MSc

2.º CICLO

FCUP ANO

U.PORTO

An EDSL for Modeling Kidney Exchange Programs

João Paulo Rocha Viana

U.PORTO

FACULDADE DE CIÊNCIAS UNIVERSIDADE DO PORTO

An EDSL for Modeling Kidney Exchange Programs

João Paulo Rocha Viana

Dissertação de Mestrado apresentada à
Faculdade de Ciências da Universidade do Porto em
Ciencia de Computadores

2019

An EDSL for Modeling Kidney Exchange Programs

João Paulo Rocha Viana
Mestrado em Ciência de Computadores
Departamento de Ciencia de Computadores
2019

Orientador
Prof. Pedro Vasconcelos, FCUP

Coorientador
Prof. João Pedro Pedroso, FCUP

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, ______/______/_________

Abstract

Kidney exchange programs have grown from localized, single-site experiences to multi-center, organized and regulated frameworks that are reaching international cooperation. These programs created opportunities for patients suffering from kidney failure to have better chances of finding a compatible organ. As the increase of scale demands more sophisticated approaches, mathematical models for the problem and its numerous variants have motivated the development of software and tools that offer real-word solutions to finding optimal exchange schemes, and also to further research by reducing the time taken to implement specific models and allowing more time for experimenting and testing.

In this dissertation we present PyKPD, a domain-specific language embedded in Python for managing, testing and optimizing kidney exchange program instances. It implements different exchange modalities, offers methods to load data on a pool of donors and patients, generates the mathematical constructs of the model, and provides abstraction when specifying the criteria that will delineate the space of optimal exchanges. The usefulness of the language is validated by expressing several distinct examples of criteria in a simple, idiomatic way, in which pure Python code is mixed without blurring the semantics of the domain.

Resumo

Os programas de doação renal cruzada partiram de pequenas experiências e evoluiram para esquemas regulados, por vezes envolvendo vários centros de transplantação e que começam a atingir a cooperação internacional. Estes programas criaram oportunidades para que pacientes que estão a sofrer de falha renal tenham maior probabilidade de encontrar um orgão compatível. Com o aumento da exigência de abordagens sofisticadas às questões logísticas intrínsecas, os avanços em modelação matemática e a incorparação de diversas modalidades de troca motivaram o desenvolvimento de *software* e de ferramentas que se traduzem em soluções realistas na procura de esquemas ótimos de trocas de rins, e também o de potenciar mais investigação ao reduzir o tempo que implementar experiências específicas leva, deixando mais tempo também para experimentação e teste de hipóteses.

Esta dissertação propõe uma linguagem específica de domínio embebida em Python, denominada PyKPD, cujo propósito é permitir a gestão, teste e otimização de instâncias de programas de doação renal cruzada. São implementadas diferentes modalidades de forma concisa e são disponibilizados métodos para construir uma base de dados relativa aos pacientes e dadores registados no programa, gerar os objetos matemáticos relativos ao modelo, e ainda permitir um nível de abstração no momento de definir critérios que delineiam a busca pelas trocas consideradas ótimas. As vantagens do uso da linguagem são validadas pela apresentação de vários exemplos de critérios de forma simples e idiomática, onde código mais característico do Python é introduzido sem confundir a semântica ao nível do domínio.

Aos meus pais.

Acknowledgements

I would like to thank my supervisors Pedro Vasconcelos and João Pedro Pedroso for all the help and support throughout these years. You have never given up on me and I am infinitely grateful for that.

Gostaria de agradecer aos meus pais, à minha irmã e cunhado Semyon, por serem um apoio incondicional neste meu caminho. É muito por vocês que vale a pena finalmente encerrar este capítulo.

To Tiago Campos, my partner and companion, thank you for this incredible year and for staying by my side through all the moments of hardship in such a devoted way. Your love, support, care and attitude inspire me to be better in every way and I could not be more excited for the new and loving memories to come after this chapter.

To my family of friends who have been fundamental in this path. In no specific order, I want to thank Ana Tarrinho, Rehana Jassat (and her family), Carina Fernandes, Marina Meira, Fabíola Fernandes, José Belo, Rita Vintém, Pedro Pereira, Sara Jesus, my goddaughter Alice, Juliana Braga, Joana Santos, Rui da Silva, Victor Hugo Ramos, João Agostinho, Cláudia Abreu, Nilufar Neyestani, Inês Soares, Filipe Pereira, Sara Pereira, Maria José Costa, Sílvia Ramos, Marisa Reis, Daniel Carvalho, Joaquim Martins, Tiago Azevedo, Joaquin Pereira, Luis Gabriel Pereira and Teresa Forte for all the love you gave me in this hard, long process. I would thank you and more individually if I could, for you have showed me support in very special, simple and unique ways. I believe you know why in particular.

To Ricardo Bessa, thank you for giving me the opportunity to work at INESC TEC, where I was able to be in contact with a research environment that enabled me to get important help on finishing this dissertation.

To researchers Ana Viana, Xenia Klimentova and Nicolau Santos from INESC TEC, thank you for your precious help and insights.

I thank all my friends and colleagues at INESC TEC who, in our everyday interaction encouraged me, cared about my progress and helped me with the content and format of this dissertation.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Kidney Exchange Programs

Patients whose renal function is severely diminished from damaged kidneys have to resort to dialysis or a kidney transplant. Globally, around 3.8 million people suffer from End Stage Renal Disease (ESRD)[ERC], when the kidneys cannot perform their normal function on their own. While dialysis is an exhausting therapy that may require several sessions per week, it may be the only option for those whose kidney condition has worsen beyond eligibility for a transplant.

Renal transplant is the most performed transplant in the world, but still the demand for organs is far too high. Some patients wait several years for an available, compatible kidney while others, having been considered fit for the procedure, end up waiting excessive time and also become too ill to be transplanted.

This method for improving kidney functionality and people's life quality began in the form of deceased organ donation. It eventually evolved to allow for living kidney donation, where a person with two healthy kidneys is willing to donate one of them. In most cases, there is an emotional relationship between the donor and the intended recipient of the organ, and it is said the donation is directed. But due to blood type and tissue incompatibilities, it is not always possible to perform the transplant. Patients suffering from ESRD still had no option but to keep waiting for an available, compatible kidney.

Around forty years ago [Rap86], a new framework for kidney donation arose from the observance of such situations of incompatibility. It would be possible for two pairs of incompatible donors and patients to exchange kidneys when the donor from the first pair is compatible with patient of the second pair, and vice-versa. A simple but initially very controversial setting that is today known as kidney paired donation or kidney exchange. At the risk of taking the emotional aspect of donation and leading an otherwise willing donor to withdraw such intention, there was a potential gain in kidney transplants that could save more lives. Patients were not buying organs (such practice is illegal in most countries) but would engage in barter, where the goods to be traded were kidneys.

Barter is the trading of goods or services where no monetary compensation is involved, as in the case of kidney exchange. The success of the practice led to the formation of organized programs at the local, regional, national and eventually at the international level. These programs can be seen as barter markets, where the agents are incompatible pairs of patients and donors who are seeking to meet their needs with the best available and suiting item, a kidney. Finding the exchange scheme that brings the highest level of satisfaction for the majority of the agents is known as market clearing. Barter is one of the oldest forms of trade and is still common nowadays. Especially with the development of the Internet and the World Wide Web, we can find a multitude of platforms that enable barter exchanges. Examples of goods and services in barter platforms are accommodation [Cou; Hom], books [Swa] or clothing [Reh].

As these markets grow in size and complexity, the job of handling all the agents' demands becomes arduous. Computerized systems offer the possibility of centralizing information, mediation and optimization of the output of market operation. Kidney exchange is no exception.

Some programs now involve hundreds of pairs, and since organ donation is strongly regulated, in order to guarantee fairness and equality conditions, it is important to develop a framework where every patient might see that needs are met in an equitable way. In that sense, a lot of research stemmed from the kidney exchange experience, and new mathematical models for optimizing market clearing have arisen. Real-world needs for program management and optimal exchange exploration led to the development of tools directed at the problem in question.

1.2 Objectives of this dissertation

Our proposal is to develop a domain-specific language embedded in Python (a general-purpose language) for handling kidney exchange instances. The challenge is dual: knowledge on language design must be paired with sufficient understanding of the domain in order to produce a tool that is useful enough for a domain expert without too much programming experience to recognize, analyze and even develop their own applications in the language. Our main goal was to leverage the Python architecture to implement a series of definitions and methods that, while not escaping the limits of the host language, will be integrated in a way where it actually gains from having general-purpose features available. The discussion of the details of the implementation attests the adequacy of host language and the examples shown in this work show how the proposed language and Python interact and make for very readable, domain-friendly code. We expect that our solution helps making the management, simulation and testing of optimization schemes and kidney exchange programs easier, both in research and real-world perspectives.

1.3 Outline of the dissertation

The rest of this dissertation is organized in the following manner:

- **Chapter 2 - Background**: gives the reader insight in the field of Kidney Paired Donation, presenting its progress in recent history, modalities in use, and the mathematical formulations underlying the computational approaches to find virtually optimal exchanges in the population available for transplantation. The chapter also includes information about domain-specific languages: the motivation for their use, examples of developed languages and the advantages on embedding their implementation in a general-purpose language.
- **Chapter 3 - EDSL for Kidney Paired Donation**: In this chapter we detail our proposed embedded domain-specific language (EDSL), discussing some of the implementation details and explaining some of the usage conventions.
- **Chapter 4 - Examples and operational features**: An illustration of how to use the language constructs, as well as proposals for the implementation of various kidney exchange optimization

schemes as validation of the language's abstraction capabilities.

- **Chapter 5 - Conclusions and Future Work**: A summary of the dissertation, where the final remarks are presented. The difficulties, challenges and opportunities for future development derived from the work are also discussed.

Chapter 2

Background

In this chapter, we present the literature pertaining the subjects underlying this dissertation. Implementing a domain-specific language implies owning a reasonably good knowledge base on the specific domain we are trying to model, and so it is important to know what are the main concepts, historical context and technical aspects of the problem at hand. Also, it is important to have a good understanding of the context of software and language development, in order to provide the implementation with good enough abstractions (in the domain) and good managing tools (for output production). The first section of this chapter introduces the relatively young but already very broad subject of Kidney Paired Donation, while the second provides context on domain-specific languages, their different modalities and forms of implementation.

2.1 Kidney Paired Donation

2.1.1 History and development

The idea of kidney swap between unrelated pairs of donor/patients was first suggested by Rapaport in 1986 [Rap86], as a strategy to help patients with willing related donors get a kidney from another pair with whom they were reciprocally compatible (initially for patient/donor blood types A-B and B-A). This idea came to fruition for the first time in South Korea, in the year 1991, when the first kidney exchange between two incompatible but cross compatible pairs was performed successfully. A few other exchanges took place in the following years (by 1995, Korea was also experimenting with simultaneous paired donations involving 3 to 4 incompatible pairs). It was not until the end of the decade, in 1999, that the first ever kidney paired exchange took place in European soil, more specifically in Switzerland [Thi+01]. The year after, the US was also having its first ever kidney exchange [ATM04].

Organized Kidney Exchange Programs

The success of the first kidney exchanges led to the development of organized programs to enroll more and more incompatible pairs with hopes of increasing the likelihood of finding a compatible organ. The bigger the pool, the greater the possibility of finding a matching donor for a patient in need and also the greater the possibility of finding a better match, while reducing waiting times for transplantation and improving the survival rates of those suffering from kidney failure.

The first programs started at a local level, mostly on a single center. Again, in South Korea, in 1995, the first organized KEP program was established at Yonsei University College of Medicine in Seoul [Huh+08]. In Europe, Romania had in 2001 two kidney exchange programs that performed 56 kidney exchanges over the course of 5 years, reporting similar graft survival rates to direct, emotionally related living donor transplants [Luc07]. India began performing KPD in the year of 2000, although only 34 transplants had been carried out in the first 9 years, being that, until 2005, the program operated only in one state [Mod+10]. Australia had also a first experience with KEP programs using a regional registry, resulting in nine kidney transplants in a year [FWC09].

With time, several countries developed KEP programs on a national level. In fact, while local, single center instances demonstrated the potential and viability of such programs, they are inherently characterized by smaller pools of incompatible pairs, and so the likeliness of finding a mutually compatible pair remained impaired. Smaller pools can also have little impact on patients who are harder to match (due to blood type or human leukocyte antigen composition, as we shall address later). By creating a framework for national operation, countries allowed the creation of greater pools, leading to more and better transplants, but also brought other challenges, on the logistical and technical levels. South Korea was again the pioneer in developing a nationwide kidney exchange registry, in 2001 [Mie+13]. The Dutch Kidney Exchange Program was the first European national program to be established, in 2004. Since then, a multitude of national kidney exchange registries have risen, each with their own specifications and logistical challenges. An example is the UK Living Kidney Sharing Scheme (UKLKSS, formed in 2006, having its first exchange performed in 2007) [UK]. Around the world, other KEP programs formed such as Canada's Living Donor Paired Exchange (LDPE, which started as a three-province program in 2009 and was extended to the entire country in 2010) [Can] and the Australian Paired Kidney Exchange Program (AKX, established nationwide in 2010).

In the United States, although there is still no unified national program, there are several programs across the country. Some are very local, geographically concentrated, while others span across multiple transplant centers and have performed matching runs on a national level. The three main programs operating are the National Kidney Registry, the Alliance for Paired Donation and the UNOS Kidney Paired Donation Pilot Program. Each of them are composed by substantial pools that overlap in some cases. So although there is no central registry for paired donation, the existing programs and specialized centers, combined, reach national coverage [Ell14].

