and gene repair for a Steady-State GA. Bear in mind we will use a static configuration of 750,000 generations, a population size of 500 and a mutation chance of 60%. In order to compare the version that employs these mechanisms to the ones that do not, we will execute these versions on the following cases:

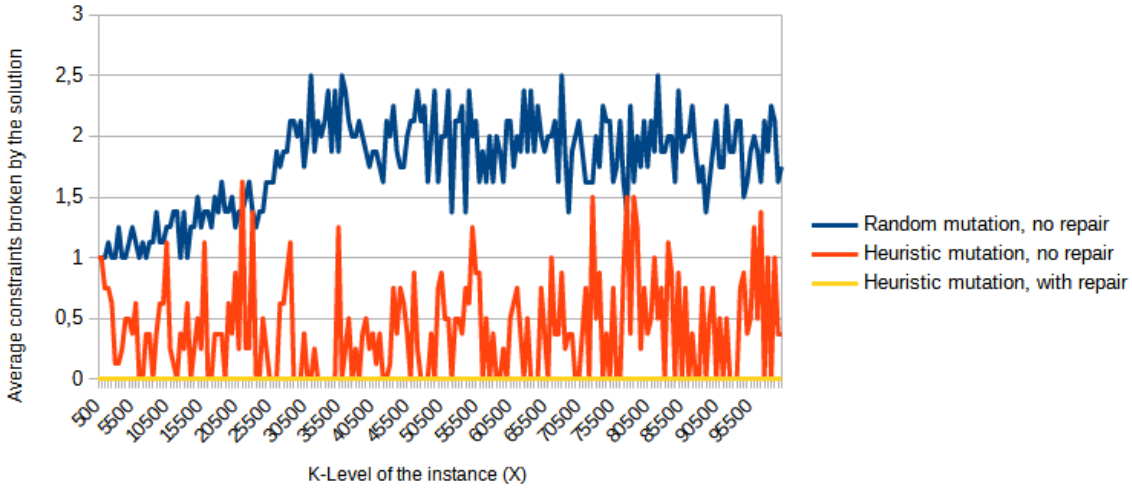
1. **Case 1, restrictive:**
   Consider 5 groups and 5 rooms, each with size X. The room configuration is as follows:

   $$\text{Rooms} = \text{Groups} = [X, \ldots, X]$$

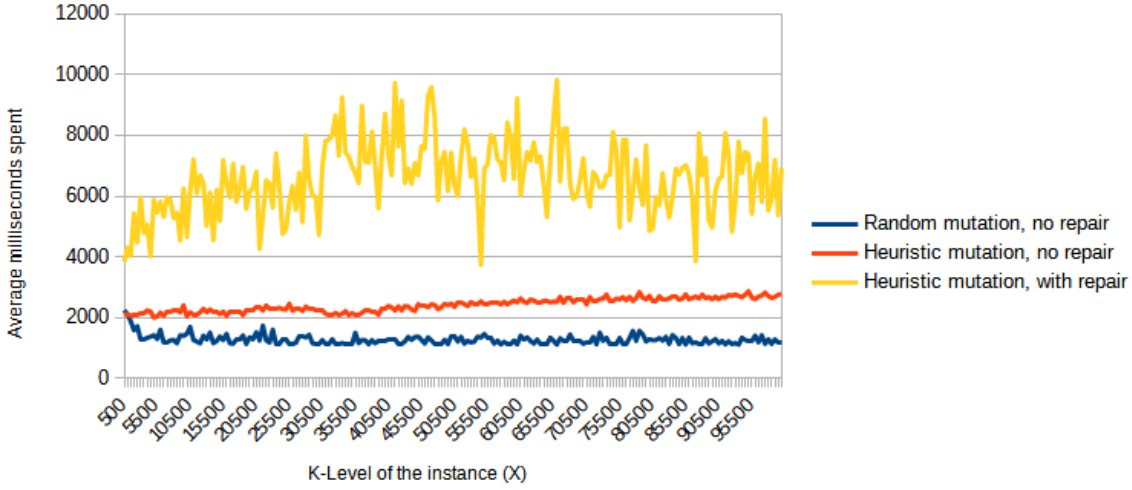
   In this scenario, X grows from 500 to 100,000 in increments of 500. See sub-figures in Figure 5.11.



(a) K-Level obtained per GA version for instances of 5 groups and 5 rooms, each of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 5×5. For 5 groups and 5 rooms of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for 5 groups and 5 rooms, each of size X (average of 10 runs per point)

Figure 5.11: Comparison of GA versions for different metrics across instances of 5 groups and 5 rooms, each of size X.

As we can see in Figure 5.11a, the maximum $K-$Level is achieved by the GA with heuristic-guided mutation and no repair. Unfortunately, we can see in Figure 5.11b that the results of this GA usually include room or group violations. Because of this, more than half of the results given by this GA cannot be implemented in practice. This problem is one where we are extremely interested in enforcing the constraints, since there are 10 possible constraint violations, one per row and column each. As we can see, our final version of the GA (with heuristic-guided mutation and repair) has no room or group violations at any point, while the alternatives without repair sit at 2 or 0.5 violations each 10 runs, respectively.

As we already mentioned, the $K-$Level of solutions generated this way, when there are many constraints to satisfy, will drop. i.e: for a case where a $K-$Level of 10.000 is reachable, we achieve a $K-$Level of roughly 800. This is because the splitting mechanism of the gene repair operation tends to split the groups, instead of finding a solution where each entire group went to some given room.

Lastly, in 5.11c we can see the overhead costs of repairing some row on the child gene. These costs increase the execution time of our GA, but, as we have already mentioned, this is worth it just because the solutions don't incur in constraint violations.

2. **Case 1, non-restrictive:**

Consider 5 groups and 6 rooms, each with size X. The room configuration is as follows:

$$\begin{aligned} \text{Rooms} &= [X, X, X, X, X, X] \\ \text{Groups} &= [X, X, X, X, X] \\ \text{where } \sum_{i=1}^{6} \text{Rooms}_i &= \sum_{j=1}^{5} \text{Groups}_j + 100 \end{aligned}$$

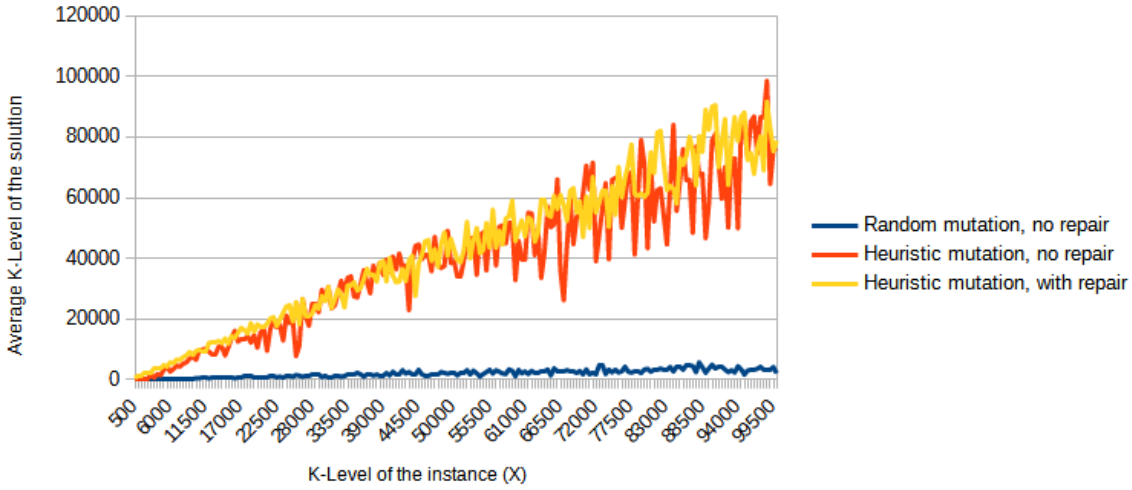
As we can see, this case is not restrictive, since there is an extra room with 100 extra spaces. In this scenario, X grows from 500 to 100,000 in increments of 500. See sub-figures in Figure 5.12.

In Figure 5.12a we can observe how having a less restrictive instance helps our GA with a heuristic-guided mutation and gene repair converge to the highest $K-$Levels while having 0 constraint violations. It does this while also violating 0 constraints in every execution.

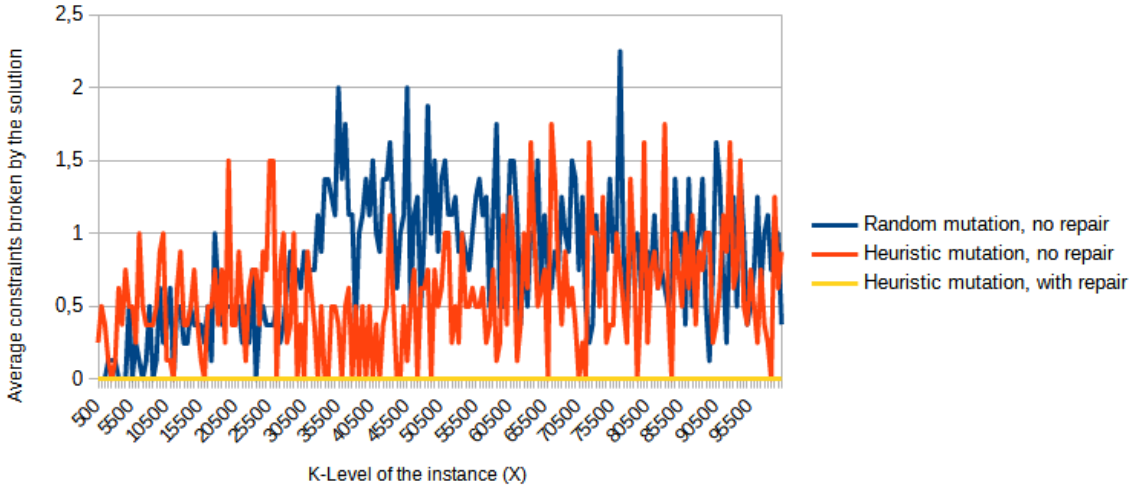
Meanwhile our version with heuristic-guided mutation and without repair keeps having similar constraint violations as before, as can be seen in Figure 5.12b. The version with random mutation and no gene repair results in less constraint violations in this case. This is to be expected, since it is harder to incur in room violations, although it still has more than the one using the heuristic, since the heuristic makes it incur in less group violations by placing entire groups. Nevertheless, it still incurs in violations, with some points on the graph having 1 or more constraint violations in an average of 10 runs.

Because of this, we can observe that our GA with gene repair and heuristic-guided mutation creates solutions that are **viable**.

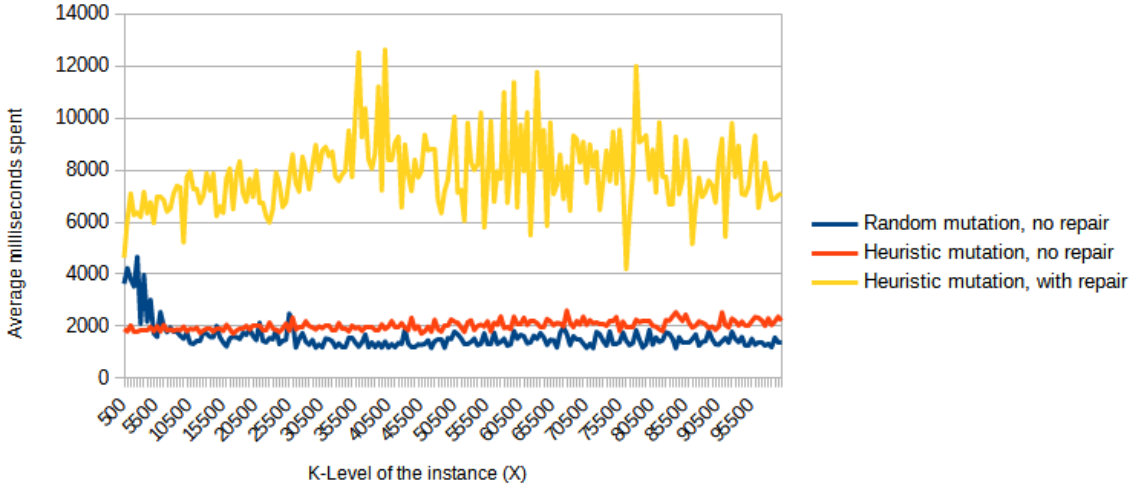
The cost of the use of the gene repair mechanism, as opposed to not using it, can be observed to be similar than in the previous case in Figure 5.12c.



(a) K-Level obtained per GA version for instances of 5 groups and 6 rooms, each of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 6×5. For 5 groups and 6 rooms of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for 5 groups and 6 rooms, each of size X (average of 10 runs per point)

Figure 5.12: Comparison of GA versions for different metrics across instances of 5 groups and 6 rooms, each of size X.

3. **Case 2, restrictive:**
   In this case, we decided to go against the mechanism of the gene repair and in favour of the heuristic-guided mutation, creating the following configuration: Consider 8 groups and 4 rooms. The rooms are of size X, and the sum of every two consecutive groups is X as well, where every first group will be randomly generated in the range of $\frac{X}{2} \leq A_j \leq \frac{3X}{2}$, and the next one will be the remaining amount to reach X (i.e. $B_j = X - A_j$):

$$\begin{aligned} \text{Rooms} &= [X_1, \ldots, X_4] \\ \text{Groups} &= [A_1, B_1, \ldots, A_4, B_4] \\ \text{where: } &\sum_{j=1}^{n} (A_j + B_j) = X \end{aligned}$$

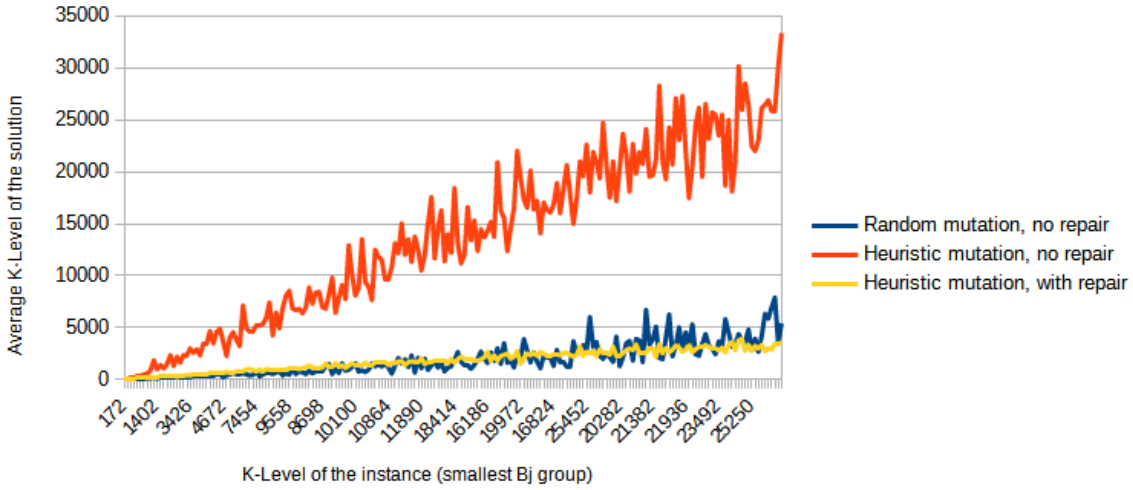
   In this scenario, X grows from 500 to 100,000 in increments of 500. So we expect A_j to grow, randomly, from 375 to 750,000 and B_j from 125 to 25,000. We will plot the values based on the maximum possible $K-$Level of the solution. The highest $K-$Level achievable in this case will be given by the smallest group for that instance (i.e. the smallest B_j). See sub-figures in Figure 5.13.

   The heuristic-guided mutation in our GA will try to place as many people as there are in some group, when mutating a random cell. To reach a correct answer, it will need to do this with two consecutive groups, for each room, and have zeroes in the rest of that column. In the case of gene repair, when repairing a group, we split it depending on the capacities of the room, making it extremely difficult to reach the highest $K-$Level in cases like these, specially when the instance is restrictive.

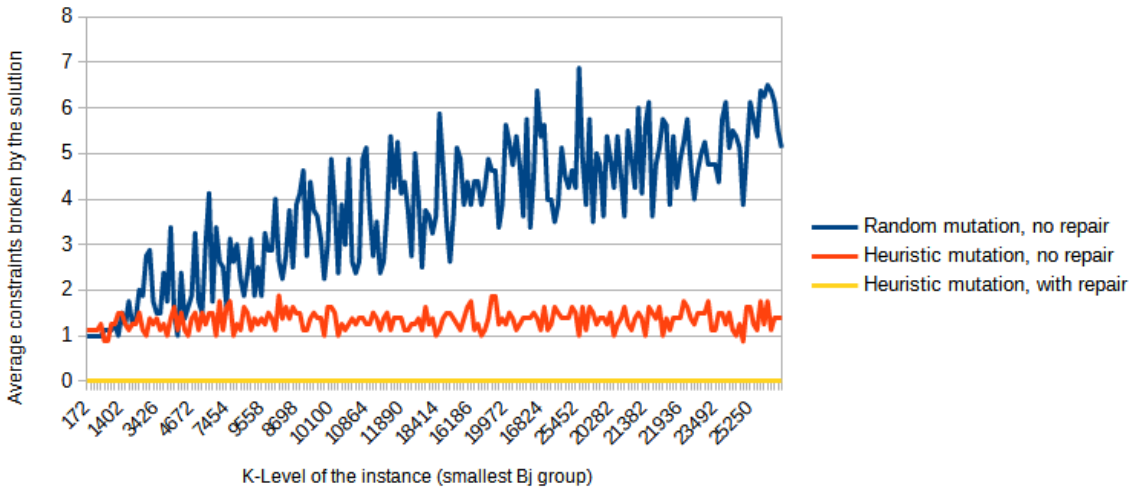
   As we can see in Figure 5.13a, the $K-$Level of our instance will not be reached by our algorithm with repair, since the configurations go against the repair mechanism, making the groups split.

   There we can also observe how the heuristic-guided mutation is an improvement over the random mutation for the $K-$Level, but unfortunately, without repair, we always break some constraint, as we can see in Figure 5.13b. This means that all solutions given by both algorithms without repair are not viable. In that same figure we can see that, as X rises, so do the constraints broken by our version with random mutation and no repair. This is because there is more room for failure, and by randomly mutating, it increasingly mismatches the group sizes in the instance. We can also observe, that even if the $K-$Levels of our version with gene repair are lower, the constraints broken are always 0. Here we can see again how all the solutions returned by this version are viable.

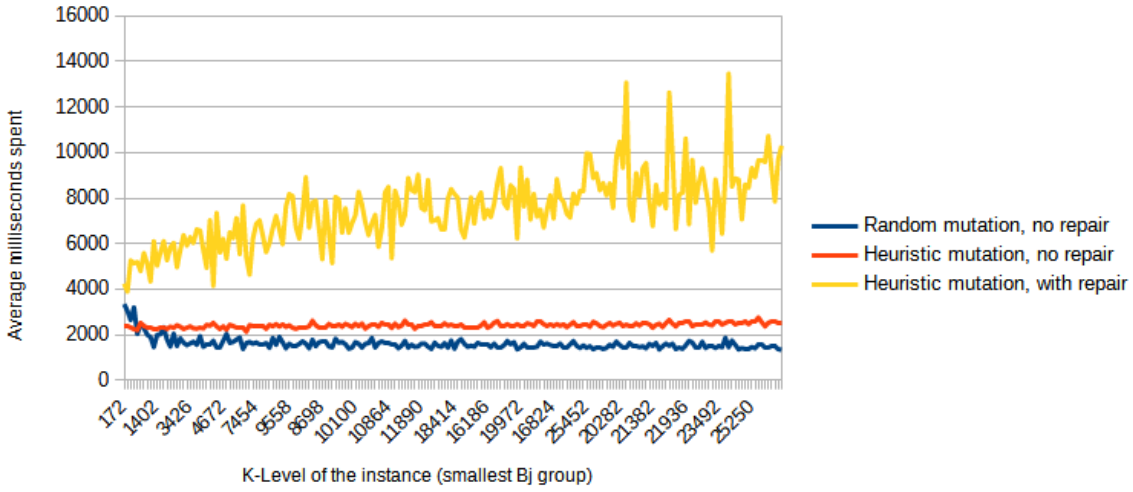
   Even though the computational cost may be slightly higher for the same number of generations, as illustrated in Figure 5.13c, incorporating gene repair ensures that the solutions generated by the GA are viable. This approach is crucial for our needs, and we can afford the extra costs, since we maintain polynomial time complexity relative to the GA's inputs.



(a) *K*-Level obtained per GA version for instances of 8 groups of varying sizes and 4 rooms of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 4×8. For 8 groups of varying sizes and 4 rooms of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for instances of 8 groups of varying sizes and 4 rooms of size X (average of 10 runs per point)

Figure 5.13: Comparison of GA versions for different metrics across instances of 8 groups of varying sizes and 4 rooms of size X.

4. **Case 2, non-restrictive:**

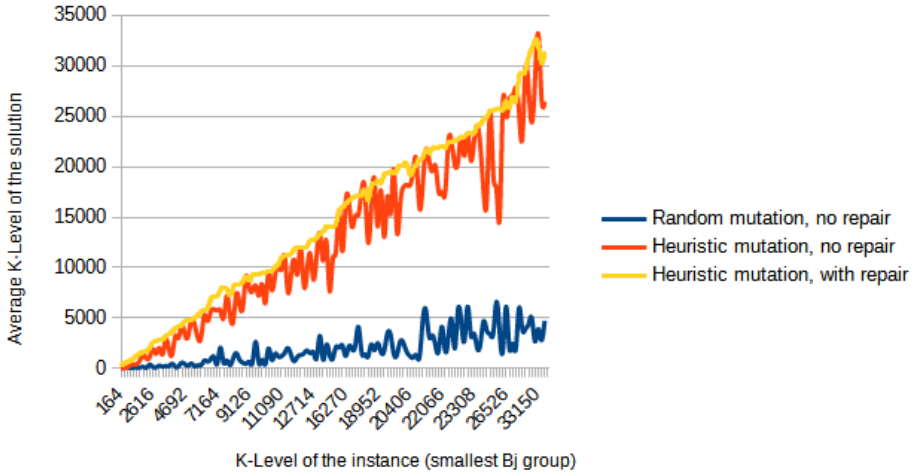
From the restrictive version we create the following one:

$$\begin{aligned} \text{Rooms} &= [X_1, \ldots, X_5] \\ \text{Groups} &= [A_1, B_1, \ldots, A_4, B_4] \\ \text{where: } &\sum_{j=1}^{n} (A_j + B_j) = X \end{aligned}$$

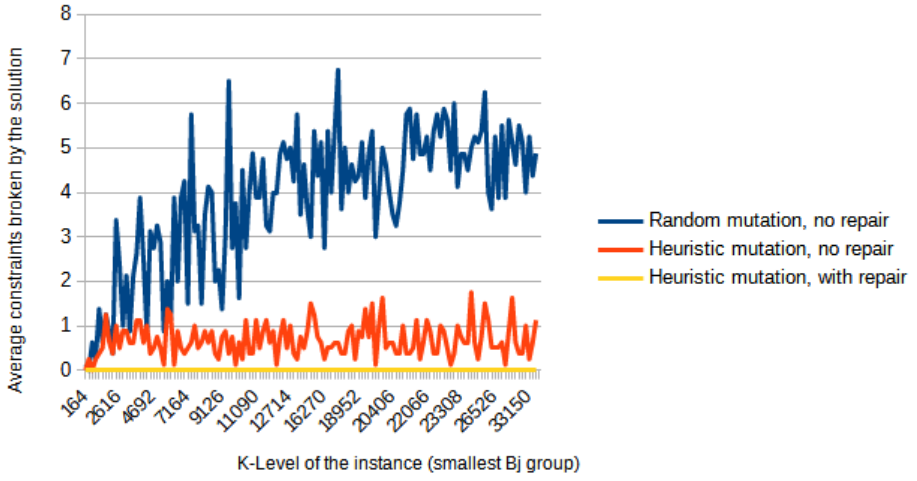
As we can see, now we use 5 rooms instead of 4, and this means we have extra room for X patients. In this scenario, X grows from 500 to 100,000 in increments of 500. So we expect A_j to grow, randomly, from 375 to 750,000 and B_j from 125 to 25,000. We will again plot the values based on the maximum possible $K-$Level of the solution. The highest $K-$Level achievable in this case will be given by the smallest group for that instance (i.e. the smallest B_j). In Figure 5.14 we can see the results obtained by our three versions.

We can observe in Figure 5.14a that the GA with gene repair and heuristic-guided mutation exhibits much higher $K-$Levels than in the restrictive instance. The extra room allows for it to find higher $K-$Levels once the constraint violations are out of the way. There we can observe how the heuristic-guided mutation is an improvement over the random mutation for the $K-$Level, but unfortunately, without repair, we may have some constraint violation in our solution, as we can observe in Figure 5.14b. This figure resembles the one of the restrictive case, with an increasing number of constraint violations for the version with random mutation and no repair. It also reinforces the idea that all the solutions returned by our GA with heuristic-guided mutation and repair are viable.

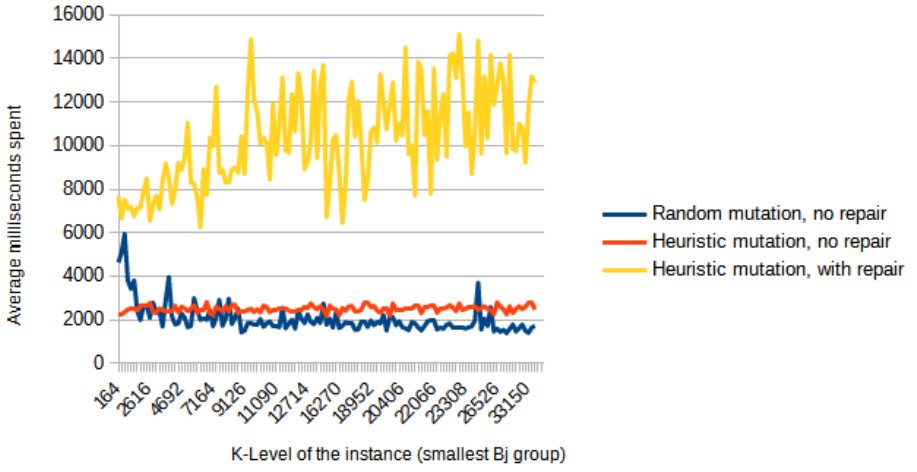
Finally, in 5.14c we can see again how the computational cost for the same number of generations is higher for our GA that uses the gene repair mechanism.



(a) K-Level obtained per GA version for instances of 8 groups of varying sizes and 5 rooms of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 4×8. For 8 groups of varying sizes and 5 rooms of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for instances of 8 groups of varying sizes and 5 rooms of size X (average of 10 runs per point)

Figure 5.14: Comparison of GA versions for different metrics across instances of 8 groups of varying sizes and 5 rooms of size X.

5. **Case 3, restrictive:**
In this case, we consider the opposing configuration to the previous one, one that challenges the heuristic and, instead, favors the repair mechanism within our GA. Specifically, we analyze a scenario involving 4 groups and 8 rooms. Let the size of each group be denoted as X, and the sizes of the rooms be configured such that the sum of the sizes of every two consecutive rooms equals X. The distribution of these room sizes follows a pattern where the first room in each pair is randomly generated within the range of $\frac{X}{2} \leq A_j \leq \frac{3X}{2}$, and the second room in each pair is calculated as the remaining amount needed to reach X, i.e., $B_j = X - A_j$. Formally, this can be expressed as:

$$\begin{aligned} \text{Rooms} &= [A_1, B_1, \ldots, A_8, B_8], \\ \text{Groups} &= [X_1, \ldots, X_8], \\ \text{where } &\sum_{i=1}^{8} (A_j + B_j) = X. \end{aligned}$$

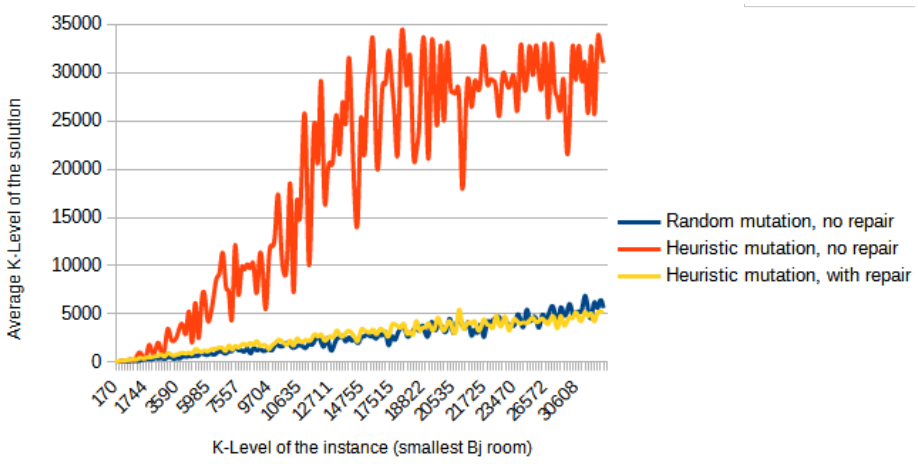
In this scenario, X grows from 500 to 100,000 in increments of 500. So we expect A_j to grow, randomly, from 375 to 750,000 and B_j from 125 to 25,000. We will again plot the values based on the maximum possible $K-$Level of the solution. The highest $K-$Level achievable in this case will be given by the smallest room for that instance (i.e. the smallest B_j). See Figure 5.15.

In this configuration, each group is larger than any single room, making it impossible to place any entire group into a single room without splitting it. This scenario conflicts with the heuristic-guided mutation strategy in our GA, which attempts to place the entire group into one room during mutation. Because of this, the heuristic fails in this situation, creating solutions with over-inflated and unrealistic K-Levels, as illustrated in Figure 5.15a. This exaggerated K-Level reflects the infeasibility of the solution and highlights the limitations of applying this heuristic indiscriminately.

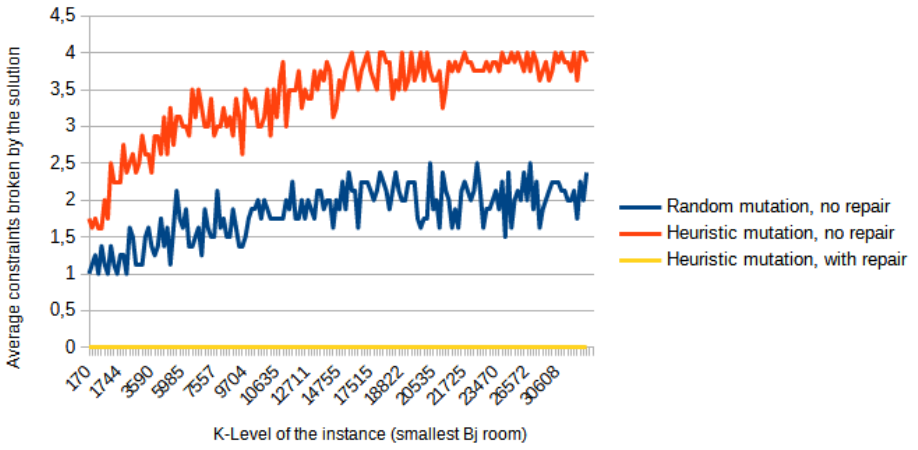
Despite the inadequacy of the heuristic in this case, the gene repair function within the GA proves to be highly effective. During the repair process, the algorithm splits groups across multiple rooms, adhering to the room capacities. Because of the restrictiveness of the instance, the repair mechanism will reduce the K-Level by distributing groups in a manner that fits within the available room capacities. This reduction in K-Level is crucial to ensure viability, but will reflect in a quality drop as the groups get larger.

Interestingly, Figure 5.15b shows that the heuristic-guided mutation is so detrimental to the search in this case that the random version incurs in less constraint violations.

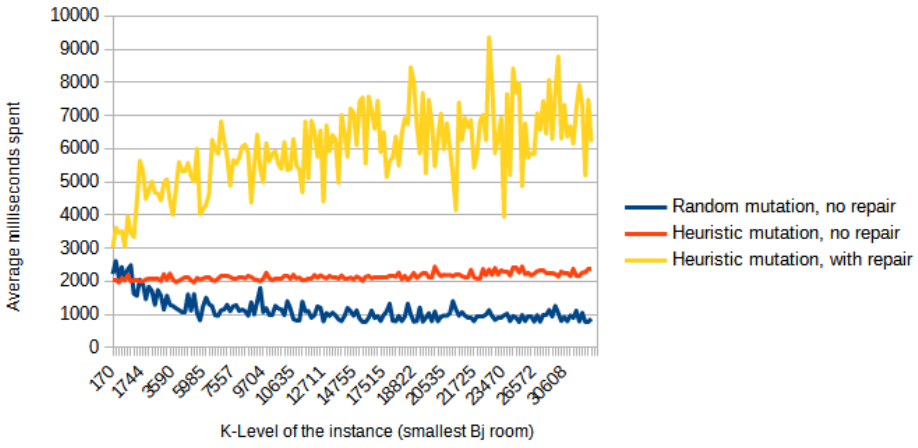
Additionally, Figures 5.15b and 5.15c present information consistent across all three cases considered in our study. By examining these figures, we can observe the impact of the repair mechanism in the computational cost and the resulting reduced constraint violations. The consistency in these results reinforces our conclusion that, for the K-Anonymity problem, our Steady-State GA equipped with heuristic-guided mutation and a our gene-repair mechanism is the most effective approach. This combination allows us to achieve solutions that may offer low K-Levels for restrictive instances, but are always viable.