On the international level, there is interesting emerging cooperation to be mentioned. In Europe, a considerable group of nations have already kicked off their national KPD program (see Figure 2.1). The experiences at the country level are leading to new considerations on the need and viability for a larger, transnational programs. An initiative called the European Network for Collaboration on Kidney Exchange Programs (ENCKEP), funded by the Cooperation on Science and Technology Association (COST) [Eur] furthers investigation and cooperation between the various KEP programs in order to assess what are the best practices in paired donation, the underlying difficulties and opportunities for growth arising from each country's particular circumstances. The goal is to promote the implementation of KEP in countries which still do not have one, and bring existing ones to build common platforms and protocols for joint operation, while accumulating and harmonizing knowledge on how to improve the performance of the programs. It is important to note the COST geographical reach extends that of the European Union to include other European countries and even some nations outside of the continent [Bir+19a]. Nowadays, experiences with international cooperation and programs include instances such as Austria and the Czech Republic, who have joined their national pools; the

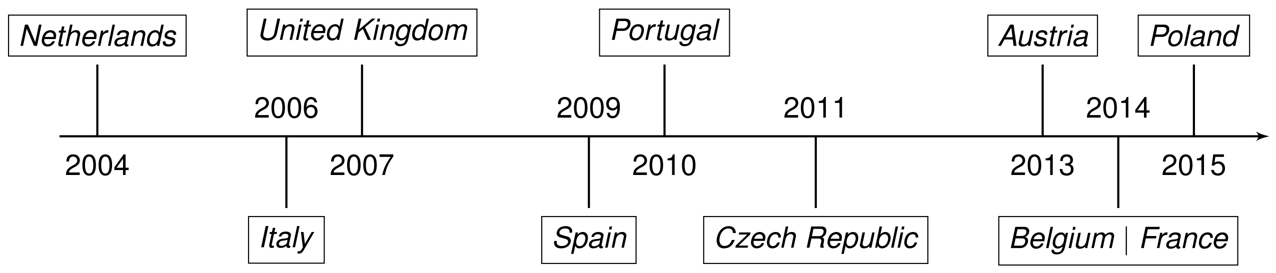


Figure 2.1: Timeline for the implementation of National KEP in Europe.

UK welcoming pairs and altruistic donors from Ireland; France and Switzerland signed a cooperation protocol, although no transnational transplants occurred as a result yet; Scandiatransplant started the process of joining Denmark, Norway and Sweden in a common KEP [Bir+19a]. Also, in the present time, Australia and New Zealand have united in a common paired exchange program [ANZ].

Kidney Exchange Program in Portugal

In Portugal, the National Kidney Exchange Program, Programa Nacional de Doação Renal Cruzada [PND], started in 2010. However, no exchanges were performed until 2013 [12; 13]. The program is maintained by Instituto Português do Sangue e da Transplantação, who reports, since 2015, four 3-way exchanges, (one in 2015, another in 2017 and two in 2018) across different transplant centers [IPS]. In [Bir+17], there is also mention of three 2-way exchanges and one 3-way exchange until the end of 2016, with a total of 50 registered pairs. In total, out of the 8 transplant centers in the country, 5 are participating in the program, as well as 3 histocompatibility and immunogenetics laboratories. Since the summer of 2018, Portugal has been in cooperation with Italy and Spain, in which after running the match search in each national pool, the three countries perform a joint matching run to try and find matches for the remaining pairs in their respective registry [Bir+19b].

2.1.2 Conditions for Kidney Exchange

Kidney exchange (in fact, any kind of donation) can only happen if certain conditions are met. These conditions determine the compatibility between a donor and a patient and are mainly related to blood type and tissue compatibility. By performing clinical tests, a donor and a patient are considered incompatible if the patient carries antibodies to any particular antigen in the donor's system. Antigens are molecules that trigger immunological responses in the body, activating the antibodies that attack them. In that case, insisting in performing transplantation would result in the patient's body rejecting the new organ, leading to even greater complications.

Blood type compatibilty (ABO)

Pertaining to blood type, the standard form of assessing compatibility is using the ABO system. Human blood can be categorized under four types: O, A, B and AB. These indicate the presence or absence of antigens A and B, which means that type O blood has none of those antigens and AB refers to a person carrying both antigens in their blood. If an organism does not possess a certain antigen, it will develop antibodies against it. If a donor has an antigen that the patient does not have, chances are that the organ will be rejected by the patient's body. For this reason, type O donors are theoretically able to donate to any blood type, while AB-type donors can donate only to other AB-type patients. Donors with type A or B can only donate to the same blood type or to patients with AB blood type. [Ame19]

Human Leukocyte Antigen (HLA) compatibility

Another important form of testing for compatibility is by the presence of specific antigens. Human Leukocyte Antigens are proteins present at the surface of white blood cells and other human tissues. These antigens are generally grouped into a small number of categories (HLA-A, HLA-B, HLA-DR, HLA-DQ are the most common). To check for compatibility between a patient and a donor, a test known as a crossmatch is performed. In it, a sample of serum taken from both subjects is mixed to test for the presence of antibodies in one organism against the antigens present in the other. If that is the case, there is said to be a positive crossmatch and those two people are rendered incompatible [Uni98].

Depending on the composition of their tissues, a person can be more or less sensitive to transplantation. If they have antibodies against a large number of antigens, they will be harder to match than a patient who has less antibodies. The percentage of donors against whom they will most likely be crossmatch positive is given by the percentage of panel reactive antibodies (PRA), and it is used to classify the sensitivity to another person's blood. Generally, if the PRA is below 20%, we say that the patient has a low level of sensitization. If it is over 80%, they are said to be highly sensitized [Uni98].

Confirming compatibility

Matching runs define the best possible exchange scheme. Especially with larger pools, most of the times, compatibility is established via computational tests. Having information about the patient and the donor's blood type and HLA antigen and antibody composition, it is possible to perform a virtual crossmatch, which might not always correspond to true compatibility. For instance, the patient might have been subject to a blood transfusion, which might temper with their constitution. In order to assure that there is still a high chance of transplant success, a final test is run before the procedure. A sample of the donor's blood and the patient's plasma are combined for a final check. In case there is no agglutination (no antigens are joined with their antibodies, creating larger masses in the mix), then the pair is considered to be truly compatible. [Cho07]

2.1.3 Modalities of Living Donor Exchanges

We now present different schemes for kidney exchange that have developed over time, as experiences and circumstantial needs pushed for new frameworks and resulted in more opportunities for patients to receive a kidney.

Note that, in each of the example figures, a dashed line means incompatibility, while a full line signals compatibility. A dashed arrow means that the transplant occurs in the presented scheme.

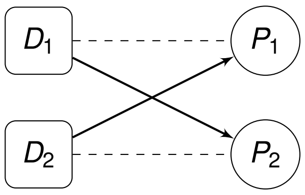


Figure 2.2: A 2-way exchange. Donor D_1 gives a kidney to patient P_2 and donor D_2 donates to patient P_1.

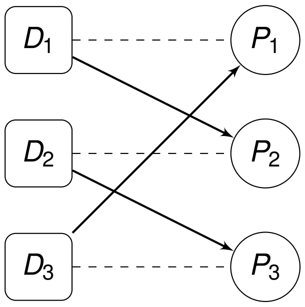


Figure 2.3: A 3-way exchange.

2-way exchange

This is the simplest form of exchange between pairs of incompatible patients and donors (Figure 2.2). In this setting, the donor of one pairs donates his or her kidney to the patient of the other pair and vice versa. Concerns about any of the participants dropping out before the two transplants are performed lead to the requirement that the exchange be made simultaneously.

3-way exchange

More than 2 pairs can be involved in an exchange. In order to alleviate the need for reciprocal compatibility, an extra pair can be introduced, which does not have to be mutually compatible with the first two pairs. In the example seen in Figure 2.3, donor D_1 is compatible with patient P_2, donor D_2 is compatible with P_3, and donor D_3 is compatible with patient P_1, forming a cycle, or closed chain, in which compatibility does not need to be mutual between all pairs. Ideally though, this would be the case, as if, for instance, one of the pairs is unable to participate in the exchange (due to, for instance, incompatibility problems arising from last minute clinical tests), there is still a chance for a 2-way exchange. These types of 3-way cycles have been referred to as effective 2-way exchanges [MO12].

n-way exchange

This concept is a generalization of 3-way exchanges to allow for an arbitrarily long cycle (Figure 2.4). In practice, for most cases, exchanges do not exceed the 3 or 4 kidney paired donations, although there have been reports of longer cycles (for instance, of 6 and 7 transplants in the Czech Republic [Bir+19a]). There are a few reasons that justify keeping cycles relatively short: first, since exchanges are required to happen at the same time, there are obvious logistical constraints, in terms of the number of operating rooms, surgeons, nurses and other medical staff, which multiply with each transplant; second, having long cycles, especially those without embedded shorter cycles, represents a risk in the sense that it is more likely that someone along the chain will drop out or that, as mentioned

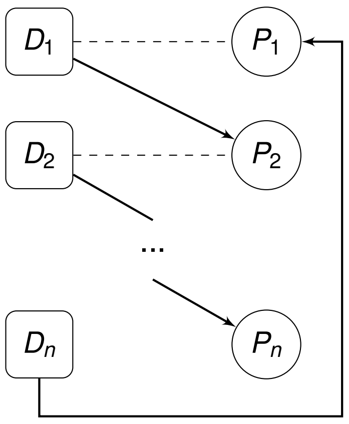


Figure 2.4: A n-way exchange.

before, new clinical tests render any one of the transplants to be unfeasible, or even if a patient becomes too ill for receiving a transplant.

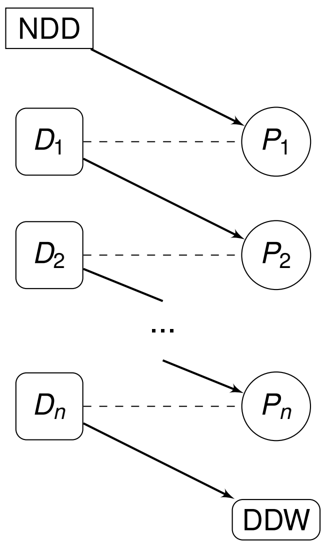
Non-directed (altruistic) donation

A different scenario from all of the listed above involves non-directed donors (NDD), also known as altruistic, good samaritan or unspecified donors, who are willing to donate a kidney to a patient in need, not being associated to any one in particular. Through them, a chain of transplants is initiated, and different variants have been used. One of the variants is the Domino-Paired Donation (DPD, Figure 2.5a), where the altruistic donor, instead of donating to the deceased donor waitlist (DDW), initiates that chain by donating to a patient in the KEP pool. The associated incompatible donor gives their kidney to another patient, and the process goes on until eventually a number of patients receives a kidney and the donor paired with the last patient donates to the deceased donor waitlist. The other variant involves what is known as bridge donation: donors who may later initiate themselves a chain, even though they are associated with a patient. The procedure initiates with an altruistic donor, the chained donations are performed until the patient in the last served pair gets their kidney. Their associated donor, the bridge donor, is available to initiate a chain later in time. This is known as a non-simultaneous extended altruistic donation (NEAD, Figure 2.5b).

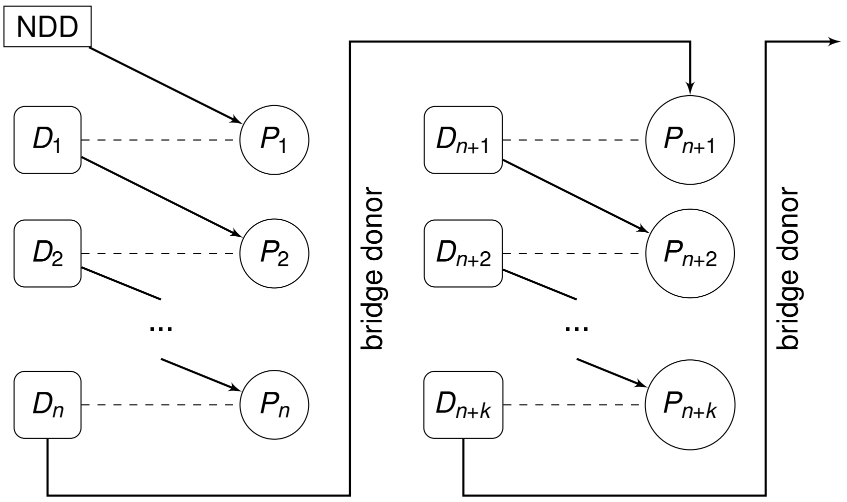
Chains have the advantage that, because it is not a closed circuit, donor withdrawal is less harmful than in a cycle. In particular, the rule of simultaneity can be relaxed. This is a natural aspect of NEAD, considering that one chain leads to a future one, and that one is the cause of a new chain, thanks to bridge donors. That is why NEAD is also known as Never-Ending Altruistic Donor chain. In 2018, a piece of news reported that a chain initiated in this fashion led to more than 100 donations, and counting [Coo18]. Nowadays, around two thirds of all the transplants performed via the Kidney Paired Donation Program in Canada are the result of chains initiated by good samaritans [Bac].

Inclusion of compatible pairs

An alternative to using incompatible donor/patient pairs or unspecified donors is admitting pairs in the pool that are compatible [Gen+07]). The idea behind this is that compatible pairs may bring donors which are in higher demand, while the patient in the compatible pair also has the chance of finding a kidney of better quality (for instance, with stronger compatibility, or a younger donor, a healthier



(a) Domino-Paired Donation (DPD).



(b) Non-simultaneous Extended Altruistic Donation (NEAD).

Figure 2.5: Altruistic donor chains.

kidney). By increasing the number of kidneys in the pool, there is a higher chance of forming new cycles or chains, while also allowing for better quality exchanges.

2.1.4 Market clearing: Models for Kidney Exchange

In the context of kidney paired donation programs, market clearing means finding an optimal exchange scheme that maximizes the gain at the collective level, where a typical goal is to provide the greatest number of transplants to patients in need of them. With the establishment of centralized kidney exchange programs, and the growing size of pools of incompatible donor/patient pairs waiting for a match, more and more research has been dedicated to finding strategies, especially computational, that allow to explore the space of possible exchanges. These approaches use optimization like Integer Programming (IP) to find an optimal exchange or exchanges to be considered. Since there are several different aspects and technicalities involved, a more formal definition of a model for kidney exchanges will help to grasp the nature and the complexity of the task at hand. This subsection provides the theoretical framework by listing the necessary definitions, notations and algorithms that were the base of the proposed implementation, which are aligned with the formalization given in [Glo14].

Definition 2.1.1. *A pool $N = N_P \cup N_U$ consists of pairs of incompatible donor and patients (N_P) and altruistic donors (N_U) waiting to be matched in a kidney exchange program.*

Virtual compatibility (typically, ABO compatibility and HLA crossmatch testing) can be represented using a directed graph, in which that compatibility is made explicit.

Definition 2.1.2. *A kidney exchange pool can be represented as a directed graph $G = (V, A)$, where V is the set of either incompatible donor/patient pairs or altruistic donors and A represents the arcs that signal compatibility between each agent, i.e., for vertices $i, j \in V$, if $(i, j) \in A$, donor in pair i can give the kidney to patient in pair j.*

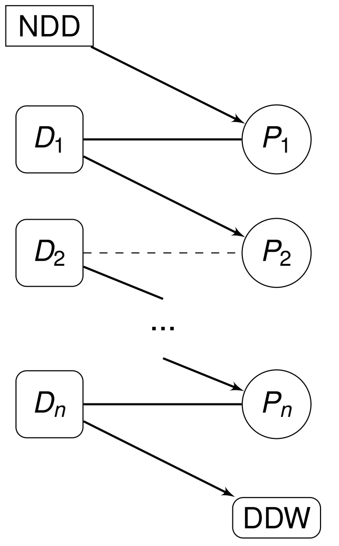


Figure 2.6: An example of a chain where the first and last pairs involved are compatible.

When two vertices are connected in both directions, those two pairs' correspondent donors and patients can exchange kidneys. Altruistic donors, because they have no associated patient, have no incoming arcs. In the simplest setting, there would be a 2-way exchange between two vertices connected directly. There may be an indirect connection between two vertices, through a path, which can be defined as follows.

Definition 2.1.3. *A sequence $\langle n_1, n_2, \dots, n_k \rangle$ is a path if $\{n_1, n_2, \dots, n_k\} \subseteq N$ and for every $1 < i \leq k$, $(n_{i-1}, n_i) \in A$.*

This simple definition allows for establishing what cycles and chains of kidney donations are in this representation.

Definition 2.1.4. *A chain of length k is a path $\langle n_0, n_1, n_2, \dots, n_k \rangle$ where $n_0 \in N_U$ and, for $1 \leq i \leq k$, $n_i \in N_P$.*

Definition 2.1.5. *A cycle of length k is a path $\langle n_1, n_2, \dots, n_k \rangle$ where for each $1 \leq i \leq k$, $n_i \in N_P$ and $(n_k, n_1) \in A$.*

The length of a chain or a cycle defines on the number of transplants they imply, and that is why a chain of length k has $k + 1$ nodes and a cycle of length k has k nodes.

As discussed previously, due to practical questions, cycles and chains are usually limited in length. Hence, it is natural to define the following set:

Definition 2.1.6. *In a graph for a kidney exchange pool G, the set of all cycles with length at most $k \in \mathbb{N}$ and chains with length at most $k' \in \mathbb{N}$ is defined as:*

$$C(k, k') = \{c\text{: } c \text{ is a cycle of length at most } k \text{ or a chain of length at most } k' \text{ in } G\}$$

The goal, as stated previously, is to find an optimal scenario where the greatest improvement, or satisfaction, is attained for the agents in the market. In the case of kidney exchange, that can be done through a combination of cycles and chains. The next concepts follows naturally.

Definition 2.1.7. *An exchange M is a set of cycles and chains $\{c_1, c_2, \dots, c_n\}$ such that, for every $1 \leq m \leq n$, $c_m \in C(k, k')$, with $k, k' \in \mathbb{N}$ and for every $c_i, c_j \in M$, c_i and c_j are disjoint, i.e., $c_i \cap c_j = \emptyset$.*

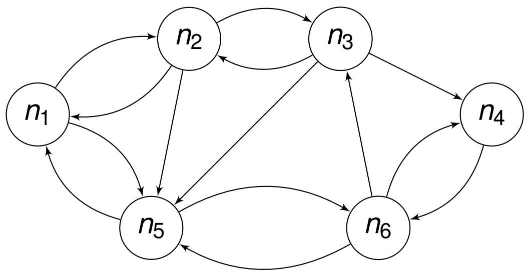


Figure 2.7: Kidney Exchange example graph.

Definition 2.1.8. *For a given kidney exchange graph G, the set of all subsets $M \subseteq C(k, k')$ such that M is an exchange is denoted by $\mathcal{M}$.*

Having defined the kidney exchange graph and the set of implicit cycles and chains, the optimal solution to the kidney exchange problem is a subset of all possible exchanges. Depending on the natural number k limiting the size of chained donations, we are looking for a subset of $\mathcal{M}$ of exchanges that clear the market according to selected criteria.

A feature such as the number of transplants can be generalized as a concept of ranking exchanges, and it is used to define those which maximize or minimize a particular criteria implied by that feature.

Definition 2.1.9. *For a given exchange set $\mathcal{M}$, a criterion is a function $f : \mathcal{M} \to \mathbb{R}$.*

With f, we can clear a problem using one particular criteria. However, we may wish to find the exchanges that optimize multiple criteria, called the *hierarchical multi-criteria clearing problem*.

Definition 2.1.10. *For a given exchange set $\mathcal{M}$ and set of criteria $\mathcal{I} = \{f_1, f_2, \dots, f_n\}$, the hierarchical multi-criteria clearing problem is to find the subset $\mathcal{M}^* \subseteq \mathcal{M}$ with exchanges M^* such that, for each $i = 1, \dots, n$, $M^* \in \mathcal{M}_i$ and each $\mathcal{M}_i$ is recursively defined as $\mathcal{M}_i := \{M \in \mathcal{M}_{i-1} : f_i(M) \geq f_i(M'), \forall M' \in \mathcal{M}_{i-1}\}$ with $\mathcal{M}_0 = \mathcal{M}$.*

Figure 2.7 illustrates an example of graph for a kidney exchange pool. If we set $k = 3$, it is possible to identify five 2-way exchanges: $c_1 = \langle n_1, n_2 \rangle$, $c_2 = \langle n_2, n_3 \rangle$, $c_3 = \langle n_1, n_5 \rangle$, $c_4 = \langle n_4, n_6 \rangle$, $c_5 = \langle n_5, n_6 \rangle$ and three 3-way cycles: $c_6 = \langle n_1, n_2, n_5 \rangle$, $c_7 = \langle n_3, n_4, n_6 \rangle$ and $c_8 = \langle n_3, n_5, n_6 \rangle$. If, however, we let $k = 4$, than we can identify more cycles, namely two 4-way cycles $c_9 = \langle n_1, n_2, n_3, n_5 \rangle$ and $c_{10} = \langle n_2, n_5, n_6, n_3 \rangle$. All of these cycles belong to the exchange set $\mathcal{M}$, as do their disjoint combinations. For example, the set $M_1 = \{c_1, c_7\}$ is a possible exchange, as is $M_2 = \{c_4, c_6\}$. When considering the number of transplants, an exchange like $M_3 = \{c_4, c_9\}$ would be considered optimal, as it serves the most patients in the pool (in this case, all pairs get and donate a kidney). By relaxing (i.e., increasing) the parameter k of the maximum length of all cycles and chains, a single cycle $c_{11} = \langle n_1, n_2, n_3, n_4, n_5, n_6 \rangle$ would clear the market entirely. Nonetheless, in a real setting, it is usually preferable to keep cycles as short as possible (due to already discussed considerations like logistics or the possibility of failure or dropout in longer cycles), so an exchange like $M_4 = \{c_2, c_3, c_4\}$ would probably be preferred, as it is composed only by 2-way exchanges and thus only implies two simultaneous transplants, i.e., four simultaneous operations.

The next subsection builds on these formulations to present strategies that enable the search for optimal exchanges.

Integer Programming Formulations

The problem of finding the optimal set of kidney exchanges can be described using optimization programs such as Integer Programs (IP). Although there are different formulations for the problem, we present the one that was used as basis for our implementation. It is the so called cycle formulation, where the objective function consists of a linear combination of all possible cycles found in the pool.

Cycle formulation

The cycle formulation was introduced by [ABS07] and is based on the cycles of the pool, where each one corresponds to a binary decision variable x such that, for every $c \in C(k, k')$:

$$x_c = \begin{cases} 1 & \text{if } c \text{ is included in the exchange set} \\ 0 & \text{otherwise} \end{cases}$$

The program is defined as follows:

$$\text{Maximize} \qquad z(x) = \sum_{c \in C(k,k')} w_c \, x_c \tag{2.1.4.1}$$

$$\text{Subject to} \qquad \sum_{c \in C(k,k') : n \in c} x_c \leq 1 \qquad \forall n \in N \tag{2.1.4.2}$$

$$x_c \in \{0, 1\} \qquad \forall c \in C(k, k') \tag{2.1.4.3}$$

In this formulation, 2.1.4.1 sets the objective function as a linear combination of variables x_c weighted by their correspondent coefficients, or weights, w_c (which are usually set as the length $|c|$ of the cycle/chain, but kept as general as possible to serve for other criteria of optimization). The set of constraints 2.1.4.2 ensure that the same pair is not present more than once in the exchange set (i.e., each pair can only appear in one selected cycle). Conditions in 2.1.4.3 set the variables as binary. Although the problem is described as a maximization program, it can easily be converted to a minimization program if we multiply each of the coefficients of the variables by -1.

This formulation is concise but can become computationally heavy. The number of cycles grows exponentially as the number of nodes increases, as well as when we increase the length of allowed cycles and chains

The next subsection presents a formulation to model problems that require multiple criteria to be considered, keeping in line with the already defined multi-criteria hierarchical clearing problem.

Multi-criteria Formulation

Some of the already established National Kidney Exchange Programs set multiple criteria that implicitly define the best possible exchanges. In order to achieve this, we recursively solve the problem regarding the first criteria, then propagate the optimal value of the objective function to the problem defined by the following criteria in the form of an additional constraint, and so on until the final criteria produces the optimal exchange set. Formally, we can describe this as iterating through a set of single objective problems using the cycle formulation as basis, with m being the number of applied criteria:

$$\begin{aligned}
&\text{Maximize} && z_i(x) = \sum_{c \in C(k,k')} w_c^{(i)} x_c && i = 1, \dots, m && (2.1.4.4)\\
&\text{Subject to} && \sum_{c \in C(k,k'): n \in c} x_c \leq 1 && \forall n \in N && (2.1.4.5)\\
& && z_j(x) \geq z_j^* && j = 1, \dots, i-1 && (2.1.4.6)\\
& && x_c \in \{0,1\} && \forall c \in C(k,k') && (2.1.4.7)
\end{aligned}$$

This approach is referred to as "hierarchical programming". As can be noted, this formulation is very similar to the cycle formulation. In 2.1.4.4, the weights are now indexed by i to refer to the problem under which they are assigned, and this definition is identical to 2.1.4.1. When $i = 1$, constraints 2.1.4.6 are not active. 2.1.4.5, again, limits each pair or unspecified donor to appear only once in the exchange (as in 2.1.4.2) and 2.1.4.7 constrain each variable x_c to be binary, as does 2.1.4.3. When i is greater than 1, 2.1.4.6 become active constraints in the corresponding problem. They ensure that each problem takes into account the optimal value (represented by z_j^*) found by all the previous objective functions. This is analogous to the condition that defines each of the exchange sets $\mathcal{M}_i$ in Definition 2.1.10. In that same definition, we can think of each of the criterion functions as corresponding to each of these problems defined sequentially.

Generalizing iterative formulations

The idea of recursively finding optimal values that are propagated to following programs is not exclusive of the presence of distinct criteria. For example, in [Mon+19], a comparative study of matching algorithms regarding waiting times describes a process that is related to one specific criteria (that is, maximizing the number of patients that have been waiting the longest) which can be solved by an iterative formulation which is, formally, similar to the one used for multi-criteria clearing problems. Considering waiting times, we first maximize the number of patients that have waited the longest observed waiting time in the pool, than maximize the number of patients waiting for the second greatest observed waiting time, and so on until we reach of exchanges where the number of long-waiting patients is optimal. This same search can be applied to other attributes of the pool such as patients' age, time in dialysis or a sequence of blood types.

Generalizing this routine we have, for a given set of values $t_1, t_2, \dots, t_n$ a sequence of problems that are solved by maximizing the number of exchanges where each value is observed. In practice, we can think of this approach as a version of the multi-criteria problem by considering that maximizing each of the listed values can correspond to a single, independent criteria. The objective function and associated constraints are the same as the ones previously presented.