(a) K-Level obtained per GA version for instances of 8 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 4×8. For 8 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for instances of 8 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)

Figure 5.15: Comparison of GA versions for different metrics across instances of 8 rooms of varying sizes and 4 groups of size X.

6. **Case 3, non-restrictive:**
   We construct the non-restrictive instances in a similar way. Let the size of each group be denoted as X, and the sizes of the rooms be configured such that the sum of the sizes of every two consecutive rooms equals X. The distribution of these room sizes follows a pattern where the first room in each pair is randomly generated within the range of $\frac{X}{2} \leq A_j \leq \frac{3X}{2}$, and the second room in each pair is calculated as the remaining amount needed to reach X, i.e., $B_j = X - A_j$. Additionally, we include two extra rooms of size $\frac{X}{2}$.

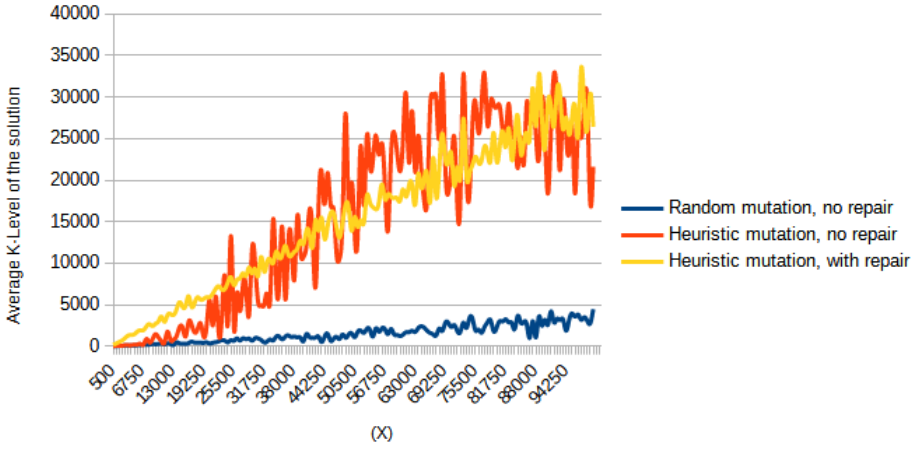
$$\begin{aligned} \text{Rooms} &= [A_1, B_1, \ldots, A_8, B_8, \frac{X}{2}, \frac{X}{2}], \\ \text{Groups} &= [X_1, \ldots, X_8], \\ \text{where } &\sum_{i=1}^{8} (A_j + B_j) = X. \end{aligned}$$

In this scenario, X grows from 500 to 100,000 in increments of 500. So we expect A_j to grow, randomly, from 375 to 750,000 and B_j from 125 to 25,000. The highest $K-$Level achievable in this case will depend on the random generation of the rooms, since the two rooms with capacity $\frac{X}{2}$ allow for more possibilities. Because of this, we will plot the instances by the X they were generated with. See Figure 5.16.

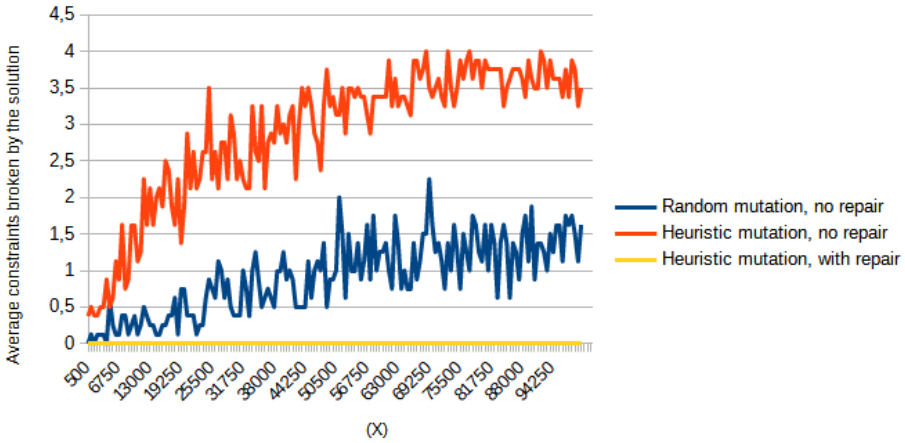
In this scenario the heuristic still fails, even if it is non-restrictive. Our GA without repair that uses the heuristic is creating solutions with over-inflated and unrealistic K-Levels, as illustrated in Figure 5.15a. Despite the inadequacy of the heuristic in this case, the gene repair function within the GA proves to be highly effective. Due to the instances being non-restrictive, the K-Level of the solutions given by the GA employing gene repair are far better. With this last case we can conclude that the GA with repair is a lot better in non-restrictive instances than in restrictive ones.

Even in a non-restrictive case Figure 5.15b shows that the heuristic-guided mutation is so detrimental to the search in these scenarios that the random version incurs in less constraint violations. Actually, only for extremely small instances we may achieve a result without constraint violations from the versions without repair, even if the instances generated are non-restrictive.

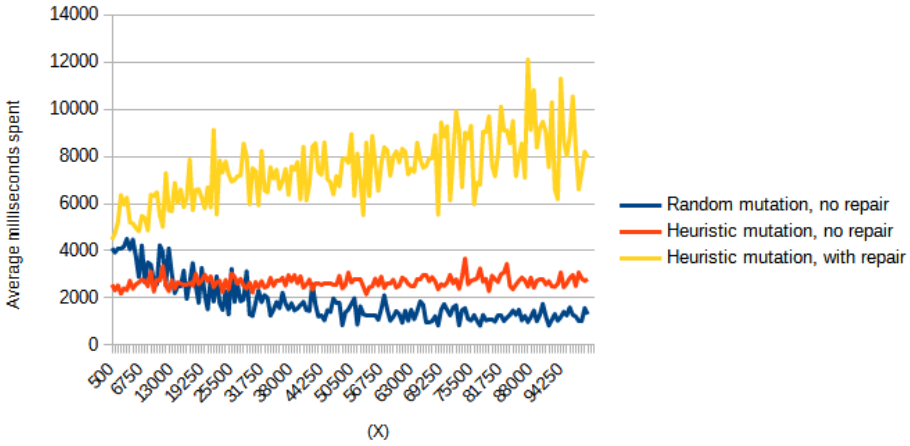
Finally, Figures 5.15b and 5.15c present consistent information. We can observe the impact of the repair mechanism in the computational cost and the resulting constraint violations and compare the results against the GAs without repair.



(a) K-Level obtained per GA version for instances of 10 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)



(b) Room and group violations per GA version for a solution matrix of 4×10. For 10 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)



(c) Milliseconds taken per GA version for instances of 10 rooms of varying sizes and 4 groups of size X (average of 10 runs per point)

Figure 5.16: Comparison of GA versions for different metrics across instances of 10 rooms of varying sizes and 4 groups of size X.

5.4.3. Conclusions

A restrictive problem instance is one where the total capacity of the rooms exactly matches the number of patients. In such situations, GA faces challenges because it has to fit patients into rooms without exceeding capacity, which limits its effectiveness.

The GA performs better when the room capacities are greater than the number of patients (i.e. the instance is non-restrictive). This is because the fitness function of the GA focuses on meeting constraints, and the gene repair mechanism helps by redistributing patients into available rooms. In restrictive cases, the algorithm tends to get simply stuck. This happens because when a mutation causes a constraint to be broken, the repair mechanism often just reassigns patients in a similar way as before, since this usually still satisfies the room constraints. As a result, the algorithm struggles to find better solutions. In fact, using gene repair in restrictive cases, the evolution to a better solution stales out after every member of the population has zero broken constraints.

For less restrictive instances, the algorithm works much more effectively, since it can still keep solutions that shuffle assignments while keeping zero constraints.

To address this issue in restrictive instances, it would be helpful to use a numerical fitness function that balances between achieving a certain quality level (K-Level) and minimizing constraint violations. This way, the algorithm can improve its solutions while also reducing violations, although this could result in solutions with constraint violations, since it would no longer be the top priority for the fitness function.

Additionally, the algorithm performs best when groups of patients can be placed entirely into individual rooms. This is because the heuristic method used tries to put whole groups into rooms. On the other hand, the algorithm performs poorly if all the rooms are smaller than any of the groups, leading to frequent violations of constraints and mutations that result in individuals that will not be stronger than their parents.

Because of this, the worst scenario for our GA is a restrictive instance where every group is bigger than any room. Meanwhile, the best scenario for our GA is a non-restrictive instance where no group is bigger than any room.

Of course, out of our results of the three versions that we measured off against each other, empirically the version of the GA with heuristic-guided mutation and gene repair is the most effective. This is because, while having lower $K-$Levels in restrictive instances, it always avoids constraint violations, which is crucial in our solutions because it makes them viable.

CHAPTER 6

Solving By Constraint Programming

Our $K-$Anonymity problem, as defined per $K-$Anonimity problem versions in section 3.1.2, is an optimization problem, where the goal is to ensure achieving the maximum $K-$level of all possible solutions while respecting some constraints. In this section *constraint programming* is going to be used to create solutions that maximize the $K-$Level while respecting these constraints.

6.1. INTRODUCTION TO MINIZINC

Constraint programming is a powerful paradigm used to solve complex problems by defining and managing constraints on decision variables defined over some domain.

To solve constraint programming problems, specialized software known as *solvers* are used. Solvers are designed to efficiently explore the solution space and apply algorithms to satisfy constraints and optimize the objective function. One widely used tool is MiniZinc, which provides a high-level modeling language for expressing constraints and optimization problems. MiniZinc interfaces with various underlying solvers to find solutions to these problems.

We hereby introduce the key components and concepts involved in constraint programming, and their representation in Minizinc syntax in the context of our problem.

6.1.1. Domains

In constraint programming, each variable is associated with a domain, which is the set of possible values that the variable can take. Defining the domain of a variable is crucial as it impacts the complexity of the problem and the efficiency of the solving process. Domains can be finite, such as integers within a specific range, or more complex, such as sets of values satisfying certain properties.

In our K-Anonymity problem, we are given an array of groups, where each group consists of indistinguishable individuals but distinct from individuals in other groups, as well as an array representing the capacities of the rooms. We represent the solution as a 2D matrix, where each entry indicates the number of individuals from each group assigned to each room. Let's say the problem statement has 3 rooms and 3 groups:

3 Groups and 3 Rooms Representation

$$\text{Groups}: \ [g1,\ g2,\ g3]$$

$$\text{Rooms}: \ [r1,\ r2,\ r3]$$

Matrix Representation

$$\begin{bmatrix} s_{11} & s_{12} & s_{13} \\ s_{21} & s_{22} & s_{23} \\ s_{31} & s_{32} & s_{33} \end{bmatrix}$$

3 Groups and 3 Rooms Example

$$\text{Groups}: \ [3,\ 4,\ 3]$$

$$\text{Rooms}: \ [6,\ 4,\ 2]$$

Solution Example

$$\begin{bmatrix} 3 & 0 & 0 \\ 0 & 4 & 0 \\ 3 & 0 & 0 \end{bmatrix}$$

In the Solution matrix, s_{ij} represents the number of people from group i screened in room j. As we can see, it is crucial that size of our solution's 2D matrix is of size Rooms × Groups, but also we can notice that there cannot be any s_{ij} such that $s_{ij} > g_i$. These notions can bound our solution matrix and reduce the exploration space and also the search time. This can be explicitly stated in Minizinc with:

```
array[1..kGroups, 1..rooms] of var 0..max(kGroupVector): groupRoomCounts;
```

This defines our 2D solution matrix as groupRoomCounts. Here groupRoomCounts ensures the correct number of rows and columns, while also establishing the domain of the numbers inside, which will go from 0 to max (Groups).

6.1.2. Constraints

Constraints are conditions or boundaries that any solution to a problem must satisfy. They define the relationships between variables and limit the possible values that these variables can take. Constraints ensure that solutions are not only optimal but also feasible within real-world limitations. They play a critical role in narrowing down the solution space and guiding the search for feasible solutions.

In the case of our $K-$Anonymity problem, a possible constraint could be that the number of patients of one group in our solution is exactly the same number in that group in our problem statement. This is crucial for the solution to be accurate to the number of people we want to screen, but it is also diminishing the search space by eliminating all possible representations of the solution where a group of screened people is unbounded by the original number given. This makes the search domain smaller and the search time, thus, quicker. This constraint can be written in Minizinc as:

```
% Ensure groupRoomCounts respects group sizes
constraint forall(g in 1..kGroups)(
  sum(r in 1..rooms)(groupRoomCounts[g, r]) == kGroupVector[g]
);
```

Where kGroupVector includes the statement information of the $K-$groups. Something similar can be done to respect the size of every room:

```
% Ensure groupRoomCounts respects room capacities
constraint forall(r in 1..rooms)(
  sum(g in 1..kGroups)(groupRoomCounts[g, r]) <= roomCapacity[r]
);
```

6.1.3. Optimization

In optimization problems, the objective is not only to satisfy all constraints but also to find a solution that optimizes a particular objective function. The objective function could be to maximize or minimize a certain value, such as cost, time, or resource usage. Optimization adds another layer of complexity, as it requires balancing between satisfying constraints and achieving the best possible outcome according to the objective function.

In our case we want to maximize the $K-$Level, which can be written, satisfyingly simply, in Minizinc as:

```
solve maximize kLevel;
```

6.1.4. Solvers

MiniZinc is a high-level constraint modeling language that allows you to easily express and solve discrete optimization problems. It provides a flexible and user-friendly syntax for defining variables, constraints, and objective functions. These representations follow the Minizinc syntax defined in the: MiniZinc Handbook. After this representation is created, a *Solver* is used to find solutions. One of them is Gecode:

Gecode 6.3.0

Gecode is a versatile and efficient constraint programming library that provides a set of powerful tools for solving constraint satisfaction and optimization problems. Version 6.3.0 of Gecode is a stable and efficient release.

Gecode can be used as a backend solver interfaced by MiniZinc, enabling users to model problems using MiniZinc's high-level language while leveraging Gecode's powerful solving capabilities. This integration combines the ease of modeling with the efficiency of a state-of-the-art solver.

6.2. MINIZINC SOLUTION TO THE K-ANONYMITY PROBLEM

MiniZinc's main file to solve problems is the Minizinc model. The following file shows the model created to maximize the $K-$Level in Minizinc:

```
include "globals.mzn";

% Data
int: kGroups;
constraint assert(kGroups > 0, "Patients are grouped");

array[1..kGroups] of int: kGroupVector;
constraint assert(forall(i in 1..kGroups)(kGroupVector[i] > 0), "Groups are positive");

int: rooms;
constraint assert(rooms > 0, "We have screening rooms");

array[1..rooms] of int: roomCapacity;
constraint assert(forall(i in 1..rooms)(roomCapacity[i] > 0), "Room Capacity is not negative");
constraint assert(sum(kGroupVector) <= sum(roomCapacity), "We can allocate all patients in our rooms");

% Decision variables
var 1..min(kGroupVector): kLevel;
array[1..kGroups, 1..rooms] of var 0..max(kGroupVector): groupRoomCounts;

% Constraints

% Ensure groupRoomCounts respects group sizes
constraint forall(g in 1..kGroups)(
  sum(r in 1..rooms)(groupRoomCounts[g, r]) == kGroupVector[g]
);

% Ensure groupRoomCounts respects room capacities
```

```
constraint forall(r in 1..rooms)(
  sum(g in 1..kGroups)(groupRoomCounts[g, r]) <= roomCapacity[r]
);

% Calculate the K-Level of the solution as:
% non-zero min(groupRoomCounts[g,r])
% for all g in groups and r in rooms
constraint kLevel == min([groupRoomCounts[g, r] | g in 1..kGroups, ↩
    ↪ r in 1..rooms where groupRoomCounts[g, r] > 0]);

% Objective function to maximize the K-Level of the solution
solve maximize kLevel;

% Output the results
output [
    "K-Level␣of␣solution:␣\(kLevel)\n",
] ++
[ "K-Groups:␣\(kGroupVector)\n"
]++
[ "Room␣Capacities:␣\(roomCapacity)␣\n"
]++
["Group␣Room␣Counts:\n"
]++
[
  "Room␣\(r):\t" ++
  concat([show(groupRoomCounts[g, r]) ++ "\t" | g in 1..kGroups]) ++
  "\n"
  | r in 1..rooms
];
```

Our Minizinc model:

- Uses assertions to ensure that the data provided is in the correct format.
- Creates the 2D matrix representation of the solution.
- Bounds the number of people from each group assigned to each room to be exactly equal to the number of people in that group.
- Bounds the number of people assigned to each room to be less or equal to the capacity of the room.
- Defines the K-Level as the smallest, non-zero group assigned to any room.
- Maximizes the K-Level of the solution.

This is the full implementation of our Minizinc mode. It is extremely lightweight thanks to the constraint programming paradigm. If the file is solicited, it can be found in [3].