Complexity of the problem and other formulations

As commented on the cycle formulation's subsection, increasing the number of nodes, or pairs/unspecified donors, as well as the limit on the length of cycles and chains can result in an exponential growth of the size of the graph and the number of cycles and chains, which in turn creates an exchange set that is intractable to enumerate. In [ABS07], the authors present statistics on instances of kidney exchange pools generated from real data from the United Network for Organ Sharing (UNOS)

Nodes	**Arcs**	**Cycles ≤ 4**	**Chains ≤ 4**	**Chains ≤ 6**
10	50	0	1.0e+1	5.4e+1
20	192	8.00 e+1	2.21e+2	1.34e+2
50	1087	2.04e+2	9.87e+3	2.51e+5
100	4443	5.07e+3	1.84e+5	2.44e+7
200	16412	1.00e+5	2.23e+6	1.34e+9
500	99501	8.58e+6	1.02e+8	5.83e+11

Table 2.1: Average number of cycles and chains sampled from the Dutch national kidney exchange program. Source: [Glo14].[1]

[Uni19], which show that, for a pool of 500 patients, an average of less than half a million cycles of length 2 and 3 are to be expected. The scale rises to a million when the number of patients doubles. Considering that each cycle corresponds to a variable in our programs, a great computational effort is expected in order to retrieve optimal exchanges. Increasing the length limit of the chains and cycles involved to 4 is enough to grasp the exponential progress of the number of variables, with mean values of close to 9 million for cycles and in the hundred million for chains for a pool of 500 patients (see Table 2.1). [ABS07] and [Glo14] present proof that, for a limit length of 3 and above, inspecting the existence of a perfect cycle cover for a kidney exchange graph is a problem that is NP-complete.

These unavoidable issues render a simple formulation unfeasible for large instances, whether by the time it takes to enumerate the solutions or by the memory outage caused by generating them (for large enough instances, simply listing all possible cycles can exhaust available memory). The demand for more effective numerical approaches to kidney exchange arose naturally. While other simple formulations (like the edge formulation, based on the arcs of the exchange graph instead of the cycles, described in [ATM04]) suffered from the same problem as the cycle formulation (in this case, it is the number of constraints that grows exponentially, not the number of variables), the application of more elaborate strategies helped scale these programs to be able to deal with increasingly greater pools.

One of the strategies is the so called branch-and-price method [ABS07; Glo14; KAV14; Dic+16], which involves starting with a subset of the cycles calculated for the original kidney exchange graph and using the linear program (LP) relaxation (i.e., relaxing the integrality restriction on the variables) of the program to get (possibly) fractional upper bounds on the solution of the IP version. Whenever specific conditions (attributing a *price*, or *cost* to the cycles) show that adding new variables help improve the optimal value, we proceed with the increment. As long as the retrieved optimal value is fractional, the branching strategy restricts one or more variables back to integral values and repeat this process.

By using a smaller set of variables, we keep the problem simpler and quick to solve. This makes sense since, for example, in the cycle formulation, the objective function consists of a linear combination of all variables that correspond to cycles in the graph, but the exchanges are usually composed of very small subsets of all cycles. Column generation algorithms (which correspond to adding variables to the program) inspect the cost of those additions and its impact on the optimal value. Linear duality conditions are used to check the improvement of the incremented program in finding the optimal value of the original, complete problem.

Another strategy consists in new integer programs which are compact. A compact formulation means that the number of variables or constraints of a problem grow but are bounded by a polynomial, which keeps the programs from exhausting computational resources as quickly. In [Con+13], the authors propose two new formulations, the *edge assignment formulation* and *extended edge formulation*, and while the classical formulations (edge and cycle) present linear relaxations with tighter upper bounds on the optimal values, results from the article suggest that these compact formulations are more capable of handling larger instances, when considering a greater limit on the cycle and chain lengths or when the graphs have higher density. There is a trade-off between finding a tight bound, reaching optimality faster and being capable of computationally handle the problem. An important contribution was the proposal of adaptations of the programs to handle other variants of the problem, such as the inclusion of altruistic donors or compatible pairs.

Authors in [Dic+16] proposed three new formulations, two of which are compact: the *position-indexed edge formulation* and the *position-indexed chain-edge formulation*. While the former handles cycles only but maintains the tightness of the linear relaxation, the latter is capable of handling chains but still produces an exponential number of variables, namely the ones pertaining to cycles, and so the program is still sensible to the limit imposed on length. In order to tackle this issue, the authors present an implementation combining this program with the branch-and-price strategy. A final formulation, which is a combination of the first two, is also compact.

2.1.5 Related tools and software

The application of kidney exchange, whether in its incipient years or in the context of defined local and national programs, has mutually influenced the extensive research that aimed to discuss its moral complexities, their implications such as criteria that translates to better outcomes, while also improving implementation at the technical level. Also many, if not all of the established KEPs use computational tools in order to decide which are the best possible exchanges. In this subsection to provide a few examples of tools developed that enable research validation and also refer some solutions that implement kidney exchange schemes or allow to customize search runs, while providing some form of visualization.

Data models and simulators

As discussed previously, a lot of investigation is dedicated to improving the performance of market clearing algorithms. As a result, it is a common and necessary practice to present benchmarking tests that assert the claims made about new implementations. Credibility is added when the data used is, at least, sampled from real records. An example is Table 2.1, where the length of cycles and chains are averaged from multiple samples of pools based on real data from the Dutch program. But this is not always possible. Even in those cases, there is the idea of producing data that is based on a reasonably realistic model of a certain population. This way, it is possible to generate multiple instances of data with the goal of providing average performance indicators.

The data generated in [ABS07], presented in the previous sections, follows a processed described in [Sai+06], which aims to assess the effectiveness of kidney paired donation programs by means of simulations based on real data. The authors present statistics derived from that data that allows the generation of new instances of incompatible donor/patient pairs.

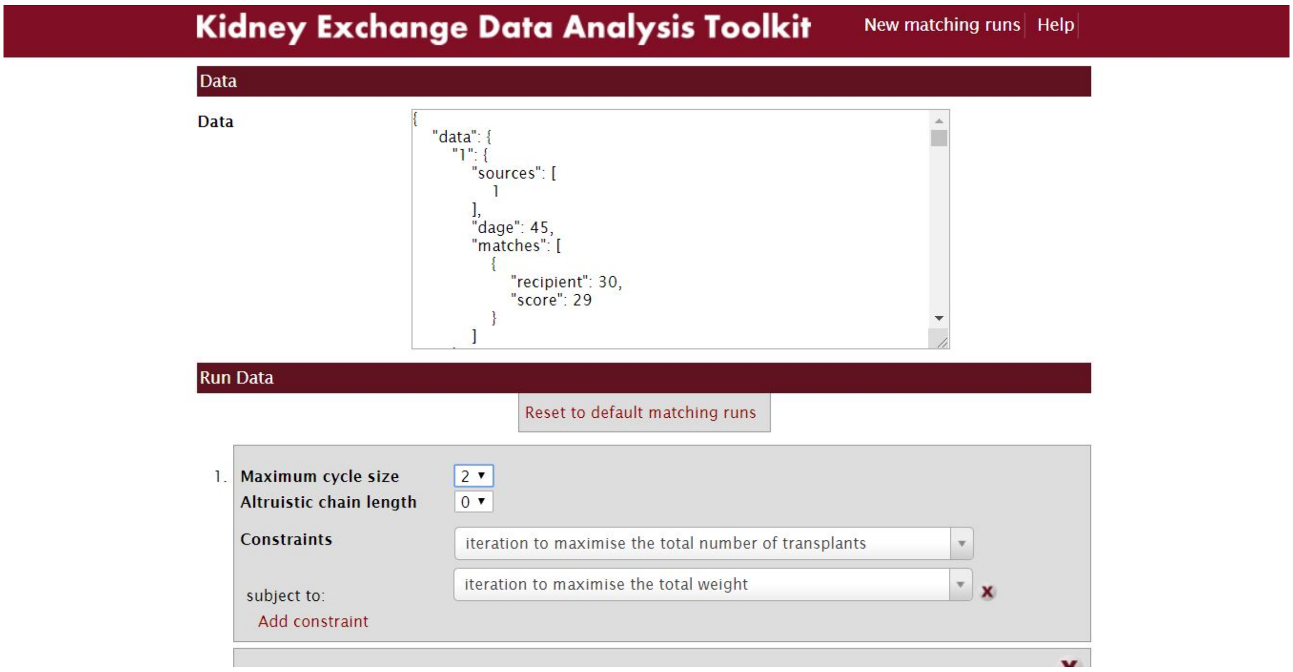
These simulations can be made parametric and formalized into specific tools. In [San+17], a simulator for both pool generation and optimization is presented. It is a system comprised of several modules that tackle each of the parts of the process of generating new data. It is highly customizable, allowing the user to choose general parameters such as different exchange modalities (include only incompatible pairs or allow the inclusion of compatible pairs and multiple donors, for instance), but also to input the characteristics of the participants of the pool, such as distribution of blood type and age or even the level of PRA. A dedicated PRA estimation module estimates PRA based on the input profile and uses it to form incompatibilities between generated couples. Finally, a pool generation module puts this all together to instantiate a virtual group of donors and patients in a KEP pool. Additional modules help manage the pool's changes over time (one of the main goals of the simulator is to study the behavior of different policies and exchange schemes as pairs are matched or become unsuitable for transplantation and eventually leave the pool) and perform the optimization itself.

Software for exploring and optimizing KEP instances

Although most of the software used in real kidney exchange programs is not public, we were still able to find some examples of applications that allow the input of a KEP pool and, to a certain degree of parameterization, enable optimization subject to desired constraints. In this subsection we present two examples.

The first one, found in [MO12], is an application dedicated to finding the optimal exchanges for the national UK kidney exchange program. In fact, the authors present two pieces of software: one, the Kidney Exchange Allocator, that implements the problem with the hierarchical criteria as defined by the National Health Service Blood and Transplant and a second one, the Kidney Exchange Data Analysis Toolkit, that enables the exploration of the impact of the same criteria. It allows the user to simultaneously test the impact of different combinations of criteria, in a simple but intuitive interface. Data describing the pool of registered pairs must be input in a JSON format and for each run there are options to define the limit on cycle and chain lengths independently, as well as the desired constraints, ordered to a preferred hierarchy. These tools come in the form of a web application, adding that the former also is available in a web API version. A screenshot of the data analysis toolkit is found in Figure 2.8.

The other example is a software for KEP management, optimization and visualization, called KPDGUI [Bra+19], and it is the most complete system that we found that was free to use. It allows the user to create a new pool entirely or upload a preexisting database of patients and donors, and then to edit that same pool either by changing the existing data or by adding or removing pairs and altruistic donors. Upon loading the information, a visualization of the pool and its virtual compatibility graph is instantly presented (see Figure 2.9a). Exploring the software, it is noted that the software is prepared to deal with different modalities for kidney exchange, like the inclusion of non-directed donors or the association of multiple donors to one patient. The "matching runs" are also very customizable, as the software allows to tune various parameters like prioritization of high PRA candidates, inclusion of compatible pairs, dedicate blood type O donors for blood type O patients, to name a few (see Figure 2.10a). Also, for more general parameters, the user can set the length limit for cycles and chains and set the number of exchanges returned; different optimization schemes are also included where, for instance, we can impose the existence of fallbacks in cycles and chains (possibility of smaller exchanges in case of failure, from 3-way exchange to a 2-way exchange, as an example).



Kidney Exchange Data Analysis Toolkit

Data

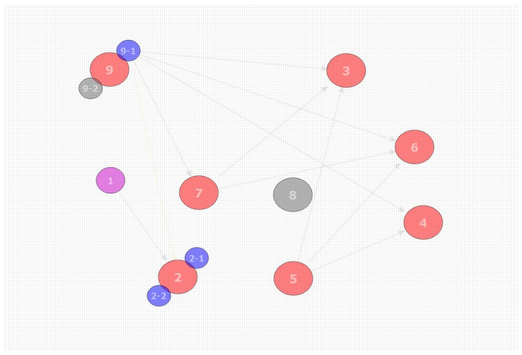
Data

```
{
   "data": {
      "1": {
         "sources": [
            1
         ],
         "dage": 45,
         "matches": [
            {
               "recipient": 30,
               "score": 29
            }
         ]
```

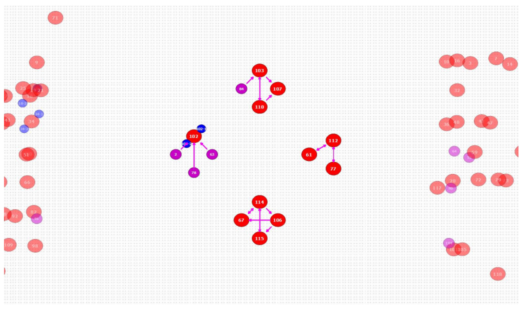
Run Data

1. Maximum cycle size 2
Altruistic chain length 0
Constraints iteration to maximise the total number of transplants
subject to: iteration to maximise the total weight

Figure 2.8: The interface of the Kidney Exchange Data Analysis Toolkit.



(a) Small KEP instance and its compatibility graph.



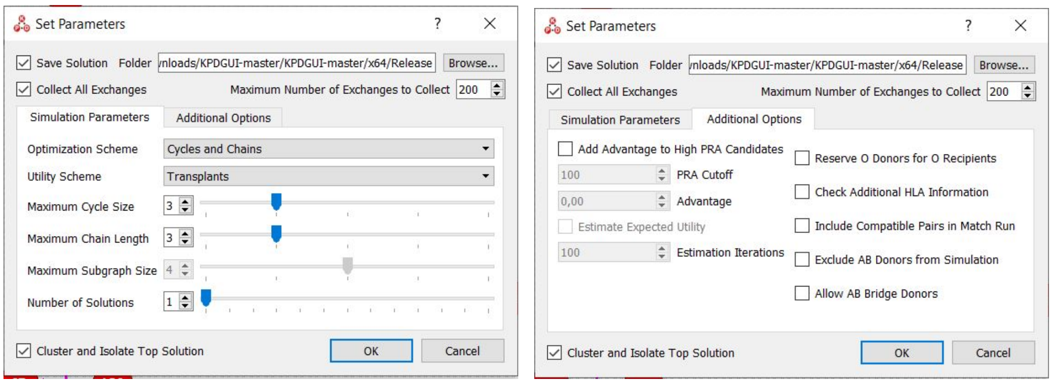
(b) Larger KEP instance after running optimization. The top solution is now represented in the graph.

Figure 2.9: Visualization examples in KPDGUI.

The software also lets the user choose different utility schemes, an option that also influences the objective of the optimization, by setting the goal on a number of goals like number of transplants, mid and long-term survival rates, and difficult transplants (see Figure 2.10b). Finally, after the matching is made, the graph's look changes in order to display which exchanges are part of the optimal setting (Figure 2.9b).

2.2 Domain-Specific Languages

A *Domain-Specific Language (DSL)* is a term describing any programming language designed to express, model and solve problems in a specialized domain. This set of problems, which motivate the development of a number of applications and for which that language constitutes the basis, is referred to as the *domain*. They contrast with General-Purpose Languages (GPL), whose domain is not specific and are capable of tackling many different domains using generic syntax and semantics.



Set Parameters

☑ Save Solution Folder ynloads/KPDGUI-master/KPDGUI-master/x64/Release Browse...
☑ Collect All Exchanges Maximum Number of Exchanges to Collect 200

Simulation Parameters | Additional Options

Optimization Scheme: Cycles and Chains
Utility Scheme: Transplants
Maximum Cycle Size: 3
Maximum Chain Length: 3
Maximum Subgraph Size: 4
Number of Solutions: 1

☑ Cluster and Isolate Top Solution OK Cancel

(a) General parameters for the optimization.

Set Parameters

☑ Save Solution Folder ynloads/KPDGUI-master/KPDGUI-master/x64/Release Browse...
☑ Collect All Exchanges Maximum Number of Exchanges to Collect 200

Simulation Parameters | Additional Options

☐ Add Advantage to High PRA Candidates
100 PRA Cutoff
0,00 Advantage
☐ Estimate Expected Utility
100 Estimation Iterations
☐ Reserve O Donors for O Recipients
☐ Check Additional HLA Information
☐ Include Compatible Pairs in Match Run
☐ Exclude AB Donors from Simulation
☐ Allow AB Bridge Donors

☑ Cluster and Isolate Top Solution OK Cancel

(b) Allow further specification of the optimization.

Figure 2.10: Parameter interface of KPDGUI.

Programming Language	**Domain**
pic, postscript	2D graphics
TEX, LATEX	document layout
Open GL	high-level 3D graphics
Mathematica, Maple	symbolic computation
Emacs Lisp	text editing
Prolog	logic
Lex and Yacc	program lexing and parsing

Table 2.2: List of DSLs and their domain. Source: [Hud98]

In fact, there are currently many examples of programming languages that can be considered domain-specific. Paul Hudak presented a simple table with some examples of programming languages and the domains they were designed for (see Table 2.2). Some of the listed items may be referring to systems with their own programming language within them.

2.2.1 Designing and implementing a DSL

There are two main categories of DSLs [Fow10]:

External DSL a domain-specific language that is created in separate to other host languages. It can be used by other languages, but usually has its own syntax and constructs. Some external DSLs are standalone products. For example, Mathematica [Mat] is a program for symbolic computation with its own programming language (the Wolfram Language) that can link to other, more general purpose languages. While these languages have the advantage of allowing syntax which is very close to the notations used by domain experts and can provide more adequate error reporting, also allowing for optimizations at the domain level, they require a great amount of effort to implement, since all of the components of the language would most likely have to be implemented for the language (like a dedicated lexer, parser, a compiler or interpreter). Moreover, language extensions would also be harder to achieve [MHS05].

Internal (or Embedded) DSL (or domain-specific embedded language) a domain-specific language that is built by using the features and constructs offered by a general-purpose language. In other words, internal DSLs are built using the constructs and abstractions of an existing general-purpose language. Main advantages are that they are easier to implement, as a lot of the fundamental mechanisms like the processor already exist; a lot of language features from the host language can already be good enough for domain-level abstraction; code is easier to extend and reuse. On the other hand, we are limited to the host language's syntax, which can distract the domain expert from the domain notation; error reporting can be ineffective if not customizable; and optimizations at the domain level may be harder to achieve [MHS05].

A good example of an external DSL is AMPL [AMP; FGK03], a language designed for optimization. It has its own syntax constructs, designed to bring the programmer closer to the domain. Since optimization relies on mathematical models, AMPL expressions are very close to what could be considered mathematical equations. It allows to describe optimization models like linear programs, and then has a data mechanism to fill the model with specific information. The language connects the model to a solver that will retrieve optimal solutions and several other related output.

In general, when creating a domain-specific language, we might have to deal with all the building blocks, more or less complex, of giving good domain abstraction to a domain expert. In fact, just like any software, programming languages are exposed to updates and optimization in a possibly endless process. It depends on the community support a programming language has, its popularity, how useful it is in the software development landscape, and just how much work and effort it takes to maintain such a language. Ultimately, the context of the domain itself (how relevant are domain-specific notations, how important are domain-specific optimizations or how necessary are good analysis constructs like error reporting) will influence the choice of the style of implementation to use.

2.2.2 The embedded case

As mentioned in the introduction, the goal of this dissertation was to develop a prototype of a domain-specific embedded language for modeling kidney exchange programs. In this section, we will address this approach in a more detailed fashion.

Like stated above, internal DSLs, also known as embedded domain-specific languages (EDSL) is an implementation technique which takes advantage of a host language, thus avoiding cumbersome work associated with implementing a completely new programming language. For instance, since we are working within the syntactical and semantical limits of an already designed language, compiling or interpreting issues will not be something to tackle with right away [VKV00]. EDSLs rely on their host language's features to become more and more expressive, in the sense that by capturing an increasingly better picture of the semantics of the domain in question, the language itself becomes more useful. Thus, when designing an EDSL, we are interested not so much in the syntactical details of its design (since, like we said, the "playground" for syntax is as limited as is the host language itself and, to some degree, the user, if still not familiarized, will have to learn at some level how to communicate with that host language, which can constitute a disadvantage in some cases), but rather in the semantics of what we are building. When reasoning about a EDSL, we must never forget who the intended future user is: a domain expert, to whom an application should be very quickly readable). An EDSL should provide very readable code at the domain level; because of this, it is often the case

Name	Domain
Django [Dja]	High-level web design
Numpy [Num]	Scientific computing
Pandas [Pan]	Fast multidimensional data processing
PyFX [Lej06]	real-time graphics effects
PyGPU [Lej06]	high-speed image processing

Table 2.3: Examples of EDSLs implemented in Python.

that they have a high-level programming language look and feel.

2.2.3 Python as a host language

Python [Pyt] is a general-purpose, high-level, multi-paradigm, substantially modular programming language that was conceived and initially implemented by Guido Van Rossum at the end of the 1980s. Like other programming languages, it has grown and matured into a fully fledged system used in a wide range of areas and suffering continuous updates and increments, currently going on version 3.7.3. Python has become one of the most popular and used programming languages around the world [TIO], and is growing, among other reasons, with the help of a large and supporting community. It was described as "batteries included" due to an extensive standard library, increasing the usefulness and variability of application domains of this truly general-purpose language. Python is also known for its high level of abstraction, using language constructs that center the programmer around the concept of the problem at hand, freeing them from the more low-level aspects of a computer system, like memory and process management.

A common form of interacting, developing and testing Python code is through an interactive environment, the Python interpreter. An advantage is that it is possible to execute single lines of code individually and inspect the result of their execution.

As a host language, Python is well suited for embedding our proposed implementation. The rest of this section lists and comments some of these features and characteristics, which are either suitable for the implementation itself or, looking from the DSL point of view, serve as useful constructs to aid in building new applications in the domain or even to extend it to some degree. Table 2.3 presents some examples of internal DSLs that were implemented in Python.

Syntax and code readability

Python prioritizes code readability and it is, indeed, one of its distinct features. Code written in Python, especially when aligned with good programming practices, is clear and concise, allowing for a good logical structure. The syntax is also a facilitating aspect of the language design: for example, its use of whitespace and indentation impose a particular structure such that even a less organized developer can maintain a minimal degree of readability. Flow control statements like the *for* loop or conditional execution (*if* statements) are very straightforward (see Figure 2.11 and also notice how indentation helps with code readability).

```python
for pair in kpd.pool:
    if pair.patient['Age'] > 40 :
        print(pair.patient['Name'])
```

Figure 2.11: Example of *for* loops in Python.

A high-level programming language

Python's high-level constructs make that language particularly advantageous for implementing an internal DSL: the host language is already powerful in the abstraction sense, so the work of keeping the new programming language focused on the domain is substantially facilitated. To that effect, Python has useful features such as built-in types (data structures like lists and dictionaries); compact but expressive constructs (such as list comprehensions and if-then-else expressions); its fundamental class system, allowing the definition of generic objects with associated methods and attributes, modeling domain level concepts; operator overloading, where we can add meaning to operations involving user-defined class instances and use Python's own syntax to express such operations; lambda expressions which are anonymous functions defined locally and allow simple function definition with single expressions; the decorator mechanism, which wraps functions with additional information, behavior and context (for example, at the domain level) and the property function which is another mechanism for setting and getting attributes from objects while performing computation that, for example, performs domain-level verification.

DSLs, whatever their form of implementation, are naturally also high-level languages, so it is in our interest to have available abstractions that will build on this domain-specific language.

Multi-paradigm design

Python offers different programming paradigm features that help bring about complex programs and integrate several approaches to problem solving and software design:

- *Imperative*: Python's execution has side effects and produces changes on the machine's state. Variable assignment and structure manipulation depend on the state of memory at the time of execution.
- *Procedural*: Statements are executed line per line, jumping to function calls and constructor methods when needed.
- *Functional*: Python's standard library carries methods and includes modules such as *itertools* and *functools*, which implement various concepts characteristic of functional programming languages, like mapping over a collections of items, using list comprehensions or currying functions. Also, functions are first class citizens, which means they can be passed to other functions and referenced as other primitive types. This is specially useful for delaying computation (a form of lazy strategy where a function is passed but is not intended to be applied immediately). An example of a function being used as an argument to another function is given in Figure 2.12 (in this example, although the output is shown as a list, it is in fact a *map* object, where the function to be applied is stored and applied only to the values of the list when the result of that application is requested).
- *Object-oriented (OOP)*: One Python's most recognized asset is its OOP constructs. They form the basis of the language since everything in it is an object. As mentioned earlier, Python also

```
>>>def square(x):
...  return x**2

>>>map(square, [1,2,3,4])
[1,4,9,16]
```

Figure 2.12: Example of functions being passed on as arguments to other functions. In this case, the built-in function *map* receives a function that squares its input as parameter and applies it to all elements in a list.

supports operator overloading, meaning that we can define the meaning of some common operators for our classes. That is especially useful for DSL design, as we get an implementation of standard syntax for free. Also, some of the methods in classes that can be overloaded represent special mechanisms in Python, like for instance the methods *__enter__* and *__exit__* which handle context-based object instantiation and destruction, with a specific kind of declaration. The double underscored method *__add__* makes the operator + available to our class instances. This gives for free some useful notations, for operators like sums and multiplications, or comparison operators like the equality operator == that is meant to return a boolean value, or even the index operator, given by the form *instance[item]*, to get some attribute named *item* from the instance in cause. Although we are still under Python's syntax definition, we are allowed to design and implement objects that look and operate like the language's built-in classes and types.

Dynamic language and typing

Python is considered a dynamic language in the sense that it has the power to change its own programs at runtime. Python has a feature called *introspection*, meaning that it associates types to values at runtime. Another dynamic feature is *reflection*, meaning that Python has the ability to inspect its own code and structure and even modify it at runtime. We get access to functionalities that allow, for example, to inspect class or instance structures, perform necessary modifications dynamically, defining functions at the time of execution, get information on a variable current value's type, among others.

Coupled with this dynamic behavior is also the strategy for typing. While other languages' type system is static (in the sense that it is the variables that are assigned types before execution and trying to assign of value of a different type results in an error), Python is, as mentioned previously, a part of another category languages where the typing is dynamic and it is the values which are typed. At any moment a variable can, for instance, be holding a boolean value and afterwards be assigned an integer or a list, if needed. Although for some this may be too permissive and error prone, it is very advantageous for metaprogramming which is, again, ideal for EDSL design. Combined with high-level constructs, it can result in higher productivity for developing applications in Python (and ultimately, in the EDSL), especially when the user is experienced. At the same time, a dynamic type system passes the responsibility of program correctness to the programmer, that applies operations and type specific methods at their own risk, and so a misguided use of variables and values can still impair the speed of production.

One other feature of the type system is that, in a few cases, when a particular definition or methods requires the argument passed to be of a certain type, the type system will try at runtime, and if necessary, to coherce the value passed to behave like being of a (possibly) different type. For example, in an *if* statement, the first argument passed must be a condition, i.e., an expression that evaluates to a value signifying *true* or *false*. So if in that case we pass an integer as a condition, Python will reinterpret that value as a boolean value, and consider a 0 to be the value *false* and any other integer to equate to a *true* expression.

Finally, when defining functions and methods, because Python does not enforce types before execution, the type of the arguments passed is not considered. However, the function might access a particular method or attribute that the argument is expected to have, and so there is an implicit assumption, not on the exact type of the value, but on the behavior and properties it holds. That is know as *duck typing*, following the metaphor “if it looks like a duck and quacks like a duck, it must be a duck”.