6.3. RESULTS OF MINIZINC WITH GECODE

Using the configuration listed in Annex A, let us reintroduce the cases we used to analyze our BnB Algorithm in Section 4.3:

1. **Case 1:**
   Consider 5 groups and 5 rooms, each with size X. The room configuration is as follows:

$$\text{Rooms} = \text{Groups} = [X, X, X, X, X]$$

In this scenario, X grows from 1 to 10,000 in increments of 5. In Figure 6.1 we can see the results obtained.

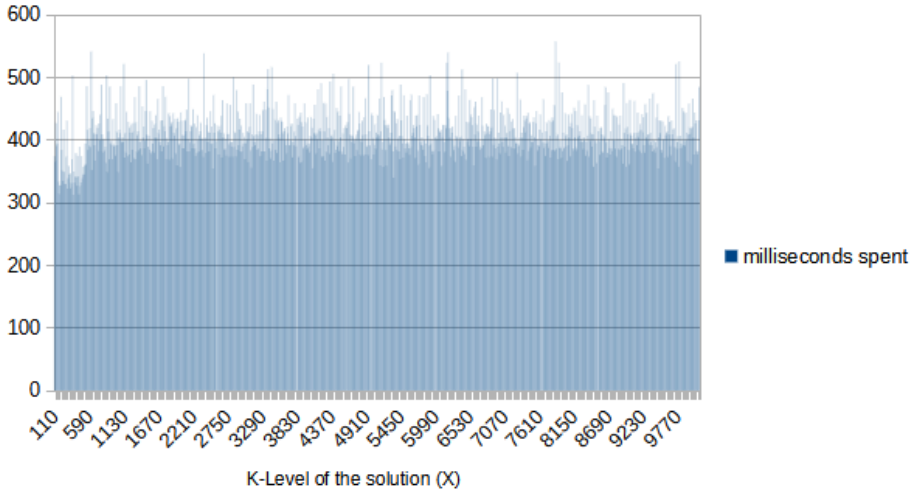


Figure 6.1: Minizinc with Gecode model performance for 5 groups and 5 rooms, each of size X.

2. **Case 2:**
Consider 5 groups and 10 rooms, with the following sizes:

$$\text{Rooms} = \left[\frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}\right]$$

$$\text{Groups} = [X, X, X, X, X]$$

Here, X grows from 2 to 10,000 in increments of 5. As we can see in Figure 6.2, the computation time remains constant.

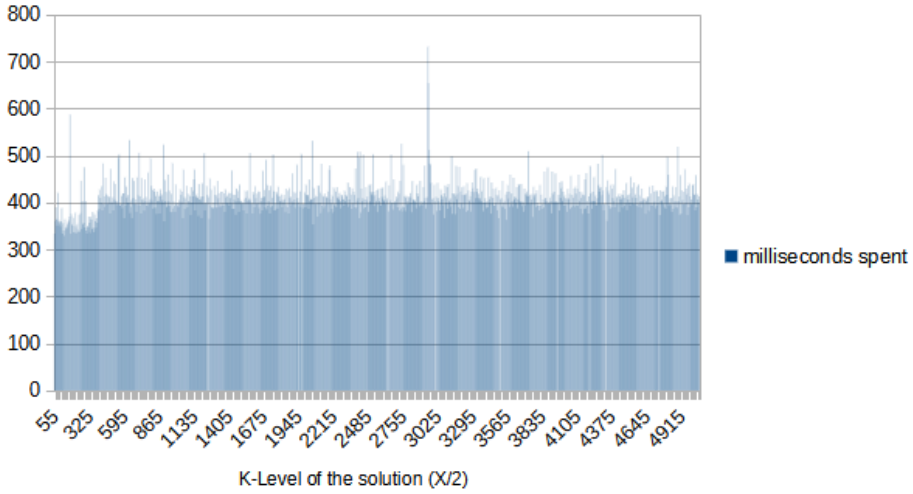


Figure 6.2: Minizinc with Gecode model performance for 5 groups of size X and 10 rooms, with capacities $\frac{X}{2}$.

3. **Case 3:**
Finally, consider 10 groups and 5 rooms, with the room sizes configured as:

$$\text{Rooms} = [X, X, X, X, X]$$

$$\text{Groups} = \left[\frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}, \frac{X}{2}\right]$$

In this case, X grows from 2 to 10,000 in increments of 5. Similarly to the previous cases, as the BnB algorithm reaches the upper bound in the first depth-first traversal, our Minizinc model solves the instances in constant time, as can be seen in Figure 6.3.

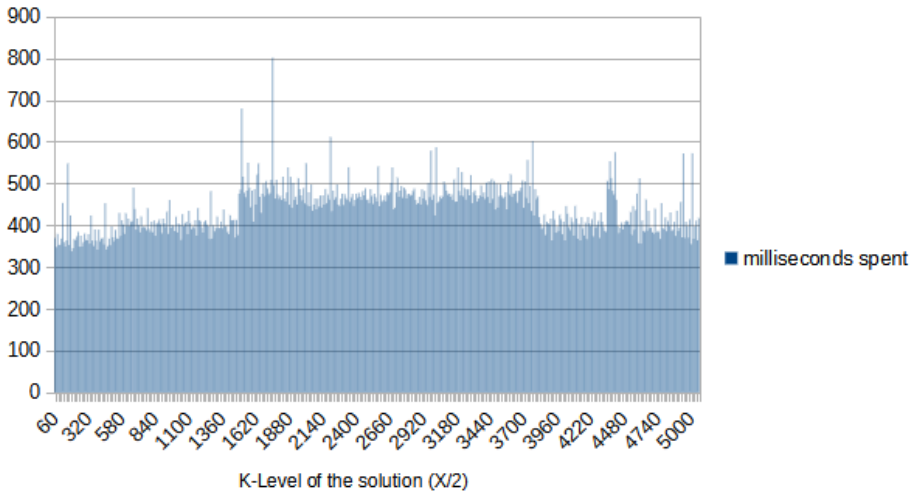


Figure 6.3: Minizinc with Gecode model performance for 10 groups of size $\frac{X}{2}$ and 5 rooms, each of size X.

These examples illustrate how, quite like our BnB Algorithm 4.1, the Gecode solver handles large inputs efficiently by using multiple optimization techniques, making the computation time relatively constant, between 300 and 600 ms.

4. **Case 4:**
To explore a case that may lead to necessary exhaustive exploration, as with BnB, we use the following configuration:

$$\text{Rooms} = [X, X, X, X, 2X]$$

$$\text{Groups} = [X + 2, 2X, 3X - 2]$$

To reach the highest K-Level, we must avoid splitting the smallest group. The distribution yielding the highest K-Level is the following (as we saw before in sub-section 4.3.2):

$$\text{Assignments} = \begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ X & X & 0 & 0 & 0 \\ 0 & 0 & X & X & X-2 \end{bmatrix}$$

Where the X values of groups 2 and 3 can be rearranged freely. The plot results of this configuration, with X ranging from 3, in increments of 10, to 10.003, can be seen in Figure 6.4.

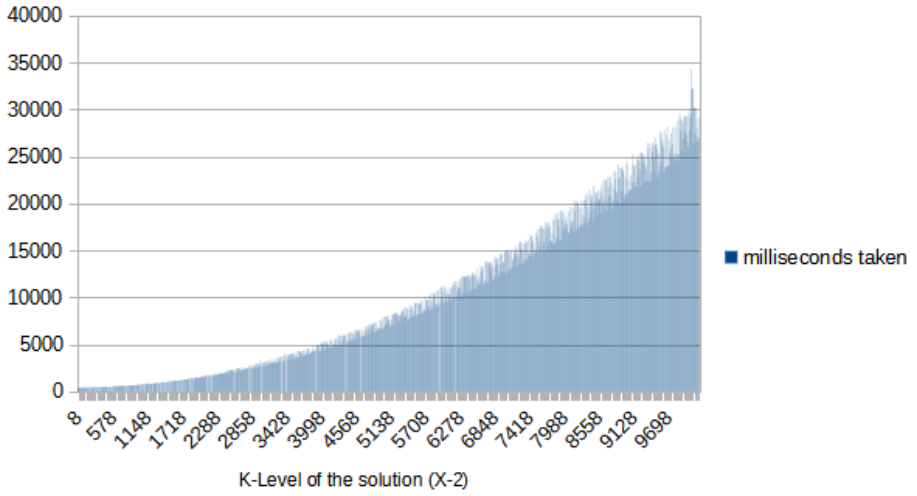


Figure 6.4: Minizinc with Gecode model performance for 3 groups and 5 rooms, with varying sizes.

In this case, the cost arises and we can see the exponential growth of the milliseconds taken. However we can see that the optimization techniques of constraint programming are beating the configuration of our Branch and Bound algorithm for this case by a truly significant margin. BnB takes close to 12,000 seconds (roughly 8 hours) to solve the instance where $X = 45$, as can be seen in Figure 4.5 of the results of our BnB Algorithm, whereas our Minizinc model with the Gecode solver takes around 275 seconds to solve an instance where $X = 30{,}000$.

We can truly see the benefit of constraint programming here, since the design of BnB in C++, which takes up around 800 lines, involved solving the subproblems by which we branch, and decide the backtracking, exploration and pruning decisions, as well as debugging and making sure the behavior is correct. In constraint programming our Minizinc model is 56 lines long and the design involves stating the constraints of our solution and the objective function. With this lightweight solution, it is still faster than our BnB, thanks to the optimizations already designed in the Gecode solver.

CHAPTER 7

Results of the three algorithms

In this chapter we will measure off the three algorithms in order to see how well they perform against each other. This will be done over varied instances, in order to explore the weaknesses and strengths of our algorithms.

7.1. CASES TO MEASURE

To pit them against each other we will use a restrictive and a non-restrictive case. Let us recall their definition:

a) **Restrictive instances**: In these instances $\sum \text{Groups} = \sum \text{Rooms}$.

b) **Non-restrictive instances**: In these instances $\sum \text{Groups} < \sum \text{Rooms}$.

We will generate two different cases, each with a restrictive and a non-restrictive version.

1. **Case 1, restrictive:**

   We will use the growing restrictive case where the upper bound is not reached by both BnB and Minizinc, i.e:

   $$\text{Rooms} = [X, X, X, X, 2X]$$

   $$\text{Groups} = [X + 2, 2X, 3X - 2]$$

   The global optima are the solutions that assign the first group (composed of $X + 2$ individuals) and $X - 2$ members of the third group to the third room. Since the remaining X are indivisible in the first 4 rooms, the alternative value is 0. Since we need to place two X for the two last groups in 4 rooms, in total there are $\binom{4}{2} = 6$ global optima. These global optima are illustrated in Figure 7.1.

Solution 1

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ X & X & 0 & 0 & 0 \\ 0 & 0 & X & X & X-2 \end{bmatrix}$$

Solution 2

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ X & 0 & X & 0 & 0 \\ 0 & X & 0 & X & X-2 \end{bmatrix}$$

Solution 3

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ 0 & X & 0 & X & 0 \\ X & 0 & X & 0 & X-2 \end{bmatrix}$$

Solution 4

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ 0 & 0 & X & X & 0 \\ X & X & 0 & 0 & X-2 \end{bmatrix}$$

Solution 5

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ 0 & X & X & 0 & 0 \\ X & 0 & 0 & X & X-2 \end{bmatrix}$$

Solution 6

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 \\ X & 0 & 0 & X & 0 \\ 0 & X & X & 0 & X-2 \end{bmatrix}$$

Figure 7.1: All six global optima for the restrictive instance.

2. **Case 1, non-restrictive:**
We will create a growing non-restrictive case from our aforementioned restrictive one. This way we will again ensure the upper bound is not reached by both BnB and Minizinc, i.e:

$$\text{Rooms} = [X, X, X, X, 2X, X]$$

$$\text{Groups} = [X+2, 2X, 3X-2]$$

In this case now we have $\sum \text{Groups} + X = \sum \text{Rooms}$. This means that we have one extra room that can fit X patients. This gives better room for improvement for our GA, while also changing the global optima of our problem. Now the highest $K-$Level attainable for the generated instances is $X-1$. This is a global optimum that achieves it:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & X+2 & 0 \\ 0 & 0 & 0 & X & 0 & X \\ X-1 & X-1 & X & 0 & 0 & 0 \end{bmatrix}$$

From this solution, we quickly understand that group 1 will always go, entirely, to room 4, since it is the only one where it can fit without splitting. After that, we split the third group into 3 subgroups of X-1, X-1 and X individuals. Then, we split the second group into 2 subgroups of X individuals each. These splits are the best possible, since any further splitting of the groups would result in a $K-$Level smaller than $X-1$. We will refer to a split of individuals belonging to a group as clusters.

We have 5 positions where we can arrange 5 clusters. Out of these 5 clusters:

a) 2 clusters belong to group 3 and are of size $X-1$.

b) 1 clusters belongs to group 3 and is of size X.

c) 2 clusters belong to group 2 and are of size X.

First we choose 2 positions for the first 2 clusters. The number of ways to do this is given by the combination:

$$\binom{5}{2} = \frac{5!}{2!(5-2)!} = 10$$

Next, we choose where to place the other cluster that belongs to group 3 in the remaining 3 positions. Trivially this can be done in 3 ways. Finally we have 2 positions for the last 2 clusters. This can only be done in one way, since we do not care which one we place first. We get:

$$\binom{5}{2} \cdot 3 = \frac{5!}{2!(5-2)!} \cdot 3 = 30$$

In total we have 30 global optima for every instance generated in this case, with a $K-$Level of $X-1$. Although every room must be used, it is still non-restrictive, since the room capacities exceed the number of patients.

Having 30 distinct global optima in total is not of much difference to our exhaustive algorithms, which will need to prune more branches than before, given that there are more possibilities of assignations. On the contrary, to our GA, as we have already seen, the non-restrictiveness of the instances generated makes a big difference in the performance it can achieve.

This case, as we have already seen, goes against the heuristic, since most groups cannot fit entirely in any given room and should be split, and all global optima need to have an exact split of $(X-1)+(X-1)+X = 3X-2$ patients from the third group arranged in different rooms.

Using cases where the heuristic works (i.e. where all groups can go entirely to some room) would mean that our BnB algorithm would hit the termination criteria (calculated by the Algorithm 4.4) quite quickly, as well as our Minizinc model would use the optimizations of the solver to finish in constant time. These solutions would be faster than the GA, so measuring them would be pointless. The best we can do for our heuristic-guided mutation is craft a case where *some* of the groups can be entirely assigned to some room.

3. **Case 2, restrictive:**

   Consider the restrictive case:

   $$\text{Rooms} = [X, 2X, 2X, X]$$

   $$\text{Groups} = [2X, 2X, 2X]$$

   A global optimum can be:

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

   The number of global optima is 33 in total, for an explanation on this number refer to Annex E. In this case, the termination condition for the BnB algorithm given by Algorithm 4.4 will be 2X, whereas the best solution can only reach X. This means that BnB will be forced to prune all non-promising branches after finding a solution with a $K-$Level of X.

4. **Case 2, non-restrictive:**

   Let the non-restrictive version of case 2 be:

   $$\text{Rooms} = [X, 2X, 2X, X, X]$$

$$\text{Groups} = [2X, 2X, 2X]$$

A global optimum can be:

$$\begin{bmatrix} X & 0 & 0 & 0 & X \\ 0 & 2X & 0 & 0 & 0 \\ 0 & 0 & X & X & 0 \end{bmatrix}$$

The number of global optima is now 207 in total, for an explanation on this number refer as well to Annex E.

In this case, not only the termination condition of the BnB algorithm will be set at 2X, but also the GA will have an extra room with size X to find a global optimum. This means that our BnB algorithm will need to terminate its execution by pruning all possible non-promising branches until it runs out of branches, and also the GA will have extra room to maneuver and fish for higher $K-$Levels.

7.2. RESULTS

For the results, we will choose to time out any execution of any algorithm that exceeds 5 minutes. We will also measure only the GA with heuristic-guided mutation and gene repair. The results will be averaged over 10 runs to account for randomness. These results are achieved in the environment described in Annex A.

The data points for all cases in this chapter are given by an X that grows from 10 to 10,000 in increments of 10.

Case 1, Restrictive

As we can observe in Figure 7.2, BnB times out by the instance where X=30. This exponential growth is the same as the one depicted in the results section of Chapter 4. Because it overshadows the results of the other algorithms, in Figure 7.3 we can see the milliseconds per instance of the GA and the Minizinc model. We can observe in these results how the growth of the Minizinc time spent in computation, although exponential, it never exceeds 35000 milliseconds. We can see it is incredibly fast in comparison.

Finally, in Figure 7.4 we can see the average performance of the GA in terms of quality. We can observe how no constraints are broken, but the $K-$Levels of the solution are really low. They are, on average, $\frac{1}{3}$ of the maximum achievable $K-$Level. It is so low due to the configuration of the instance and the To be able to see this clearly, the maximum achievable $K-$Level is also plotted. The results are consistent with our conclusions of the results section in 5 in the sense that we can see very poor results in restrictive instances.