Custom exceptions and error handling

A very important part of programming is the ability to report errors and trace the execution that lead to an unexpected program termination. A compiler will be able, to the degree of its implementation, to denote such things as syntax errors, type errors and indicate runtime exceptions. For an interpreted language like Python, error handling happens, without surprise, at runtime, which is an asset for implementing a domain-specific language, since we can evaluate exceptions and provide domain-specific error reports.

Chapter 3

EDSL for Kidney Paired Donation

In this section we present PyKPD, our EDSL for managing and optimizing kidney paired donation, PyKPD. It has tools that allow to load, consult and manage pools of pairs and donors, a module for translating the data into a KEP graph as defined in the previous chapter, and an interface to declare criteria to be applied in the optimization process. PyKPD can handle three different kinds of participants in the pool: incompatible donor/patient pairs, compatible donor/patient pairs and altruistic donors. As use cases for this language, we identified the following scenarios:

Support for KEP development and research domain experts can model and experiment with the application of different criteria and achieve better KEP programs. Also, investigators in the field might not have to implement a KEP model from scratch if the constructs of the language are sufficient;

Software development more and more countries are implementing KEP programs at several different operating levels (local, regional, national, and even international). This implementation is a proposal of a tool that is integrated in a general-purpose language and thus can be made part of a larger solution by enhancing modularity in a common platform, which will facilitate the additon of layers such as GUI visualization (no need to use different programming languages or to integrate isolated modules of a more complex solution.)

We shall give an overview of the structure of the code, starting with the definition of the underlying architecture, and then discussing some of the details of each module, which equates to different steps in the process.

3.1 Architecture overview

A diagram for the structure of the implementation can be found in Figure 3.1. Conceptually, the modules that make up the EDSL can be divided into three layers. We give a brief description of each.

The *semantic layer* holds definitions of the basic elements of a KEP and its formulation, while also providing custom exceptions to better communicate programming errors to a domain-expert. It is composed by the following modules:

Entities module defines the classes that describe several concepts in kidney exchange, like patients, donors, pairs, among others. All the modules in lower layers use these classes as basis to build the process and impose semantic coherence from the domain perspective.

Exceptions module where the custom exceptions are defined.

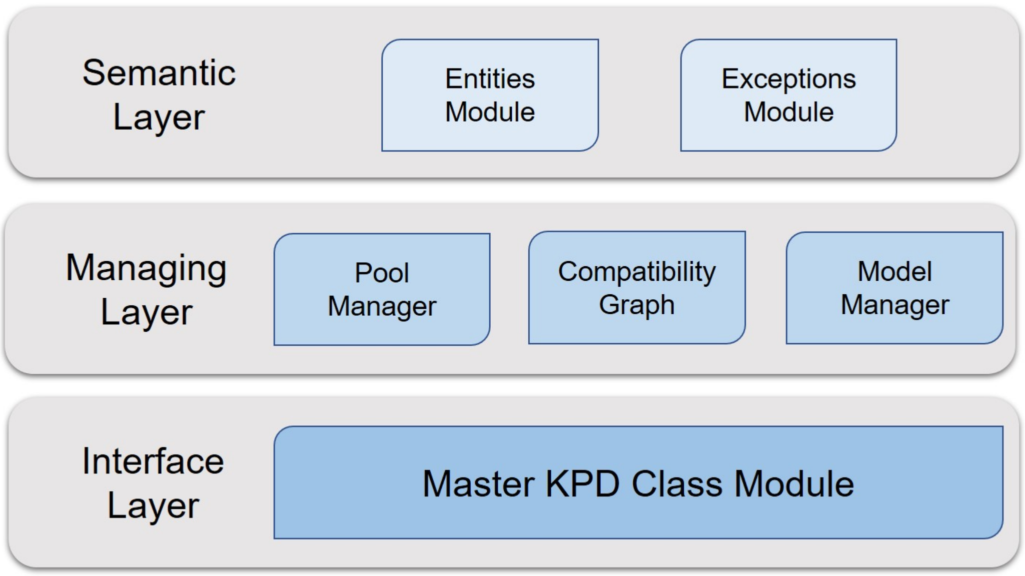


Figure 3.1: PyKPD Architecture overview..

The *managing layer* is responsible for instantiating the data about patients and donors, whose profiles induce a compatibility graph that is used to build an optimization model:

Pool Manager responsible for keeping information about the donors and patients registered for kidney exchange.

Compatibility Graph module defines the class that computes compatibility between donors and patients, builds the kidney exchange graph and lists all possible cycles and chains implied by the graph.

Model Manager linkage to the underlying optimization engine. It is where the routines for applying the models described in Section 2.1.4 for both simple and hierarchical formulations, after which the optimal exchanges are generated.

The *interface layer* combines the upper layers to provide a single, centralized tool to the user:

Main KPD module where the master class for describing KPD problems is implemented. This is where all the other modules are combined and made accessible to the user as a cohesive whole.

3.2 Implementation details

This section goes further into each of the modules, explaining their purpose in more detail and how to be leveraged for an optimized exploration of the KPD instance. We start by explaining the definitions in the entities and the exceptions module, which serve as basis for the rest of the implementation.

3.2.1 Entities module

This module is composed of all the classes that implement the fundamental elements of a KEP program. Figure 3.2 contains the classes declaration headings.

Class *Person* is used as a superclass for both the *Patient* and *Donor* classes. It provides a common interface for object initialization and overrides the indexed get and set methods (*__getitem__* and

```
class Person:
    (...)

class Patient(Person):
    (...)

class Donor(Person):
    (...)

class Pair:
    (...)

class UnspecifiedDonor:
    (...)

class Cycle:
    (...)

class Chain:
    (...)

class Exchange:
    (...)
```

Figure 3.2: Class definition headings for various KEP concepts.

__setitem__, respectively) to allow for attribute retrieval and setting. Internally, it does so by accessing and modifying a dedicated dictionary that holds such attributes.

Class *Pair* holds one patient and one donor object. As a way of ensuring some form of association, the initialization method for this class checks the internal identifier of each of the objects in the KEP pool, only allowing to create the instance of said identifiers are identical. Class *UnspecifiedDonor* is used to hold a single donor instance, which is considered to be an altruistic donor since it will have no associated patient.

Classes *Cycle* and *Chain* represent objects that hold corresponding sequences induced from a KEP graph. Finally, class *Exchange* represents a single exchange as in Definition 2.1.7.

All of these classes override the methods *__repr__* and *__str__* to present more readable representations of these entities.

3.2.2 Exceptions module

It is important to provide domain-specific information on erroneous uses of the languages, even when performing peripheral tasks such as file loading. This module contains simple exception classes that are used throughout the implementation to build an additional layer on top of Python and bring the language closer to the problem at hand. Please refer to Figure 3.3 for the listing of such exceptions. Since they can (and should) be derived from Python's *BaseException* class, they are left without any particular implementation, as they borrow all the properties and methods that the master exception class contains.

```
class KPDPersonDataError(BaseException):
    pass

class KPDPairError(BaseException):
    pass

class KPDFileTypeError(BaseException):
    pass

class KPDIndexError(BaseException):
    pass

class KPDNoExchangesError(BaseException):
    pass
```

Figure 3.3: Custom exceptions for PyKPD.

3.2.3 Pool Manager

This is the module responsible for loading a database of patients and donors and form the KEP pool. It holds only one method, *load*, which receives as input a path to a file as a string. It then proceeds by loading the file to a pandas *DataFrame* [Pan], which is used to traverse the list, line by line, and check whether that record refers to a patient/donor pair or a non-directed donor and what attributes are assigned to each person.

For each line, the method creates *Patient* and *Donor* objects accordingly and fills them with the corresponding attributes given by the data file. Then, depending on the case, these instances are injected into either *Pair* or *UnspecifiedDonor* objects. The pool assigns an integer as an identifier to each.

In order to facilitate the retrieval of information from the pool, a few of the class built-in methods were overridden. In particular, it is possible to retrieve individual elements from the pool by using the index operator (method *__getitem__*). Using an identifier not assigned in the pool will raise a custom exception to let the user know that a non-existent index was used. Also, the pool object can be iterated over in, for instance, *for* loops, thanks to the method *__iter__* which returns an iterator object.

Through the *property* mechanism, two attributes of the class were defined: the *index* and *registered*. If *pool* is a PoolManager instance, then *pool.index* and *pool.registered* return the list of identifiers and the list of corresponding elements in the pool, respectively. Figure 3.4 lists the main headings in the class.

As mentioned previously, an instance of a KEP pool is created through the loading of a data file that contains the specifications of all the registered patients and donors in the pool. The following section details the required form of a data file.

File for pool generation

The class for pool managing accepts comma-separated values files, i.e., files with extension *.csv*, that have semi-colons as separators. The file should have two-header columns, where the first line indicated to which participant the attribute corresponds to (either patient or donor) and the second line indicates the name of such attribute. In order to list a non-directed donor, just include a line where only the donor attributes are specified. A very simple example with fictional data is provided in Table

```
class PoolManager:

    def __init__(self, pool, settings):
        (...)

    def load(self, file):
        (...)

    def __getitem__(self, item):
        (...)

    @property
    def index(self):
        (...)

    @property
    def registered(self):
        (...)

    def __iter__(self):
        (...)
```

Figure 3.4: Main elements of the Pool Manager class.

Patient	Patient	Patient	Donor	Donor	Donor
Name	**Age**	**ABO**	**Name**	**Age**	**ABO**
Maria Simões	50	O	Roberto Oliveira	47	A
Miguel Tavares	57	A	Eunice Barbosa	55	B
João Fernandes	43	B	Vítor Fonseca	31	AB
Anabela Queirós	65	A	José Matos	28	AB
			Nuno Lopes	37	O

Table 3.1: An example of a KEP pool file structure.

3.1, with four blood type incompatible pairs and one altruistic donor. The attributes do not need to be the same for both elements, and the user has the freedom to input whichever information they feel is necessary. Notice that no identifiers are necessary: the method that loads the file automatically creates them by doing line numbering.

3.2.4 Compatibility Graph module

In this module, the class that handles the graph portion of the problem is implemented. As defined in Section 2.1.4, a KEP can be mathematically represented as a directed graph where the vertices are the pairs or altruistic donors involved, and directed edges imply that the donor in the origin vertex can donate an organ to the patient in the destination vertex. The structure of the graph will allow to describe the cycles and chains on which the IP formulation is based. In order to build this representation graph two main functionalities are needed: a method for establishing compatibility (which will originate the arcs in the graph) and a method to compute all the cycles and chains in the graph. The class for the KEP graph and the main methods are represented in Figure 3.5. As it can

```
class CompGraph:

    def __init__(self, pool, settings):
        (...)

    @staticmethod
    def _is_compatible_ABO(donor, patient):
        (...)

    @staticmethod
    def _is_compatible_HLA(donorHLA, patientAntiHLA):
        (...)

    def _is_compatible(self, donor, patient, how=['ABO', 'HLA']):
        (...)

    def create_comp_graph(self, how=['ABO', 'HLA']):
        (...)

    def compute_cycles_chains(self, cycleK, chainK=None):
        (...)

    def _get_all_cycles(self, K):
        (...)

    def _get_all_chains(self, K):
        (...)
```

Figure 3.5: The CompGraph class with some of its methods and private helper functions.

be noted, this class is instantiated by receiving as input a pool object, meaning that the operations it performs depend on a database of patients and donors.

This module has two main methods: *create_comp_graph* and *compute_cycles_chains*. The former is responsible for building the KEP graph and is where the compatibility testing functions are used to establish the graph arcs. Two static methods serve as helper functions for standard compatibility assertion: *_is_compatible_ABO* receives a donor and patient objects as input, expecting them to have a field name *ABO* where blood type is discriminated and check for a blood match. In turn, the method *_is_compatible_HLA* tests for HLA compatibility by traversing the fields in the donor profile whose name start with *HLA*, signifying the presence of different categories of antigens, and comparing them with the fields in the patient's attributes whose name starts with *AB HLA* (an indication of the presence of antibodies for that same HLA category). If there are no matches, then the pair is said to be compatible. Notice that the method *create_comp_graph* has a keyword argument named *how*, whose default value is a list of strings "ABO" and "HLA", meaning that, by default, the method checks for both ABO and HLA compatibility consecutively. This can be parametrized to perform only one of the checks but the method also accepts a new, user-defined compatibility function that should receive a donor and patient objects as input and return a boolean value. The latter processes the graph to extract all the cycles and chains up to a certain length limit. We follow the idea presented in [Con+13] and change the graph to allow for the description of cycles (including compatible pairs) and chains. In order to keep cycle and chain limits independent, we implemented a separate procedure to list chains. It was adapted from the routine we included in the implementation for enumerating cycles, which are part of the code that was made available in [Ped14].

3.2.5 Model Manager

The Model Manager leverages the work accomplished by the previous two modules to build the IP program and optimize our KPD instance after the desired criteria are defined. As usual, we present the headings for the main methods of the class, in Figure 3.6. Again, some dependency between the managing module and the previously defined modules can be noted, as the Model Manager takes as input for initialization a pool and a compatibility graph object. In fact, we need to have access to the pool to retrieve actual data from patients and donors when calculating the weights for each cycle and chain in the formulation. The graph object is where this module gets the information on said cycles and chains, which will be translated to the variables in the optimization problem.

The optimization itself depends on an external package, called *gurobipy*, for interacting with the Gurobi Optimizer [Gura] solution through Python [Gurb]. To a certain degree, this class serves as a wrapper that internally instantiates a *gurobipy* model and adds the necessary constraints and objective functions, but adds a level of abstraction that frees the user from having to learn too many specific details of the optimizer.

The method *create_model* initializes the gurobipy model, adding the binary variables that correspond to the cycles and chains and the constraints that ensure that every pair or altruistic donor only appears once in each solution (as described in the cycle formulation in Section 2.1.10). But even before this initialization it is possible to define the set of criteria that is going to be applied to the problem. This is done through the private methods *_maximize*, *_minimize* and *_ranked*. In each one, we provide the weighting attribution function as input (in the case of the first two methods, the function will be applied to each cycle and chain, so that should always be considered the standard input for the weighting function), and an optional description of the criteria for later reference. In the case of *_ranked*, because it implements the generalized iterative approach described in Section 2.1.10, the method expects a function that takes a pair and/or unspecified donor instance as input, in order to indicate from which attribute we are gathering the values that will be searched for in each cycle and chain. Also, this method takes two additional parameters: *ascending*, to indicate the order in which the values are to be iterated over, and *maximize*, to signal if the goal is to maximize the observation of each value or to minimize it. Both should be boolean parameters. One important detail is that these three functions are made private (in the Python sense), which means they are not to be accessed directly by the user. This should be done by the class defined inside the Model Manager class, called *_CriteriaDecl*. The class serves as an interface to access these functions and the motivation for this strategy will be made clear in the discussion about the main class for the EDSL. For any of the cases, adding criteria is deferred, meaning that the only thing that the methods do is store the information about each criteria in a sequence that will be later on traversed when the exchanges are to be extracted from the problem with an explicit command.

The method *solve* is the command to optimize the model. It looks up the previously declared criteria, checks if it is a simple maximization/minimization or an iterative and acts accordingly, calling helper functions to apply the corresponding objective function and run the solver. At each step, the optimal value of the last optimization run is added as a constraint on the new problem. Finally, method *generate_exchanges* retrieves the solutions given by the optimizer, which are described in terms of the variables of the problem. They are translated back to their cycle/chain form and injected into an

Exchange object, being stored stored in the Model Manager.

3.2.6 Main PyKPD module

The last module implements the main class in the PyKPD package and is the top-level interface used by the domain expert for interacting with our DSL. It integrates all the modules from the upper layers and structures the usage of such modules. The basic definitions of the class can be found in Figure 3.7. By examining the contents of the module, the first observed class is *KPDSettings*. It is a minimal class to hold parameter values that can be edited by the user. In the current state of PyKPD, they are only four: maximum cycle length (attribute *cyclemax*), maximum chain length (attribute *chainmax*), number of exchanges retrieved by the solver (attribute *numberexchanges*) and the criteria for compatibility (attribute *compatibility*). Default values can also be seen in the initialization method.

The class has different properties and only one method. These values are part of an idealized short set of keywords through which the user can specify a KEP problem and instantiate with specific pool data:

- The property *settings* accesses the settings of the instance (namely, the parameters in class *KPDSettings*);
- The property *pool* accesses the object's associated pool manager;
- The property *select* is the model manager's private class that exposes methods to declare criteria;
- The property *optimalexchanges* exposes a dictionary with information about the exchange set resulting from the optimization process, providing the optimal values for each criteria and the optimal exchanges;
- The method *solve* is the master command through which this main class checks for all points in the process (loaded data, graph generated and criteria declared) and runs the model manager's solving method in case all is correctly defined.

```
class ModelManager:

    class _CriteriaDecl:

        def __init__(self, modelmanager):
            self._manager = modelmanager

        def max(self, func, desc=None):
            self._manager._maximize(func, desc)

        def min(self, func, desc=None):
            self._manager._maximize(func, desc)

        def ranked(self, func, desc=None, ascending=True, maximize=True):
            self._manager._ranked(func, desc, ascending, maximize)

    def __init__(self, name, pool, compgraph, settings):
        (...)

    def create_model(self, verbose=0):
        (...)

    def _maximize(self, func, desc=None):
        (...)

    def _minimize(self, func, desc=None):
        (...)

    def _ranked(self, func, desc=None, ascending=False, maximize=True):
        (...)

    def _solve_ranked(self, func, opt_dir, ascending, desc=None):
        (...)

    def _solve_multi(self):
        (...)

    def solve(self):
        (...)

    def generate_exchanges(self):
        (...)
```

Figure 3.6: The ModuleManager class main components.

```
class KPDSettings:

    def __init__(self):
        self.cyclemax = 3
        self.chainmax = 3
        self.numberexchanges = 1000
        self.compatibility = ['ABO', 'HLA']

class KPD:

    def __init__(self, name=None):
        self._settings = KPDSettings()
        self._poolmanager = PoolManager(self._settings)
        self._compgraph = CompGraph(self._poolmanager, self._settings)
        self._modelmanager = ModelManager(name, self._poolmanager, self._compgraph,
            self._settings)


    @property
    def pool(self):
        return self._poolmanager

    @property
    def settings(self):
        return self._settings

    @property
    def select(self):
        return self._modelmanager.criteria_declarator

    @property
    def optimalexchanges(self):
        if self._optimalexchanges != set():
            return self._optimalexchanges
        else:
            raise KPDNoExchangesError('No optimal exchanges from current model listed.')

    def solve(self):
        self._modelmanager.solve()
        self._optimalexchanges = self._modelmanager.generate_exchanges()
```

Figure 3.7: The main KPD class.

Chapter 4

Examples and operational features

In this chapter, we present various usage examples of PyKPD (please refer to Appendix B for details on how to install the language), with the goal of illustrating the details that were discussed in the previous chapter. As a demonstration of the potential of the language, we provide a list of different criteria and define the weighting functions that will allow the optimizer to apply such restrictions.

4.1 A first example

We shall start by showing a simple example where to demonstrate basic functionality of the language constructs. In a Python interactive environment, we use the built-in mechanism and begin by importing PyPKD's main class, followed by the assignment of an instance object:

```
from pykpd import KPD

>>>kpd=KPD("KEP instance example")
```

Afterwards, the next command will import the dataset from a specific file (notice the attempt to load an invalid file, which raises an error):

```
>>>kpd.pool.load('poolexample1.xlsx')
(...)
pykpd.exceptions.exceptions.KPDFileTypeError: Selected file is not an CSV.
>>>kpd.pool.load('poolexample1.csv')
```

This a file with a very small pool, generated from an implementation of the generator described in [San+17], to which some additional information was added (the edit was done in a completely naive way and in no way represents any population-based distribution, as the purpose of the exercise is to simply show functionality. See Appendix A for inspection of the data.)

We now have loaded our set of registered pairs and altruistic donors. At this point, we can already perform some queries to the pool and find some information about the individuals in it:

```
>>>pair0=kpd.pool[0]
Pair (Patient=Mario Lisboa, Donor=Ricardo Lisboa)

>>>pair0.patient
Patient 0
```

```
Name: Mario Lisboa
AB HLA A: 1,10,15
AB HLA B: 76,79,85
AB HLA DR: 199
ABO: O
Age: 37.0
Country: PT
Cross match probability: 3.0
PRA: 0.0735711411594917
Year of arrival: 2015.0

>>>pair0.donor['Name']
Ricardo Lisboa

>>>len(kpd.pool)
14

>>>kpd.pool[14]
(...)
pykpd.exceptions.exceptions.KPDIndexError: ID not found in pool.
```

With simple expressions, we were able to retrieve data from a specific pair, as well as get other simple information such as the pool size (*len* is a built-in method that can be overriden to express length of an object). An example of an error situation where the pool is queried for an element that is not present in the registry is provided.

For the next step, and before performing optimization, we need to declare at least one criteria to define the problem completely. We will simply formulate a problem that maximizes the number of transplants. Our command will use as input a function that acts as a weight function for the cycle formulation:

```
>>>@criteria('Maximize transplants')
...def size(cc):
...   return len(cc)

>>>kpd.select.max(size)
```

In a very simple manner a criteria is defined as a Python function. With a decorator, the function definition is signaled as criteria inducing. The function *size* returns the length of the cycle (or chain, in this particular case, as there are altruistic donors present in the pool). The last line indicates the selection of maximum size, indicating that the underlying objective function has variable coefficients that correspond to the length of the cycles and chains. Optimizing such a problem will yield solutions where the number of transplants is maximum. The declaration, although very short, could still be made shorter: instead of defining *size* and passing it on to the criteria method, one could just have passed *len*. However, the form of declaration has two reasons: first, because the decorator mechanism is a way of declaring the intention of using the function as criteria for the optimization, while also providing a description for that criteria and thus making the code clearer; second, because the weighting function could easily become more complex depending on what is intended to be extracted from the problem variables. We are now ready to retrieve the optimal exchanges. First, the compatibility graph has to be generated:

```
>>>kpd.graph.generate(how=['ABO','HLA'])
Generated compatibility graph in 0s
>>>kpd.graph.adjacencies
{0: [1, 2, 5, 6, 8], 1: [2, 5, 6], 2: [0, 1, 2, 3, 4, 6, 7, 8, 12], 3: [1, 2, 5],
4: [13], 5: [1, 2, 5, 6, 8], 6: [], 7: [2, 5, 6, 8], 8: [1, 2, 5, 6, 8], 9: [1, 2, 5, 6, 8],
10: [1, 2, 5, 6, 8], 11: [0, 1, 2, 3, 4, 6, 11, 13], 12: [], 13: [2, 5, 6, 8]}
```

Property *adjacencies* is a dictionary that maps each vertex in the graph (represented by the identifiers in the pool) to the other vertex with which they are compatible, i.e., it forms an adjacency list for the compatibility. In the *generate* method, although it tests both ABO and HLA compatibility by default, that indication is passed explicitly for clarity.

We can now enumerate all cycles and chains. Before that, it is better to make sure the parameters for cycle and chain maximum length are set:

```
>>>kpd.settings.cyclemax=3
>>>kpd.settings.chainmax=3
>>>kpd.graph.compute_cycles_chains()
Computed cycles and chains in 0s
>>>kpd.graph.cycleschains
((9, 5, 2, 7), (1, 2, 8), (2, 8), (9, 8), (9, 8, 2), (9, 5, 8), (9, 5, 2, 8), (9, 1, 5, 8),
    (11,), (9, 2, 1, 6), (9, 1, 5, 2), (9, 5, 8, 6), (2, 7, 5), (5, 8), (9, 8, 1), (0, 5,
    2), (9, 8, 6), (9, 2, 8, 5), (9, 2, 8, 6), (9, 2, 7, 8), (9, 8, 5), (9, 8, 2, 4), (...))
```

Property *cycleschains* holds a list with sets of identifiers with indiscriminate cycles and chains. In this case, the chains of the list could be identified because of the fact that only the entity with identifier number 9 is an altruistic donor. The output is abbreviated, due to the size of the list.

Finally, we can now obtain a list of optimal exchanges. It is done by simply using the following command:

```
>>>kpd.solve()
>>>kpd.optimal['values']
{'0: Maximize transplants': 8}
>>>kpd.optimal['exchanges']
[Exchange((1, 5), (2, 4, 13), (9, 8, 6), (11,)), Exchange((2, 4, 13), (8,), (9, 1, 5, 6),
    (11,)),
Exchange((2, 4, 13), (5,), (9, 8, 1, 6), (11,)), Exchange((2, 4, 13), (5, 8), (9, 1, 6),
    (11,)),
Exchange((1, 5), (2, 4, 13), (8,), (9, 6), (11,)), Exchange((2, 4, 13), (8,), (9, 5, 1, 6),
    (11,)),
Exchange((1, 5, 8), (2, 4, 13), (9, 6), (11,)), Exchange((2, 4, 13), (5,), (8,), (9, 1, 6),
    (11,))]
```

After solving the problem, the property *optimal* returns the results of the run. This property returns a dictionary, where the item *values* presents another dictionary with the optimal values for each criteria and item *exchanges* contains the list of all the exchanges that achieve optimality for all the selected criteria. Note the singleton *(11,)* in the presented exchanges, an already compatible pair.

4.2 Other examples of criteria application

The previous section presented a simple example that showed PyKPD's functionality. It gave an overview of the tools available and the commands needed to run the entire optimization process. Due to the strategy chosen for criteria application, it is possible to instantiate a KPD object and immediately declare weight functions using the commands shown in the example. Criteria declarations are delayed until an explicit command that initiates the optimizer.

An important requirement for our DSL is the ability to express hierarchical, multi-criteria problems, it was fundamental that the language provided a mechanism to define a wide range of such criteria. This section presents numerous examples provided by other researchers in the field as a way of demonstrating the validity of the current state of the project. This time, we shall omit the initialization of the KPD instance and focus on the expression of the optimization criteria. We always assume that we have a KPD instance and the focus is on the weight function of each criteria:

1. ***Solutions with maximum number of transplants which serve the maximum number of patients waiting longer, lexicographically***: this item corresponds to two successive applications of different criteria, namely, where as many transplants as possible are achieved, and then prioritizing the longer waiting patients, which corresponds to that generalized iterative approach discussed in Section 2.1.4. The formulation can be defined in the following way:

```
>>>@criteria('Maximize transplants')
...def size(cc):
...    return len(cc)

>>>@criteria('Serve patients waiting longer')
...def waiting_time(p):
...    return None if kpd.is_altruistic(p) else p.patient['Year of arrival']

>>>kpd.select.max(size)
>>>kpd.select.ranked(waiting_time, ascending=True, maximize=True)
```

   This case, although not too complex, has already some new details that should be discussed. The first criteria was already observed in the past example and is reproduced in the same way. However, the second one required more attention, as it pertains a ranked optimization. First of all, the function does not operate on cycles and/or chains but on elements of the pool, because that is where the model retrieves the values that are used in building this ranked search. The second detail is that this criteria concerns only patients. We need to test first if the element is an altruistic donor, and return no value (in this case, the Python value of type None). The algorithm will ignore these and only consider valid observations. Finally, we declared the two criteria in the intended order (this is important, as using a different order of declaration would result in a different optimization program). Note the helper function *is_altruistic* provided by the class to test if a pool element is an altruistic donor.