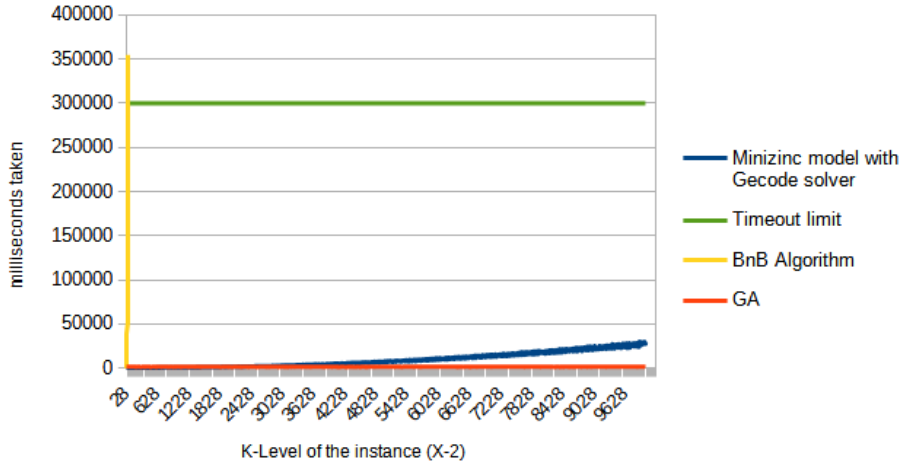


Figure 7.2: Results for Case 1, Restrictive: Milliseconds of the algorithms

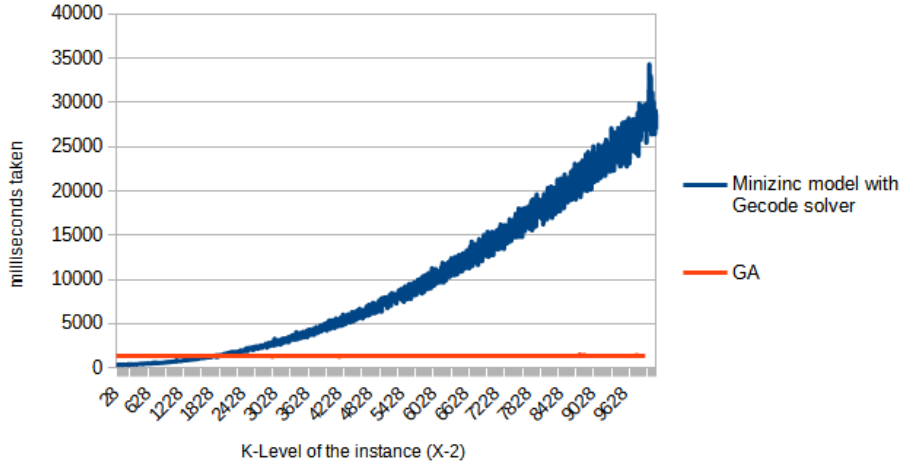


Figure 7.3: Results for Case 1, Restrictive: Milliseconds of the GA and the Minizinc model

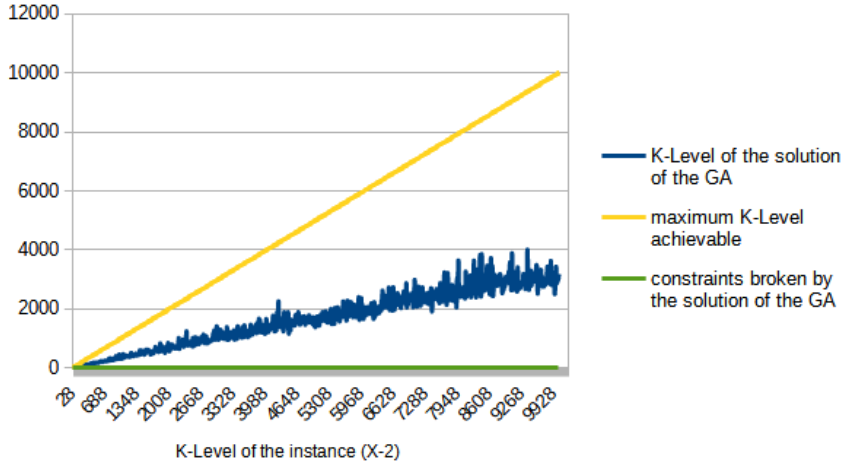


Figure 7.4: Results for Case 1, Restrictive: Average GA Quality of Solutions

Case 1, Non-Restrictive

As we can observe in Figure 7.5, BnB does not time out. This is because the non-restrictive version has a different growth factor in the amount of nodes to explore in the DFS in order to find the right

solution and also stop the exploration. Still, it overshadows the results of the other algorithms, in Figure 7.6 we can see the milliseconds per instance of the GA and the Minizinc model. We conjecture in these results that the growth of the Minizinc time spent in computation is very likely exponential, since it grows and very likely follows the trend established in the non-restrictive case, where it describes a similar shape as our BnB algorithm but is much faster. In fact, it is so fast that it is faster than our GA.

Finally, in Figure 7.7 we can see the average performance of the GA in terms of quality. We again observe how no constraints are broken, except this time around the quality of every solution is much better. These results are also consistent with our conclusions of the results section in 5, since results drastically improve in non-restrictive instances.

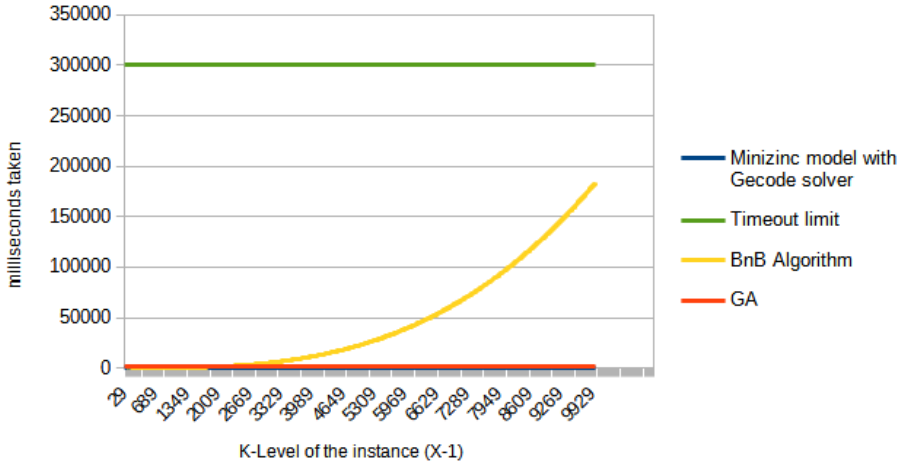


Figure 7.5: Results for Case 1, Non-Restrictive: Milliseconds of the algorithms

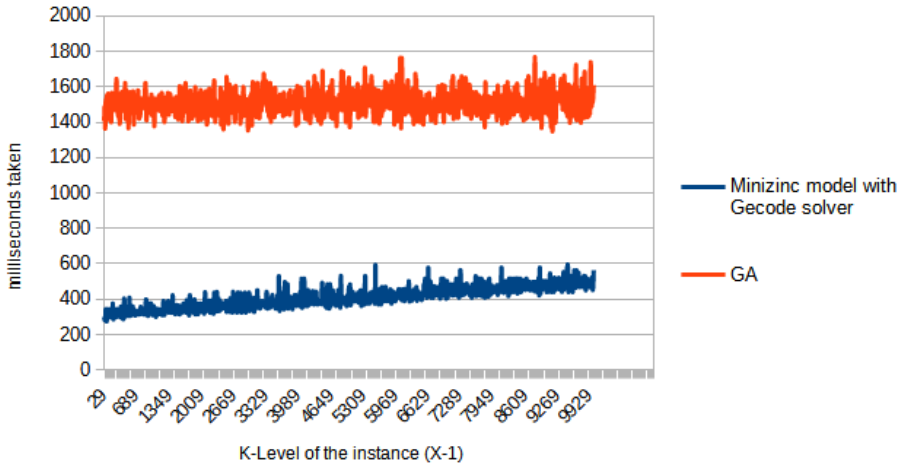


Figure 7.6: Results for Case 1, Non-Restrictive: Milliseconds of the GA and the Minizinc model

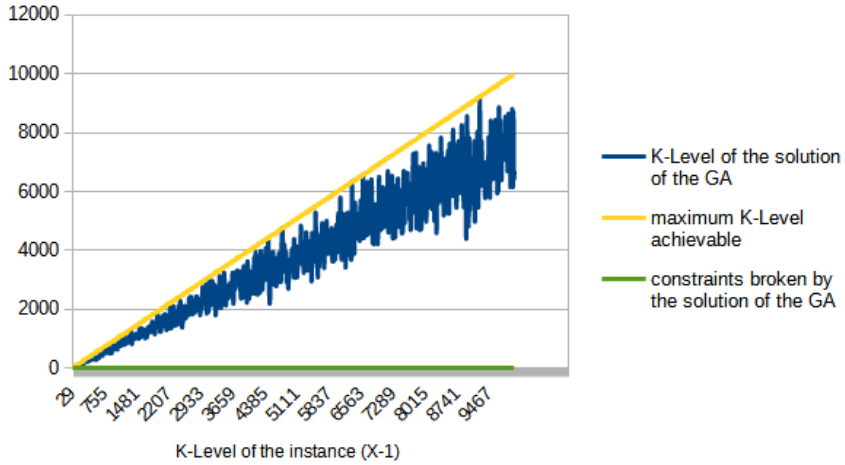


Figure 7.7: Results for Case 1, Non-Restrictive: Average GA Quality of Solutions

Case 2, Restrictive

As we can observe in Figure 7.8, BnB is suddenly the fastest. In Figure 7.9 we can see the reason why. BnB is reaching the optimal solution only traversing 8 nodes in the solution tree. This is because of the mechanisms of DFS and pruning in the design of the BnB algorithm. We can also observe how the time spent by the Minizinc model is constant and low. The GA in this case is the slowest.

Finally, in Figure 7.10 we can see the average performance of the GA in terms of quality. We again observe how no constraints are broken, and the results are poor since we deal with a restrictive instance again. We can observe that even though the $K-$Levels of the solutions are poor, they are still better than the ones of the first case (see Figure 7.10). This is because the heuristic-guided mutation helps with the assignations of groups of size $2X$ to rooms of size $2X$, since the heuristic tries to place an entire group in some room. The results are around half of the maximum achievable since the best results the GA gets involve filling up some room of size X with two splits of different groups of size $\frac{X}{2}$.

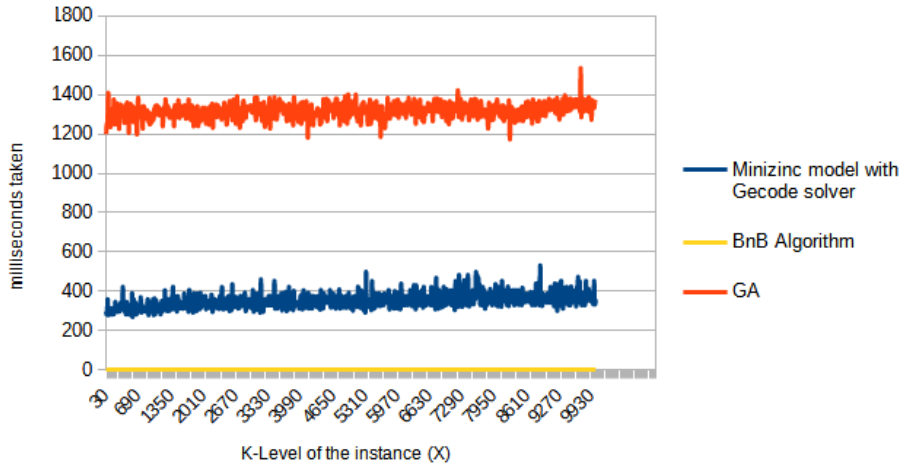


Figure 7.8: Results for Case 2, Restrictive: Milliseconds of the algorithms

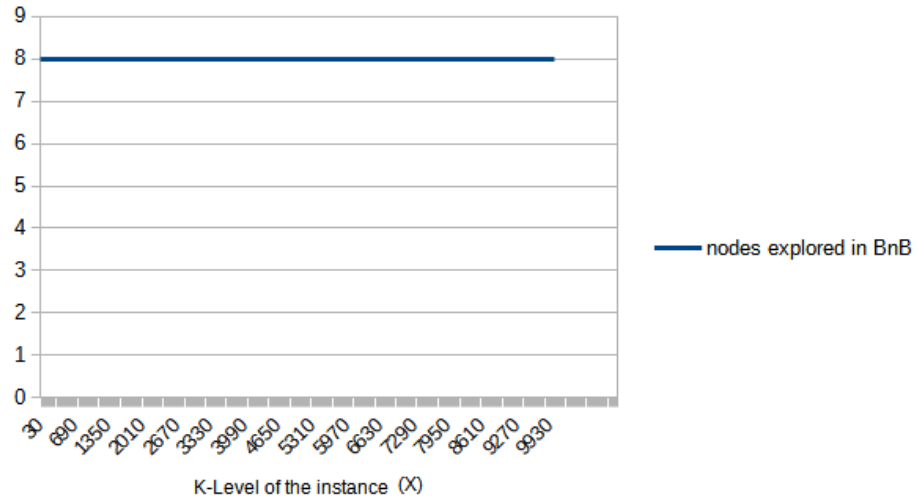


Figure 7.9: Results for Case 2, Restrictive: Nodes explored by BnB

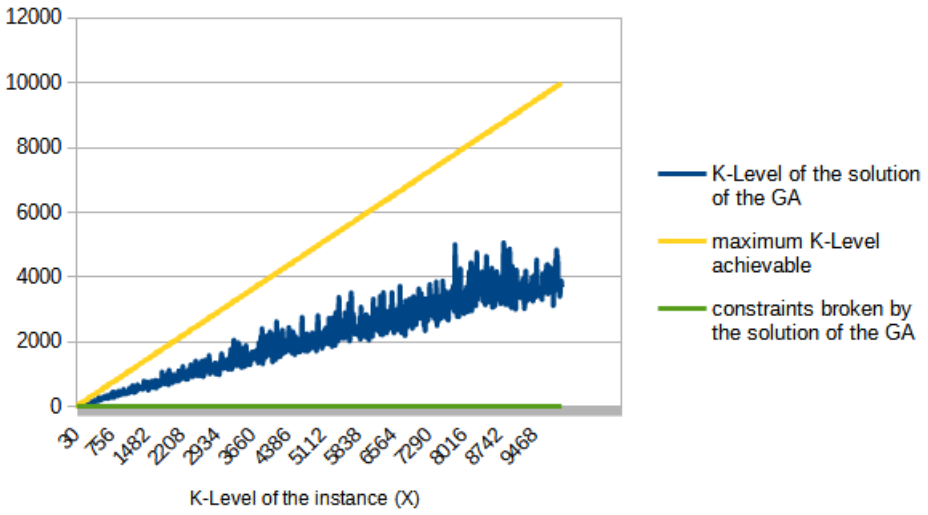


Figure 7.10: Results for Case 2, Restrictive: Average GA Quality of Solutions

Case 2, Non-Restrictive

Here, in Figure 7.11, BnB is again the fastest. In Figure 7.12 we can see it only needs to traverse 8 nodes again to find a global optimum. This is because of the mechanisms of DFS and pruning in the design of the BnB algorithm. We can also observe how the time spent by the Minizinc model is constant and low. The GA in this case is again the slowest.

Finally, in Figure 7.13 we can see the average performance of the GA in terms of quality. We again observe how no constraints are broken, and the results are far better since we deal with a non-restrictive instance. In this case, sometimes it is so good that it returns an average of 10 points of the global optimum. This is because this case goes somewhat in favour of the heuristic, as we already mentioned when discussing its design, since there are some groups that can go entirely to some rooms. We can see the effect of this configuration in the results obtained.

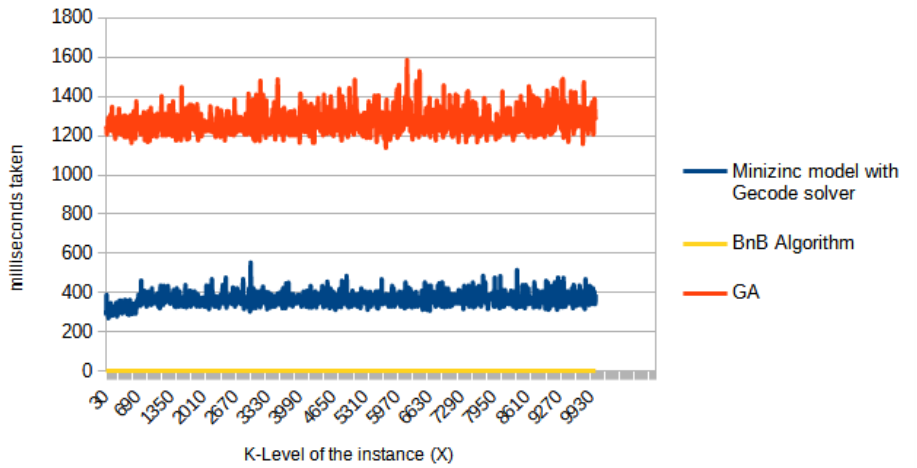


Figure 7.11: Results for Case 2, Non-Restrictive: Milliseconds of the algorithms

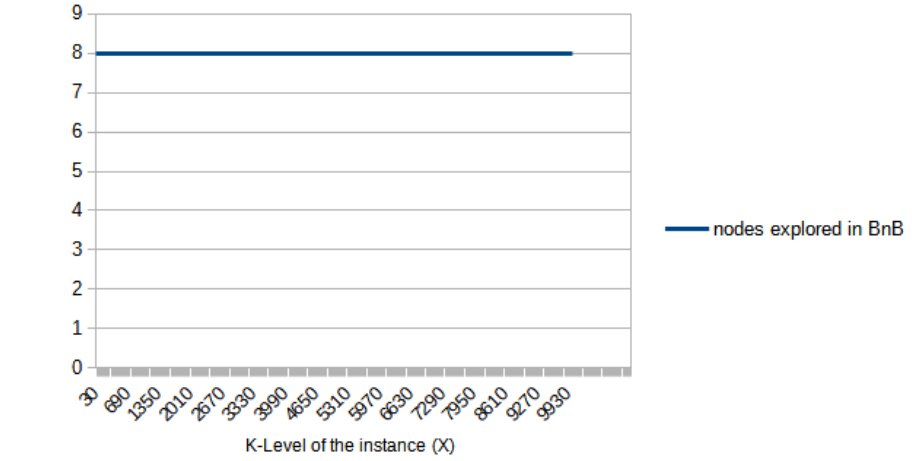


Figure 7.12: Results for Case 2, Non-Restrictive: Nodes explored by BnB

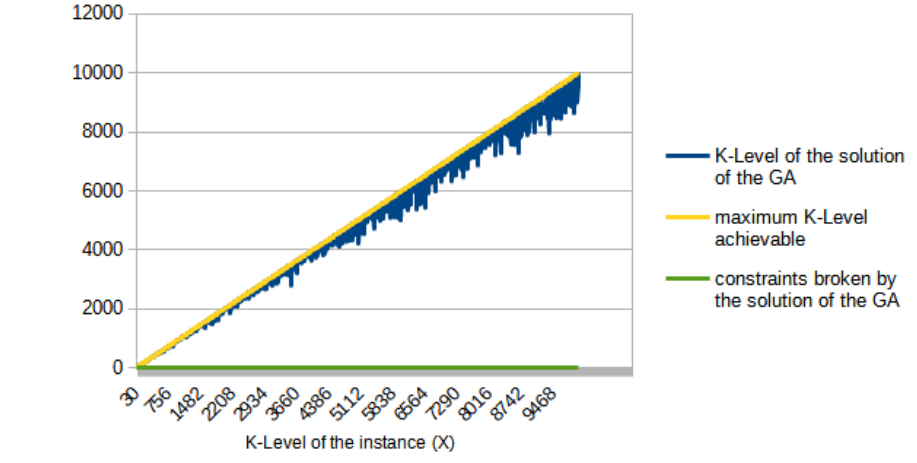


Figure 7.13: Results for Case 2, Non-Restrictive: Average GA Quality of Solutions

7.3. CONCLUSIONS

We do not have any indicators with statistical relevance that we can use to support a definitive decisions. Nonetheless, we can empirically conjecture the following:

a) The BnB algorithm is the slowest when pruning does not discard the whole tree or it does not reach a termination condition.

b) The results by the GA tend to be suboptimal, especially in restrictive cases where the designed heuristic is not applicable.

c) The Minizinc model using the Gecode solver generally outperforms the BnB algorithm due to Gecode's internal architecture while also offering in all cases a global optimum.

d) When the BnB algorithm can find the solution in constant time, so can the version of the Minizinc model.

Thus, we have strong empirical evidence to suggest that the Minizinc model is the most effective solution among the three presented in this work, although we lack a statistical analysis to back this up.

CHAPTER 8

Achievements, obstacles and future lines of work

Anonymized data is data that has been altered in a way that makes it impossible, or very difficult, to identify the person associated with it.

The process of data anonymization removes personally identifiable information from a dataset while ensuring the data remains functional for software business analytics, customer support, development and testing, and other use cases.

Nowadays, data anonymization is crucial for enterprises: as the amount of data companies collect and store increases, the risk of a data breach or compliance violation is greater than ever, resulting in the clients' loss of trust, or costly lawsuits.

In this context: we achieved the following goals, faced the following obstacles and devised the following future lines of work.

8.1. GOALS ACHIEVED

The objectives described in section 1.2 have been achieved. The work carried out can be summarized, schematically, in the following points:

1. Study and understand the problem of data anonymization, both from the point of view of its ethical and economic implications and the importance of the use of data for scientific, demographic studies, etc.

2. Know and understand the need to preserve the $K-$level of anonymity of an already $K-$anonymized database when resources are assigned to the entities that appear in said database. In particular, but not limited to, the problem of assigning appointments for medical check-ups.

3. Formally, identify the computational complexity of the underlying mathematical problem and its relationship with the Partition problem.

4. Design algorithms to calculate an exact solution to the problem of $K-$Anonymity for small size datasets, using: constraint programming and BnB. Study and use genetic algorithms to solve $K-$Anonymity in polynomial time.

5. Perform tests and obtain results that verify the established properties and serve to compare and evaluate the exact solutions in small instances with BnB and constraint programming, and the calculation time with the approximate solutions obtained through genetic algorithms and the efficiency in the execution.

8.2. DIFFICULTIES ENCOUNTERED

8.2.1. Choices of the Genetic Algorithm design

Gene repair

Gene repair in the context of $K-$Anonymity was a tricky mechanism to design. The code felt unreadable, and the algorithm complexity was too high, well over $O\left(r \times g\right)$, where r is the number of rooms, and g is the number of groups, as can be seen in Appendix D.

Creation of growing cases for the exhaustive approach

The exhaustive BnB algorithm was too good at finding the upper bound in arithmetically growing cases with no special treatment to any member of the problem instance. It was a challenge to create a case that could:

1. Grow increasingly and slowly so that the growth in complexity for a higher number of patients is clear.
2. Missing the upper bound determination for all cases while growing.
3. Have a small enough first instance in order to compute at least a certain amount of data points.

8.3. FUTURE LINES OF WORK

The results achieved beg the following questions:

1. In BnB, what performance increases can be gained if the following mechanisms are applied:
   a) If all rooms are full after assigning all patients, but the upper bound is higher than the smallest room, in which cases can the upper bound be decreased to the size of the smallest room, thus stopping the search?
   b) What configurations of remaining groups and patients can be known to not be worth exploring without exploring them? i.e. Given some $K-$Level and some remaining capacities, and some remaining groups, where there is still a room with capacity higher than K, and some group yet unassigned bigger than K, when can it be known that exploring those branches will not yield a better $K-$Level.
   c) Heuristics to prune branches, as well as stopping the search all together, resulting in a non-exhaustive version of BnB, with which performance and which corner cases.
2. In our GAs, what performance increases can be gained if the following mechanisms are applied:
   a) Other gene repair mechanisms, like fixing up the capacities of one room that is over capacity.
   b) Other mutation mechanisms that keep the integrity of the groups, let's say that a `Solution` class can keep the invariants of the groups by always assigning the exact amount, and on mutation, re-configuring the assignations while keeping the exact amount.
   c) Using a numerical fitness score instead of a deterministic fitness score. This numerical fitness score would involve some trade-off between constraints broken and higher $K-$Levels.
   d) Study a different combination of different design choices that could perform better than the one chosen, (i.e. the aforementioned design choices and a room based crossover).
3. How good can it be to pre-process the group and room arrays in an instance? Meaning, eliminating small rooms or splitting big groups from our instance.
4. Certain machine learning algorithms can also effectively analyze patterns found in anonymized datasets, making it easier to re-identify the person behind the data. Analyze these attacks to anonymized databases.

5. Analyze the accuracy change of these attacks if some quasi-identifier was anonymized using the algorithms defined in this work.
6. How can the algorithms created account for concepts that further help anonymize datasets, such as $L-$Diversity and $T-$Closeness, as defined in [11] and [12], respectively.
7. Study the inapproximability of the problem as defined in the definition of inapproximability, and the limit of the performance ratio of 2.

Bibliography

[1] Zakariae El Ouazzaniand and Hanan El Bakkali. A new technique ensuring privacy in big data: K-anonymity without prior value of the threshold k. Journal homepage: https://www.sciencedirect.com/science/article/pii/S187705091830108X, 2018.

[2] Rafael Caballero, Sagar Sen, and Jan F. Nygard. Anticipating anonymity in screening program databases. 2017.

[3] Mars Wave. Tfg_files - bachelor degree work files. https://github.com/Mars-Wave/TFG_Files. GitHub repository, 2024.

[4] Traian Marius Truta, Farshad Fotouhi, and Daniel Barth-Jones. Global disclosure risk measures and k-anonymity property for microdata. Journal homepage: https://www.emis.de/journals/AUA/pdf/45_683_marius_truta.pdf, 2006.

[5] Latanya Sweeny. K-anonymity: a model for protecting privacy, 2002.

[6] Oana Niculaescu. LaTeX Wikibook. k-Anonymity and cluster based methods for privacy, https://elf11.github.io/2017/04/22/kanonymity.html, 2017.

[7] J-P. Barthelemy, G. Cohen, and A. Lobstein. Algorithmic complexity, and communication problems, 1996. ISBN: 1-85728-451-8 HB.

[8] G. Syswerda. A study of reproduction in generational and steady-state genetic algorithms. In Gregory J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, San Mateo, CA, USA, 1991. Morgan Kaufmann.

[9] John H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975. Reprinted by MIT Press, 1992.

[10] George G. Mitchell, Diarmuid O'Donoghue, David Barnes, and Mark McCarville. Generepair - a repair operator for genetic algorithms. Department of Computer Science, National University of Ireland Maynooth, 2004.

[11] Ashwin Machanavajjhala, Johannes Gehrke, Daniel Kifer, and Muthuramakrishnan Venkitasubramaniam. L-diversity: privacy beyond k-anonymity. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006.

[12] Ninghui Li, Tiancheng Li, and Suresh Venkatasubramanian. T-closeness: Privacy beyond k-anonymity and l-diversity. In *23rd International Conference on Data Engineering (ICDE'07)*, pages 106–115. IEEE, 2007.

APPENDICES

APPENDIX A

Configuration to generate plots

Throughout the plots in this work, we present the execution times of our various algorithms, as well as other metrics, on a computer with the following specifications:

- AMD Ryzen 7 5800X3D 8-Core Processor, 3.40 GHz
- 32.0 GB of RAM
- In the case of BnB the algorithm was implemented in C++ (see Annex B) and compiled using MinGW.
- In the case of the GA the algorithm was implemented in C++ (see Annex C) and compiled using MinGW.
- In the case of Minizinc the algorithm was implemented in Minizinc (see the implementation in the Minizinc section 6.2) and ran using the Minizinc framework on the Gecode 6.3.0 solver.

If the implementation of the algorithms is solicited, it can be found in [3].

APPENDIX B

Exhaustive BnB Algorithm C++

This is the main skeleton of the BnB algorithm:

```
using namespace std;

// Recursive function to find the next useful, valid distribution
bool findNextUsefulDistribution(const int remainingPatients, const vector<vector<int> > &currentSolution,
                                const vector<int> &roomCapacities, set<vector<vector<int> > > &exploredDistributions,
                                const int &kLevelToBeat, vector<int> &nextDistribution) {
    if (remainingPatients != 0 && remainingPatients <= kLevelToBeat) {
        return false; // Optimization:  No use in exploring this one further, since split resulted in bad K-Level already
    }

    // If the current distribution sums up to the remainingPatients and is within the number of rooms and is useful (all elements > minKlevel)
    if (remainingPatients == 0 && nextDistribution.size() <= roomCapacities.size() &&
        findKLevelOfNextBranch(nextDistribution) > kLevelToBeat) {
        // Check if this distribution has not been used already
        vector<vector<int> > currentDistWrapper = currentSolution; // Match the set type for some branch node
        currentDistWrapper.push_back(nextDistribution);
        if (!exploredDistributions.contains(currentDistWrapper)) {
            return true; // Return true to indicate a valid distribution has been found
        }
        return false;
    }

    // If the number of rooms used exceeds the available rooms, return
    if (nextDistribution.size() >= roomCapacities.size()) {
        return false;
    }

    // Explore further patient distributions
    for (int i = min(remainingPatients, roomCapacities[nextDistribution.size()]); i >= 0; --i) {
        //only check promising patient splits
        if (i == 0 || i > kLevelToBeat) {
            // Include i patients in the current distribution
            nextDistribution.push_back(i);
            // Recursively find distributions with the remaining patients
            if (findNextUsefulDistribution(remainingPatients - i, currentSolution, roomCapacities,
                                           exploredDistributions, kLevelToBeat, nextDistribution)) {
                return true; // Return immediately if a valid distribution is found
            }
            // Backtrack:  remove the last element and continue
            nextDistribution.pop_back();
        }
    }

```

```
    return false; // No valid distribution was found in this path
}

pair<vector<int>, bool> getNextUsefulPatientDistribution(const vector<vector<int> > &currentSolution,
                                                         set<vector<vector<int> > > &exploredDistributions,
                                                         const int kLevelToBeat, const int totalPatients,
                                                         const vector<int> &roomCapacities) {
    // Current setting is not promising and different from root
    if (findKLevelOfBranch(currentSolution) != -1 &&
        findKLevelOfBranch(currentSolution) <= kLevelToBeat) {
        return {{}, false};
    }

    vector<int> nextDistribution = {};

    // Start the recursive distribution generation, including zero patients
    bool foundNewDistribution = findNextUsefulDistribution(totalPatients, currentSolution,
                                                           roomCapacities, exploredDistributions, kLevelToBeat,
                                                           nextDistribution);

    return {nextDistribution, foundNewDistribution};
}

// Function to calculate the remaining capacities
vector<int> calculateNextCapacities(const vector<vector<int> > &currentSolution, const vector<int> &roomCapacities) {
    vector<int> remainingCapacities = roomCapacities;

    // Calculate the sum of each column in the currentSolution matrix
    for (int col = 0; col < roomCapacities.size(); ++col) {
        int columnSum = 0;

        for (const auto &row: currentSolution) {
            if (col < row.size()) { columnSum += row[col]; }
        }
        // Calculate remaining capacity for each room, invariantly >= 0 because of currentSolution's construction
        remainingCapacities[col] = roomCapacities[col] - columnSum;
    }

    return remainingCapacities;
}

void recursiveKanonymousSearch(vector<vector<int> > &bestSolution, vector<vector<int> > &currentSolution,
                               int &globalMin, const vector<int> &patientGroups,
                               const vector<int> &roomCapacities, const int maximumKLevelPossible,
                               set<vector<vector<int> > > &exploredDistributions,
                               int groupIndex = 0) {
    if (globalMin >= maximumKLevelPossible) {
        // Termination condition:  K-level of assignments equals the smallest group in max bound
        // Backtrack condition:  We have already found a solution of a K-level and we need to backtrack until the first promising node
        return;
    }

    if (groupIndex >= patientGroups.size()) {
        int currentMin = findKLevelOfBranch(currentSolution);
        if (currentMin > globalMin) {
            globalMin = currentMin;
```

```
100             bestSolution = currentSolution;
101         }
102         return;
103     }
104
105     // Continue exploring all possible valid and unexplored distributions for the current group
106     while (globalMin < maximumKLevelPossible) {
107         // Get the next valid and unexplored distribution for the current group
108         auto nextBranch = getNextUsefulPatientDistribution(
109             currentSolution, exploredDistributions, globalMin, ↩
                ↪ patientGroups[groupIndex],
110             calculateNextCapacities(currentSolution, roomCapacities));
111
112         // If there is no more promising and unexplored distributions, exit the loop
113         if (!nextBranch.second) {
114             // Mark the current distribution as explored
115             exploredDistributions.emplace(currentSolution);
116             break;
117         }
118
119         // Push the next valid distribution to the current solution
120         currentSolution.push_back(nextBranch.first);
121
122         // Mark the current distribution as explored
123         exploredDistributions.emplace(currentSolution);
124
125         // Recursively search for the next level
126         recursiveKanonymousSearch(bestSolution, currentSolution, globalMin, ↩
                ↪ patientGroups,
127                                   roomCapacities, maximumKLevelPossible, ↩
                                        ↪ exploredDistributions, groupIndex + 1);
128
129         // Backtrack:  remove the last element and continue
130         currentSolution.pop_back();
131     }
132 }
133
134 int findSmallestGroupSplitThatFits(vector<int> patientGroups, const ↩
        ↪ vector<int> &roomCapacities) {
135     while (true) {
136         // Sort patient groups in ascending order
137         ranges::sort(patientGroups);
138
139         for (int capacity: roomCapacities) {
140             if (patientGroups.front() <= capacity) {
141                 return patientGroups.front();
142             }
143         }
144         vector<int> splitGroups;
145
146         int half1 = patientGroups.front() / 2;
147         int half2 = patientGroups.front() - half1;
148
149         // Add the new subgroups to the splitGroups vector
150         splitGroups.push_back(half1);
151         splitGroups.push_back(half2);
152
153
154         // Replace the original patientGroups with splitGroups
155         patientGroups = splitGroups;
156     }
157 }
158
159
160 tuple<vector<vector<int> >, int, int> findKanonymousDistribution(const ↩
        ↪ vector<int> &patientGroups,
161                                                                  const ↩
                                                                      ↪ vector<int> ↩
                                                                      ↪ &roomCapacities) ↩
                                                                      ↪ {
162     vector<vector<int> > bestSolution; //modified in recursion, return value
163     int globalMin = -2; //modified in recursion
164     set<vector<vector<int> > > exploredDistributions = {}; //modified in recursion
```

```
    {
        vector<vector<int> > currentSolution = {}; //modified in recursion
        recursiveKanonymousSearch(bestSolution, currentSolution, globalMin, patientGroups, roomCapacities,
                                  findSmallestGroupSplitThatFits(patientGroups, roomCapacities), exploredDistributions);
    }

    return {bestSolution, globalMin, exploredDistributions.size()};
}
```

If the full implementation is solicited, it can be found in [3].

APPENDIX C

Genetic Algorithm C++

Excluding unnecessary functions, library includes and declarations, the GA in C++ is the following:

```
// Global variables

// Best K-Level possible is given as an argument
int MAXIMUM_POSSIBLE_KLEVEL;

// Define the static capacity of a room (e.g., maximum capacity).
int ROOM_CAPACITY;

// Define the group range of patients.
int PATIENTS_GROUPS;

// Define the total number of rooms available (e.g., for allocation).
int NUMBER_ROOMS;

// Define the maximum number of generations
int MAX_GENERATIONS;

// Define the population size
int POPULATION_SIZE;

// Allow printing lines in stdout 0=no, 1=yes
int PRINT_TO_COUT;

// Result mode
// 0=mean of 10 points for 1 result of total runtime + K-level of best + Overall violations
// 1=growth evolution of best in every generation with K-Level and violations, with generations until convergence
int RESULT_MODE;

// MetaHeuristicVersion or Random version 0=Random, 1=Meta
int SOLUTION_TYPE;

//output file
std::string OUTPUT;

// Aliases for code clarity
enum class SolutionType : int {
    RandomType = 0,
    HeuristicGuidedType = 1,
    SelfRepairingHeuristicGuidedType = 2
};

// Data definitions
struct PatientGroup {
    int amountOfPatients;
};

struct Room {
    int capacity;
};

Solution *SelectParent(std::vector<Solution *> &population) {
    // Implementation of tournament selection

```

```
54      // Defintion of the size of the tournament (2 for a binary tournament).
55      const int tournamentSize = 2;
56
57      // Randomly select individuals for the tournament.
58      auto bestParent = population[rand() \%
59                                   population.size()];
60      FitnessScore bestFitness = bestParent->getFitness();
61
62      for (int i = 0; i < tournamentSize - 1; ++i) {
63          int randomIndex =
64                  rand() \%
65                  population.size(); // Randomly choose an individual from the population.
66          const FitnessScore currentFitness = ↩
                ↪ population[randomIndex]->getFitness();
67
68          if (bestFitness < currentFitness) {
69              bestParent = population[randomIndex];
70              bestFitness = currentFitness;
71          }
72      }
73
74      return bestParent;
75  }
76
77  Solution *Crossover(const Solution * &parent1, const Solution * &parent2) {
78      // Implementation of crossover method, two-point
79      // crossover, to create a child solution from the parents.  Returns the child
80      // solution as a vector of Assignments
81      Solution *child = parent1->clone();
82
83
84      // Randomly choose two distinct columns for the crossover.
85      int point1 = rand() \% NUMBER_ROOMS;
86      int point2 = rand() \% NUMBER_ROOMS;
87
88      // Ensure point1 is less than point2.
89      if (point1 > point2) {
90          std::swap(point1, point2);
91      }
92
93      // Replace groups from parent1 with groups from point1 to point2 from parent2
94      for (int j = point1; j < std::min(point2, NUMBER_ROOMS); j++) {
95          for (int i = 0; i < PATIENTS_GROUPS; i++) {
96              int amount = parent2->getPeopleFromGroupInRoom(i, j);
97              child->assignAmountInGroupToRoom(i, j, amount);
98          }
99      }
100
101     return child;
102 }
103
104 void Mutate(Solution *child) {
105     // Implement a mutation method to introduce random changes to the child
106     // solution.
107     // Randomly choose an assignment to mutate.
108     int mutationI = rand() \% PATIENTS_GROUPS;
109     int mutationJ = rand() \% NUMBER_ROOMS;
110
111     child->mutateAmountInGroupAndRoom(mutationI, mutationJ);
112 }
113
114 Solution *GetBestSolution(std::vector<Solution *> &population) {
115     // Find and return the best solution from the population based on fitness.
116     // The best solution is the one with the best fitness score.
117
118     Solution *bestSolution = population[0];
119     FitnessScore bestFitness = bestSolution->getFitness();
120
121     for (Solution * &solution: population) {
122         FitnessScore fitness = solution->getFitness();
123
124         if (bestFitness < fitness) {
125             bestFitness = fitness;
```

```
                bestSolution = solution;
            }
        }
        return bestSolution;
    }

    void ReplaceWeakest(std::vector<Solution *> &population, Solution * &child) {
        // Compare the fitness of the child solution with the fitness of the weakest
        // solution in the population.  If the child's fitness is better, replace the
        // weakest solution with the child solution.

        // Evaluate fitness
        FitnessScore childFitness = child->getFitness();

        // Find the index of the weakest solution in the population.
        FitnessScore weakestFitness = population[0]->getFitness();
        int weakestIndex = 0;

        for (int i = 1; i < population.size(); ++i) {
            FitnessScore fitness = population[i]->getFitness();
            if (fitness < weakestFitness) {
                weakestFitness = fitness;
                weakestIndex = i;
            }
        }

        // Compare the child's fitness with the weakest solution's fitness.
        if (weakestFitness < childFitness) {
            // Replace the weakest solution with the child solution.
            population[weakestIndex] = child;
        }
    }

    Solution *GeneticAlgorithm(std::vector<Solution *> &population) {
        // Main Genetic Algorithm Loop
        for (int generation = 0; generation < MAX_GENERATIONS; ++generation) {
            // Selection
            const auto *parent1 = SelectParent(population);
            const auto *parent2 = SelectParent(population);

            // Crossover
            Solution *child = Crossover(parent1, parent2);

            // Mutation
            Mutate(child);

            //Gene repair
            child->repairSomeGroupInGene();

            // Update population
            ReplaceWeakest(population, child);
        }
        // Output the best solution
        return GetBestSolution(population);
    }
```

It uses the Solution class, using polymorphism to describe the different versions defined as:

```
    //Problem structures

    struct FitnessScore {
        int roomsOverMaxCapacity, groupsOverMaxCapacity, levelOfK;

        //Deterministic Fitness Scoring

        // Define the less than operator to compare FitnessScore objects
        // i.e:  a < b implies a is less fit than b
        bool operator<(const FitnessScore &other) const {
            // Compare based on roomsOverMaxCapacity
            if (roomsOverMaxCapacity != other.roomsOverMaxCapacity) {
                return roomsOverMaxCapacity > other.roomsOverMaxCapacity;
            }
            // Compare based on groupsOverMaxCapacity
```

```
        if (groupsOverMaxCapacity != other.groupsOverMaxCapacity) {
            return groupsOverMaxCapacity > other.groupsOverMaxCapacity;
        }
        // If groupsOverMaxCapacity is equal, compare based on levelOfK
        return levelOfK < other.levelOfK;
    }
};

struct ProblemContext {
    static std::vector<Room> roomCapacities;
    static std::vector<PatientGroup> patientGroups;
};

// Initialize static members
std::vector<Room> ProblemContext::roomCapacities;
std::vector<PatientGroup> ProblemContext::patientGroups;

class Solution {
protected:
    std::vector<std::vector<int> > assignments;

    std::vector<int> peoplePerRoom;

    std::vector<int> patientsPerGroup;

    std::priority_queue<std::tuple<int, int, int>,
        std::vector<std::tuple<int, int, int> >,
        std::greater<> > kLevelminHeap;

    int minPeople;
    int roomsOverCapacityCount;
    int wrongGroupsCount;


    /// Initialize the min-heap with non-zero assignment values
    void initKLevelMinHeap() {
        for (size_t i = 0; i < assignments.size(); ++i) {
            for (size_t j = 0; j < assignments[i].size(); ++j) {
                if (assignments[i][j] > 0) {
                    kLevelminHeap.emplace(assignments[i][j], i, j);
                }
            }
        }
    }

    void clearMinHeap() {
        std::priority_queue<std::tuple<int, int, int>,
            std::vector<std::tuple<int, int, int> >,
            std::greater<> > newHeap;

        while (!kLevelminHeap.empty()) {
            auto [value, groupId, roomId] = kLevelminHeap.top();
            if (assignments[groupId][roomId] == value) {
                newHeap.push(kLevelminHeap.top());
            }
            kLevelminHeap.pop();
        }

        kLevelminHeap = std::move(newHeap);
    }

    void updatePeoplePerRoom(int roomId, int delta) {
        int oldCount = peoplePerRoom[roomId];
        peoplePerRoom[roomId] += delta;
        if (oldCount <= getRoomCapacity(roomId) &&
            peoplePerRoom[roomId] > getRoomCapacity(roomId)) {
            // We exceeded capacity in this room
            roomsOverCapacityCount++;
        } else if (oldCount > getRoomCapacity(roomId) &&
                   peoplePerRoom[roomId] <= getRoomCapacity(roomId)) {
            // We went from exceeding capacity to being in range
            roomsOverCapacityCount--;
        }
```

```
89      }
90
91      void updatePatientsPerGroup(int groupId, int delta) {
92          int oldGroupCount = patientsPerGroup[groupId];
93          patientsPerGroup[groupId] += delta;
94          if (oldGroupCount != getPatientGroupCount(groupId) &&
95              patientsPerGroup[groupId] == getPatientGroupCount(groupId)) {
96              wrongGroupsCount--;
97          } else if (oldGroupCount == getPatientGroupCount(groupId) &&
98                     patientsPerGroup[groupId] != ↩
                           ↪ getPatientGroupCount(groupId)) {
99              wrongGroupsCount++;
100         }
101     }
102
103     static int getRoomCapacity(int roomId) {
104         return ProblemContext::roomCapacities[roomId].capacity;
105     }
106
107     static int getPatientGroupCount(int groupId) {
108         return ProblemContext::patientGroups[groupId].amountOfPatients;
109     }
110
111     void updateKLevel() {
112         // Clear min entries from the heap
113         while (!kLevelminHeap.empty()) {
114             auto [value, row, col] = kLevelminHeap.top();
115             if (assignments[row][col] == value) {
116                 minPeople = value;
117                 return;
118             }
119             kLevelminHeap.pop();
120         }
121     }
122
123     void initAmountInGroupToRoom(int groupId, int roomId, int amount = -1) {
124         if (amount == -1) {
125             amount =
126                     rand() \%
127                     (std::min(getRoomCapacity(roomId),
128                               getPatientGroupCount(groupId)) + 1);
129         }
130         if (amount > 0) {
131             // Update K-level
132             kLevelminHeap.emplace(amount, groupId, roomId);
133             updatePeoplePerRoom(roomId, amount);
134             updatePatientsPerGroup(groupId, amount);
135         }
136         // Update the assignments matrix
137         assignments[groupId][roomId] = amount;
138     }
139
140
141     void updateDataStructures(int groupId, int roomId, int amount) {
142         int oldAmount = assignments[groupId][roomId];
143         assignments[groupId][roomId] = amount;
144         if (amount > 0) {
145             kLevelminHeap.emplace(amount, groupId, roomId);
146             // Update K-level
147             updateKLevel();
148         }
149         updatePeoplePerRoom(roomId, amount - oldAmount);
150         updatePatientsPerGroup(groupId, amount - oldAmount);
151     }
152
153 public:
154     // Initialize the solution with rooms and groups
155     Solution()
156         : assignments(PATIENTS_GROUPS, std::vector<int>(NUMBER_ROOMS, 0)),
157           peoplePerRoom(NUMBER_ROOMS, 0), patientsPerGroup(PATIENTS_GROUPS, 0),
158           minPeople(INT_MAX), roomsOverCapacityCount(0), ↩
                  ↪ wrongGroupsCount(PATIENTS_GROUPS), kLevelminHeap(
159               std::priority_queue<std::tuple<int, int, int>,
```

```
                    std::vector<std::tuple<int, int, int> >,
                    std::greater<> >{}) {
    }

    virtual ~Solution() = default;

    void initializeSolution() {
        // Initialize internal data
        for (int groupId = 0; groupId < PATIENTS_GROUPS; ++groupId) {
            for (int roomId = 0; roomId < NUMBER_ROOMS; ++roomId) {
                initAmountInGroupToRoom(groupId, roomId); // Random init
            }
        }

        // Update KLevel after initialization
        updateKLevel();
    }

    void assignAmountInGroupToRoom(int groupId, int roomId, int amount) {
        updateDataStructures(groupId, roomId, amount);
    }

    virtual void repairSomeGroupInGene() {
    };

    virtual void mutateAmountInGroupAndRoom(int groupId, int roomId) = 0;

    ///auxiliary function for crossover
  [[nodiscard]]  virtual Solution *clone() const = 0;

[[nodiscard]]    int getPeopleFromGroupInRoom(int group, int room) const {
        return assignments[group][room];
    }

    /// Get the number of people in a specific room
  [[nodiscard]]  int getPeopleInRoom(int roomId) const { return
      ↪ peoplePerRoom[roomId]; }

    /// Get the number of rooms that are over capacity.
  [[nodiscard]]  int getRoomsOverCapacityCount() const { return
      ↪ roomsOverCapacityCount; }

    /// Get the number of groups that have an incorrect number of patients.
   [[nodiscard]] int getWrongGroupsCount() const { return wrongGroupsCount; }

    /// Get the minimum number of people from a group assigned to any
    /// room (K-level).
    [[nodiscard]] int getKLevel() const { return minPeople; }

    /// Get defined fitness score
    [[nodiscard]] FitnessScore getFitness() const {
        return {roomsOverCapacityCount, wrongGroupsCount, getKLevel()};
    }

    //printer
    void printSolution() const {
        // Print assignments matrix
        std::cout << "Assignments Matrix:" << std::endl;
        std::cout << "--------------------" << std::endl;
        for (int groupId = 0; groupId < PATIENTS_GROUPS; ++groupId) {
            std::cout << "Group " << std::setw(3) << groupId + 1 << ": ";
            for (int roomId = 0; roomId < NUMBER_ROOMS; ++roomId) {
                std::cout << std::setw(3) << assignments[groupId][roomId] << "
                    ↪ " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;

        // Print other statistics
        std::cout << "Statistics:" << std::endl;
        std::cout << "-----------" << std::endl;
        std::cout << "Bad Rooms: " << roomsOverCapacityCount << std::endl;
```

```
230         std::cout << "Bad Groups: " << wrongGroupsCount << std::endl;
231         std::cout << "K-Level: " << minPeople << std::endl;
232         std::cout << std::endl;
233     }
234 };
235 
236 /** Override mutateAmountInGroupAndRoom in a standard, random approach
237      * In this approach there is:
238      *       - a 100\%  probability to make the assignation a random split of the group
239      */
240 class RandomSolution : public Solution {
241 public:
242     using Solution::Solution; // Inherit constructors if needed
243 
244     [[nodiscard]] Solution *clone() const override {
245         return new RandomSolution(*this);
246     }
247 
248     //no repair
249     void repairSomeGroupInGene(int groupId, int roomId) {
250         clearMinHeap();
251     }
252 
253     // Override mutateAmountInGroupAndRoom for random approach
254     void mutateAmountInGroupAndRoom(int groupId, int roomId) override {
255         int amount =
256                 std::rand() \%
257                 (std::min(getRoomCapacity(roomId),
258                           getPatientGroupCount(groupId)) + 1);
259 
260         assignAmountInGroupToRoom(groupId, roomId, amount);
261     }
262 };
263 
264 /** Overrides mutateAmountInGroupAndRoom for HeuristicGuidedType approach
265      * In this approach there is:
266      *       - a 25\%  probability to make the assignation 0
267      *       - a 50\%  probability to make the assignation the max group count for that group
268      *       - a 25\%  probability to make the assignation a random split of the group
269      */
270 class HeuristicGuidedSolution : public Solution {
271 public:
272     using Solution::Solution; // Inherit constructors if needed
273 
274     [[nodiscard]] Solution *clone() const override {
275         return new HeuristicGuidedSolution(*this);
276     }
277 
278     //no repair
279     void repairSomeGroupInGene(int groupId, int roomId) {
280         clearMinHeap();
281     }
282 
283     void mutateAmountInGroupAndRoom(int groupId, int roomId) override {
284         // Generate a random number between 0 and 99
285         int randValue = std::rand() \%  100;
286 
287         int amount;
288         if (randValue < 25) {
289             // 25% probability
290             amount = 0;
291         } else if (randValue < 75) {
292             // 50% probability (25% + 50% )
293             amount = getPatientGroupCount(groupId);
294         } else {
295             // 25% probability
296             amount = std::rand() \%  (std::min(getRoomCapacity(roomId), getPatientGroupCount(groupId)) + 1);
297         }
298 
```

```
299             assignAmountInGroupToRoom(groupId, roomId, amount);
300         }
301     };
302
303
304     /** Overrides mutateAmountInGroupAndRoom for SelfRepairingHeuristicGuidedSolution approach
305       * In this approach there is:
306       *       - a 25\%  probability to make the assignation 0
307       *       - a 50\%  probability to make the assignation the max group count for that group
308       *       - a 25\%  probability to make the assignation a random split of the group
309       *
310       * And a reparation of the matrix ensuring that, after a mutation, there never is:
311       *       - a group assigned unequal to the statement
312       *       - a room exceeding capacity
313       */
314     class SelfRepairingHeuristicGuidedSolution : public Solution {
315     private:
316         int getTotalGroupDifferenceForAssignment(int groupId, int roomId) {
317             int groupCount = getPatientGroupCount(groupId);
318             int currentGroupSum = patientsPerGroup[groupId];
319             if (currentGroupSum == groupCount) {
320                 return -1;
321             }
322
323             return std::max(std::min(std::max(0, assignments[groupId][roomId] - (currentGroupSum - groupCount)),
324                                      getRoomCapacity(roomId) - peoplePerRoom[roomId]), 0);
325         }
326
327         int getRoomDecrementForGroup(int groupId, int roomId) {
328             int roomCapacity = getRoomCapacity(roomId);
329             int currentRoomSum = peoplePerRoom[roomId];
330             if (currentRoomSum <= roomCapacity) {
331                 return -1;
332             }
333
334             return std::rand() \%  (std::max(0, assignments[groupId][roomId] - (currentRoomSum - roomCapacity)) +
335                                     1);
336         }
337
338     public:
339         using Solution::Solution; // Inherit constructors if needed
340
341         void repairSomeGroupInGene() override {
342             std::vector<int> badGroups;
343             for (int i = 0; i < PATIENTS_GROUPS; i++) {
344                 if (getPatientGroupCount(i) != patientsPerGroup
345                     [i]) {
346                     badGroups.push_back(i);
347                 }
348             }
349             if (badGroups.empty()) {
350                 clearMinHeap();
351                 return;
352             }
353             int wrongGroup = badGroups[rand() \%  badGroups.size()];
354
355             if (getPatientGroupCount(wrongGroup) > patientsPerGroup[wrongGroup]) {
356                 // Underassignment case
357                 // Collect rooms with available capacity
358                 std::vector<int> availableRooms;
359                 for (int r = 0; r < NUMBER_ROOMS; r++) {
360                     if (getRoomCapacity(r) > peoplePerRoom[r]) {
361                         availableRooms.push_back(r);
362                     }
363                 }
364                 while (getPatientGroupCount(wrongGroup) != 
```

```
                    ↪ patientsPerGroup[wrongGroup]) {
365                 if (availableRooms.empty()) {
366                     // No available rooms to assign patients to, exit the loop
367                     break;
368                 }
369
370                 int targetRoom = availableRooms[rand() \%  ↩
                        ↪ availableRooms.size()];
371                 int availableCapacity = std::max(getRoomCapacity(targetRoom) ↩
                        ↪ - peoplePerRoom[targetRoom], 0);
372
373                 // Determine how many patients we can move to the target room
374                 int patientsToMove = ↩
                        ↪ std::min(getPatientGroupCount(wrongGroup) - ↩
                        ↪ patientsPerGroup[wrongGroup],
375                                                 availableCapacity);
376
377                 // Update assignments and data structures
378                 updateDataStructures(wrongGroup, targetRoom, ↩
                        ↪ assignments[wrongGroup][targetRoom] + patientsToMove);
379
380                 // If the room is now full, remove it from available rooms
381                 if (peoplePerRoom[targetRoom] >= getRoomCapacity(targetRoom)) {
382                     availableRooms.erase(std::remove(availableRooms.begin(), ↩
                            ↪ availableRooms.end(), targetRoom),
383                                          availableRooms.end());
384                 }
385             }
386         } else if (getPatientGroupCount(wrongGroup) < ↩
                ↪ patientsPerGroup[wrongGroup]) {
387             // Overassignment case
388             // Collect rooms with patients on it
389             std::vector<int> roomsOccupied;
390             for (int r = 0; r < NUMBER_ROOMS; r++) {
391                 for (int g = 0; g < PATIENTS_GROUPS; g++) {
392                     if (assignments[g][r] > 0) {
393                         roomsOccupied.push_back(r);
394                         break;
395                     }
396                 }
397             }
398             while (getPatientGroupCount(wrongGroup) != ↩
                    ↪ patientsPerGroup[wrongGroup]) {
399                 if (roomsOccupied.empty()) {
400                     // No available rooms to subtract patients from, exit the loop
401                     break;
402                 }
403
404                 int targetRoom = roomsOccupied[rand() \%  roomsOccupied.size()];
405
406                 // Determine how many patients we can erase in the target room
407                 int newAmount = std::max(
408                     assignments[wrongGroup][targetRoom] - (
409                         patientsPerGroup[wrongGroup] - ↩
                                ↪ getPatientGroupCount(wrongGroup)), 0);
410
411                 // Update assignments and data structures
412                 updateDataStructures(wrongGroup, targetRoom, newAmount);
413
414                 // If the room is now full, remove it from available rooms
415                 if (assignments[wrongGroup][targetRoom] == 0) {
416                     roomsOccupied.erase(std::remove(roomsOccupied.begin(), ↩
                            ↪ roomsOccupied.end(), targetRoom),
417                                         roomsOccupied.end());
418                 }
419             }
420         }
421         clearMinHeap();
422     }
423
424     [[nodiscard]] Solution *clone() const override {
425         return new SelfRepairingHeuristicGuidedSolution(*this);
426     }
```

```
427
428     void mutateAmountInGroupAndRoom(int groupId, int roomId) override {
429         // Generate a random number between 0 and 99
430         int randValue = std::rand() \%  100;
431
432         int amount;
433         if (randValue < 25) {
434             // 25% probability
435             amount = 0;
436         } else if (randValue < 75) {
437             // 50% probability (25% + 50% )
438             amount = getPatientGroupCount(groupId);
439         } else {
440             // 25% probability
441             amount = std::rand() \%  (std::min(getRoomCapacity(roomId), ↩
                    ↪ getPatientGroupCount(groupId)) + 1);
442         }
443
444         assignAmountInGroupToRoom(groupId, roomId, amount);
445     }
446 };
```

If the full implementation is solicited, it can be found in [3].

APPENDIX D

Gene Repair Discarded Costly Designs

The first design for the for the gene repair algorithm was the following:

1. **Handling group sizes**: Initially, we attempt to assign the difference between the currently assigned members and the actual number of people in that group to some room, restricted by the room's capacity.
2. **Propagating Changes**: If the number of assigned members does not yet meet the required count, the adjustment is propagated to the same group in another room.
3. **Handling Room Capacity**: If this adjustment results in exceeding the room's capacity, the number of members assigned to another group in the same room is decreased accordingly.
4. **Recursive Adjustment**: These operations are performed recursively until both all group and room constraints are satisfied.

This design is extremely costly due to the recursive step, and enforcement of constraints on every row and column on the matrix. Because of recursion, the stack may overflow. It also results in a lot of split groups, because instead of helping solutions converge to better ones, it enforces all constraints in one step, resulting in a solution that fulfils all constraints but probably has a bad $K-$Level.

Then, in order to avoid the cost of trying to fix up every constraint in one step, the following design was implemented:

1. **Initialization and Group Iteration**: The algorithm begins by initializing variables to store the new assignment for some group to a room. It then iterates over all groups and rooms.
2. **Calculate Adjustments**: For each group and room, the algorithm calculates the total possible adjustment without violating constraints. If a feasible adjustment is found, the algorithm updates the variables for that assignment for some group to a room, then breaks out of the loop to apply the changes.
3. **Update Data Structures**: When a valid adjustment is identified and applied, the algorithm updates the relevant data structures with the new assignment.
4. **Propagation of Changes**: After applying the adjustment, the algorithm checks whether the number of assigned members matches the required group count. If not, it rechecks and applies the prior steps on that same group.
5. **Handling Room Overcapacity**: If a room exceeds its capacity after the adjustment, the algorithm attempts to resolve this by randomly selecting two different rooms and swapping their assignments for the same group. This operation continues until the room's capacity constraints are satisfied or a predefined depth limit is reached.

This was also costly. This one intended to enforce the restraints on every group, and then for a certain depth of steps, it attempted to help with the constraint for the rooms.

While less costly, the problem with this design was that usually all solutions ended up breaking the capacity of some room. Not only it offered invalid solutions, but the $K-$Level of the invalid

solution was not significantly better either.

After conducting several experiments, it became evident that the design should prioritize simplicity and facilitate convergence, rather than focusing on a complex mechanism to enforce constraints. This insight aligns with the fundamental principles of Genetic Algorithms, which emphasize simplicity and convergence over exhaustive constraint enforcement.

Subsequently, the revised mechanism was designed to address a single group at a time while maintaining respect for room capacities. This design proved to be more effective and was the one ultimately implemented.

APPENDIX E

Total number of global optima for a case more favorable for the heuristic

E.1. RESTRICTIVE CASE:

To know the number of global optima we need to find the number of distinct arrangements of two clusters from group 1, two clusters from group 2 and two clusters from group 3. Since the case is restrictive, we will have 6 positions to assign them. To better solve it, let us rearrange it as a problem of combinatorics:

> *We have 4 bags, two of capacity 1 and two of capacity 2. They are ordered in the following fashion:* [_, __, __, _]*, where _ indicates a slot for an element (so __ denotes two slots). How many different bags can I create with 6 elements, that are two As, two Bs and two Cs and each take 1 capacity? The order of the elements inside the bags of capacity 2 does not result in different arrangements, and the order of the bags between each other also does not matter.*

Approach

1. Find out how many possibilities exist when we put two identical items in the bags in the middle, i.e: $[C, AA, BB, C]$
2. Find out how many possibilities exist when one bag in the middle holds identical items, i.e: $[C, AA, BC, B]$
3. Find out how many possibilities exist when both bags in the middle hold different items, i.e: $[C, AB, BC, A]$
4. Sum all of these possibilities to get the total number of arrangements.

The following list demonstrates all of the possibilities for one of the bags in the middle:

- AA, identical elements
- AB, different elements
- AC, different elements
- BB, identical elements
- BC, different elements
- CC, identical elements

First step:

For the first step, we know that there are 3 possible arrangements from the 6 mentioned that consist of two identical elements. Since there are 2 bags and the order between these two matters, we get $3 \cdot 2 = 6$ possible arrangements. When these are placed, the remaining two elements are identical, and will go into the bags of capacity 1, which is only one possibility. So in total we have 6 possible

arrangements when we put two identical items in the bags in the middle.

Second step:

For the second step, we have 2 different positions where to put the two identical elements. Then we have 3 different combinations of identical elements. Of course, when we place two identical elements in a bag, we cannot use them anymore. Since the other bag needs two different elements, and we only have two remaining elements, there is only 1 possible combination remaining to place. Finally, we have two different elements remaining to place in the bags that can fit one element. This gives us 2 possibilities.

In summary, for the second step we get:

$$\underbrace{2}_{\substack{\text{Two identical elements}\\ \text{to one bag or the other}}} * \underbrace{3}_{\substack{\text{Letter of the}\\ \text{identical elements}}} * \underbrace{2}_{\substack{\text{Permutations of the}\\ \text{remaining two items}}} = \underbrace{12}_{\substack{\text{Total possibilities when}\\ \text{one bag in the middle}\\ \text{holds identical items}}}$$

Third step:

The third step needs to split into two subcases, since when the bags in the middle hold identical items (i.e. $[C, AB, AB, C]$), the elements in the bags of 1 will be identical, and when the bags in the middle have different items (i.e. $[C, AC, AB, B]$), the elements in the bags of 1 will be different.

Following the list above, there are 3 combinations of two different elements. When we place two identical combinations of 2 items in the bags of capacity 2, there is only 1 possibility for the remaining items. This is because we do not care about the order between identical items in the bags of 2 and the bags of 1, respectively. In total we have 3 possibilities.

For the remaining combinations, since there are 3 combinations of two different elements, and they will be different between each other between the 2 bags in the middle, we have $3 \cdot 2 = 6$ possible arrangements for the bags in the middle. Afterwards, we will be left with two different items to place in the remaining 2 bags, which can be trivially done in 2 ways. In total, we have $6 \cdot 2 = 12$ possible arrangements.

For step three, adding the arrangements of the two possible situations, we get a total of 3 + 12=15 arrangements.

Fourth step, addition of three first steps:

In total, we have:

$$\underbrace{6}_{\substack{\text{Step 1:}\\ \text{Two bags of two}\\ \text{identical elements}}} + \underbrace{12}_{\substack{\text{Step 2:}\\ \text{One bag of two}\\ \text{identical elements}}} + \underbrace{15}_{\substack{\text{Step 3:}\\ \text{Two bags of two}\\ \text{different elements}}} = \underbrace{33}_{\text{Total arrangements}}$$

E.2. NON-RESTRICTIVE CASE:

To know the number of global optima we need to find the number of distinct arrangements of two clusters from group 1, two clusters from group 2 and two clusters from group 3. In the non-restrictive version, we will have 7 positions to assign them. To better solve it, let us again rearrange it as a problem of combinatorics:

> *We have 5 bags, three of capacity 1 and two of capacity 2. They are ordered in the following fashion:* [_, __, __, _, _]*, where _ indicates a slot for an element. How many different bags can I create with 6 elements, that are two As, two Bs and two Cs and each take 1 capacity? The order of the elements inside the bags of capacity 2 does not result in different arrangements, and the order of the bags between each other also does not matter.*

E.Total number of global optima for a case more favorable for the heuristic

Approach

Before we used the following approach for four bags, where two bags could hold 2 elements and the remaining two could hold just 1:

1. The number of possibilities that exist when we put two identical items in the bags in the middle is 6.
2. The number of possibilities exist when one bag in the middle holds identical items is 12.
3. The number of possibilities exist when both bags in the middle hold different items is 15.
4. The total number of arrangements is $6 + 12 + 15 = 33$.

Now, our approach will follow all of these steps and it will account for the possibilities generated by adding an extra bag that will end up empty, or will take one element of the other generated bags.

First step:

To create a change in the 6 possible distributions generated before, we know that we can take an element from the bags created and put then in the extra bag. Since the elements are always the same inside the bags of 2, for each of them we get one extra possibility. The elements of the bags on the sides also can be moved, and for each possibility with an empty bag we have 3 new possibilities. In total this makes 5 new arrangements that are created from each of the 6 arrangements calculated before. This makes $6 \cdot 5 = 30$ total arrangements.

Second step:

Similarly, in this second step we account for the extra bag and the new possibilities it provides, except in this case we get one extra possibility from the bags in the middle, since they contain three different letters, makes 3 ways to place one element from the big bags in the new small bag. Accounting as well for the possibilities given by the small bags, we get 6 new arrangements created from the 12 arrangements calculated before. In total, that means for the second step we get $12 \cdot 6 = 72$ total arrangements.

Third step:

Following the same reasoning, for the sub case where the bags in the middle hold identical pairs of elements and for the sub case where the bags in the middle hold different pairs of elements, we have 4 ways to end up with new arrangements by taking one element from them and putting it in the new bag. The possibilities of the small bags are also still 3.

In total, for the third step we get a total of $3 * 7 + 12 * 7 = 105$ possibilities. **Fourth step, addition of three first steps:**

In total, when we add a small extra bag we have:

$$\underbrace{6 * 5}_{\substack{\text{Step 1:}\\ \text{Two bags of two}\\ \text{identical elements}}} + \underbrace{12 * 6}_{\substack{\text{Step 2:}\\ \text{One bag of two}\\ \text{identical elements}}} + \underbrace{3 * 7 + 12 * 7}_{\substack{\text{Step 3:}\\ \text{Two bags of two}\\ \text{different elements}}} = \underbrace{207}_{\substack{\text{Total arrangements}}}$$