2. ***Solutions that minimize the difference of age between donor and patient, summed for all patients, with maximum transplants***: The formulation of this combined criteria (again, it is actually two) is subtle: if we simply minimize the age differences, we will obtain with an empty set of exchanges, which guarantees the optimal zero value. So we should first maximize the number of transplants and then the sum of the age differences. We omit the definition of the

size function which is as before, and only the second criteria is shown now:

```
>>>@criteria('Minimize age differences, summed for all patients')
...def sum_age_differences(cc):
...   age_differences = [abs(donor['Age'] - patient['Age']) for donor, patient in
    cc.coupled()]
...   return sum(age_differences)

>>>kpd.select.max(size)
>>>kpd.select.min(sum_age_differences)
```

As criteria gets more complex, the need to resort to the host language constructs becomes more evident. It can be seen in the *for* loop or in the mathematical built-in function *abs* that calculates the absolute value. Although these might represent overhead in the learning process when using this tool, this allows for a wide set of functions and methods for free. This example uses a method present in both cycles and chains, *coupled*. It basically iterates through the object, but now associated the donor with the recipient implied by the path.

3. ***Solutions with maximum number of transplants and minimum number of international transplants***: Starting with the maximization of the number of transplants, and assuming that we have, registered in the pool, the country where every person is originally from, we can have a weight function that counts the number of international transplants for each cycle and chain, of the form:

```
>>>@criteria('Minimize number of international transplants')
...def number_intl_transplants(cc):
...   international_transplants = [1 for donor, patient in cc.coupled() if
    donor['Country'] != patient['Country']]
...   return len(number_transplants)

>>>kpd.select.max(size)
>>>kpd.select.min(number_intl_transplants)
```

We use a list comprehension to filter the cycle/chain and only keep the transplants that occur between a donor and a patient from different countries, each represented by the value 1. The length (or alternatively, the sum of the elements) of the list yields the number of international transplants.

4. ***Solutions with maximum number of transplants without any international transplants***: Ensuring maximum transplants with only national-level exchanges can be thought as a two-step weighting function. If there are no international transplants in a cycle or chain, the weighting function returns the number of transplants; otherwise, it is not considered and is assigned zero weight:

```
>>>@criteria('Maximize transplants without international transplants')
...def size_without_intl_transplants(cc):
...   intl_transplant = [donor['Country'] != patient['Country'] for donor, patient in
    cc.coupled()]
...   weight = 0 if any(intl_transplant) else len(cc)
...   return weight
```

```
>>>kpd.select.max(size_without_intl_transplants)
```

5. ***Solutions that maximize PRA, summed for all patients***: A simple criteria where the the coefficient of each cycle/chain is the sum of the PRA of each patient involved:

```
>>>@criteria('Maximize the sum of all patient PRA')
...def sum_pra(cc):
...   only_patients = [p.patient for p in cc if not kpd.is_altruistic(p)]
...   pra_list = [patient['PRA'] for patient in only_patients]
...   weight = sum(pra_list)
...   return weight

>>>kpd.select.max(sum_pra)
```

6. ***Solutions with maximum transplants that maximize PRA, summed for all patients***: This set of criteria, like other examples before, consists in first maximizing the number of transplants, followed by a maximization of the sum of PRA. The functions defined above can be used to achieve this:

```
>>>kpd.select.max(size)
>>>kpd.select.max(sum_pra)
```

7. ***Solutions with maximum transplants that maximize the number of patients transplanted with PRA above 80%***: After maximizing the number of transplants, the subsequent criteria demands a count of patients with PRA above 0.80 in each cycle/chain. That count is used as the weight:

```
>>>@criteria('Maximize number of patients with PRA above 80%')
...def count_pra_above_80(cc):
...   pra_list = [p.patient['PRA'] for p in cc if not kpd.is_altruistic(p)]
...   count_above_80 = len([pra for pra in pra_list if pra > 0.80]
...   return count_above_80
>>>kpd.select.max(size)
>>>kpd.select.max(count_pra_above_80)
```

9. ***Solutions with maximum transplants that minimize the geographical distance between donor and patient, summed for all patients***: In order to enforce this criteria, some form of location of every pair or non-directed donor is needed in order to compute distances. If, for example, every element in the pool is associated with a transplant center, and the GPS coordinates are given, a helper function to calculate distance can be used in the definition of the weighting function:

```
>>>import math
>>>def calculate_distance(donor, patient):
...   sq_lat = (donor['latitude'] - patient['latitude'])**2
...   sq_lon = (donor['longitude'] - patient['longitude'])**2
...   return math.sqrt(sq_lat + sq_long) # math.sqrt is the square-root function
```

```
>>>@criteria('Minimize distance, summed for all patients')
...def sum_distances(cc):
...   distances = [calculate_distance(donor,patient) for donor,patient in
    cc.coupled()]]
...   return sum(distances)

>>>kpd.select.max(size)
>>>kpd.select.min(sum_distances)
```

10. ***Solutions with maximum transplants that minimize the exchanges between Antarctica and Greenland***: Given a maximum set of possible transplants, minimizing the exchanges between a pair of localities translates to a minimization problem where the weights of the cycles/chains correspond to the number of exchanges between those locations:

```
>>>@criteria('Minimize exchanges between Antarctica and Greenland')
...def exchanges_antarctica_greenland(cc):
...   count = [1 for donor, patient in cc.coupled() if {donor['Country'],
    patient['Country']} == {'Antarctica','Greenland'}]]
...   return len(count)

>>>kpd.select.max(size)
>>>kpd.select.min(exchanges_antarctica_greenland)
```

Again, a list comprehension is used to filter the exchanges that happen between those specific locations. Because either the donor or the patient can be from one or the other country, comparing sets (Python structures to hold unordered, unique elements) simplifies the function definition.

11. ***Solutions with maximum transplants that maximizes number of O-type patients transplanted***: Very similar to the previous scenario, in the sense that weight is based on the observation of a particular value. In this case, after maximizing the number of transplants, the interest is to maximize type O patients in the exchanges:

```
>>>@criteria('Maximize number of type O patients transplanted')
...def count_type_O_patients(cc):
...   type_O_patients = [1 for donor, patient in cc.coupled() if patient['ABO'] == 'O']
...   return len(type_O_patients)

>>>kpd.select.max(size)
>>>kpd.select.max(count_type_O_patients)
```

In order to avoid having to check if the element is an altruistic donor, the use of the *coupled* method allows to always have a patient object in each iteration.

12. ***Solutions where maximum number of O-type patients are transplanted followed by maximization of number of transplants***: In spite of its triviality after the last situation (in fact, it is a simple switch on the order of the weight function declarations), this example illustrates the importance of hierarchy in the implementation, in the sense that order can have an impact on the final optimal exchanges retrieved:

```
>>>kpd.select.max(count_type_O_patients)
```

```
>>>kpd.select.max(size)
```

13. ***Solutions that maximize the number of transplants using the minimum number of altruistic donors***: The first priority is to maximize the number of transplants. Following the first criteria, the idea is to minimize the presence of altruistic donors in the optimal exchanges. A strategy for achieving this is to give weight 0 to all cycles and weight 1 to altruistic donors. Because of the constraint generated from the first run, the second run will necessarily retrieve solutions with the maximum number of transplants. This way, in the second run, the idea is that the optimal number of transplants is achieved by, if possible, only cycles, when that value is 0. When, due to the participants in the pool, that is not possible, it will try to satisfy the previous condition by combining the cycles with some chains. Testing for the presence of altruistic donors equates to testing if the variable corresponds to a chain.

```
>>>@criteria('Minimize altruistic donors')
...def altruistic_donor(cc):
...   return kpd.is_chain(cc)

>>>kpd.select.max(size)
>>>kpd.select.min(altruistic_donor)
```

14. ***Solutions where Barack Obama is served, with maximum number of transplants***: In this last example, we are interested in including a specific patient in the optimal exchanges. Because of the formulation for the criteria, it is one case where the order of the optimization is irrelevant:

```
>>>@criteria('Serve Barack Obama')
...def presence_barack_obama(cc):
...   presence = 0
...   is_patient_barack = [patient['Name']=='Barack Obama' for donor, patient in
    cc.coupled()]
...   if any(is_patient_barack):
...       presence = 1
...   return presence

>>>kpd.select.max(size)
>>>kpd.select.max(presence_barack_obama)
```

If the selected patient is present in the pool, the optimal value of the second optimization is known and it is 1. This follows naturally from the formulation. Each cycle and chain will have 1 has weight in the case where the patient is included and 0 when it is not. But because each patient can appear in only one cycle or chain in the exchange, the objective function will not be greater than 1. Hence, by using this function as criteria, we are simply choosing exchanges where there is one cycle or chain that includes the patient. Those exchanges, under the second optimization, will have value 1 and will be considered optimal exchanges by the solver. In this case, the fact that the optimal value of the criteria can only be 1, the order of the criteria is irrelevant.

Chapter 5

Conclusions and Future Work

5.1 Discussion of the presented work

In this dissertation we presented the design and implementation of PyKPD, an embedded domain-specific language for modeling kidney paired donation.

We provided context on kidney exchange's history, its conditions and many modalities for operation, the mathematical formulations for its optimization and related software that aim at managing and optimizing kidney exchange programs. We also gave background on domain-specific languages by differentiating them from general-purpose languages and presenting different strategies for their implementation, highlighting the one we adopted. All of this served as motivation and input for the work.

The language's architecture was detailed as a means to show structure and functionality. Finally, several examples were introduced as illustration of the proposal's capabilities. Its design was refined to allow expressions of a wide range of optimization criteria in a uniform way, and while it is not exhaustive in modeling the different aspects of the domain, has a structure that is capable of accommodating new features in the future.

The result, in our opinion, is a solution that allows a considerable degree of freedom in the specification and simulation of different optimization scenarios. We feel that the set of examples presented in the previous chapter document to the potential of the language, as several different combinations of criteria were described, pertaining realistic considerations about KEP issues.

5.2 Future Work

The accumulated experienced and knowledge led to several ideas of extensions for PyKPD that time and current resources did not make possible to explore:

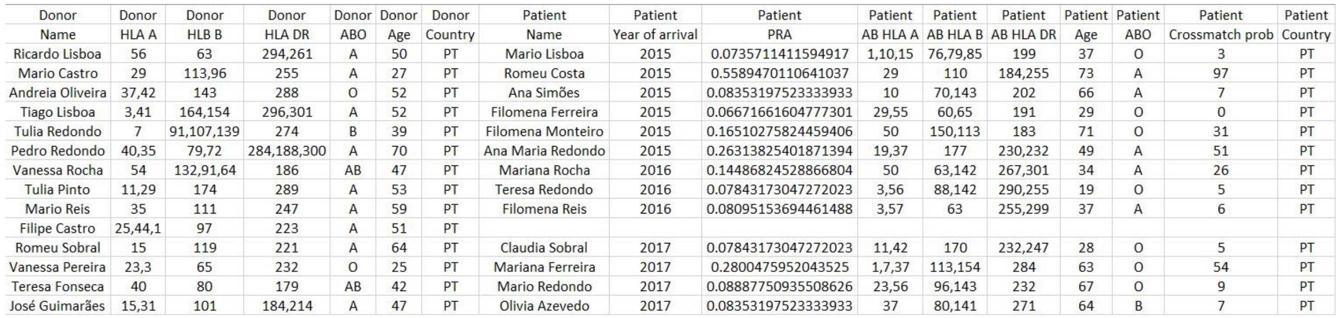
Alternative model formulations PyKPD models problems using the cycle formulation. It would be interesting to implement different formulations for the IP problems, leaving the choice up to the user of which one to use in a specific instance, or even extend to a dynamic process where the language, according to the size of the pool, would be capable of adapting and choose a more efficient model. Also, although performance was not assessed in this work, literature review pointed to the need, as KEPs grow in size and start to take more complex forms such as international cooperation, the computational approaches need to remain feasible time and memory-wise. In that sense, an idea would be to implement other strategies like the branch-and-price method that was presented in Section 2.1.4. Also, while the examples listed were numerous and allowed for the illustration of how several combinations of models are fitted easily in the language, we feel that the representation is not exhaustive. More research can be dedicated to understand which kind of criteria can be implemented that the language still falls short of representing;

Development of the syntax and language structure Python has many capabilities for DSL embedding. Exploring more implementation techniques might result in code that, while still being hosted in general-purpose language, will have enough abstraction on the domain-level to have an increasingly independent look and feel. As examples, overloading more operators and methods in our classes to add functionality to the language, such as editing the pool via the language constructs, or using more custom exceptions to provide more domain-level information on possible programming mistakes. This would be exceptionally useful when achieving some of the ideas described in the previous point.

Further testing Application to large data sets and testing other criteria would add to the implementation's credibility. For example, an interesting idea would be to try to model existing KEP programs, like the UK or the Portuguese program, as a new form of validation.

Appendix A

Example data file



Donor	Donor	Donor	Donor	Donor	Donor	Donor	Patient	Patient	Patient	Patient	Patient	Patient	Patient	Patient	Patient	Patient
Name	HLA A	HLB B	HLA DR	ABO	Age	Country	Name	Year of arrival	PRA	AB HLA A	AB HLA B	AB HLA DR	Age	ABO	Crossmatch prob	Country
Ricardo Lisboa	56	63	294,261	A	50	PT	Mario Lisboa	2015	0.0735711411594917	1,10,15	76,79,85	199	37	O	3	PT
Mario Castro	29	113,96	255	A	27	PT	Romeu Costa	2015	0.5589470110641037	29	110	184,255	73	A	97	PT
Andreia Oliveira	37,42	143	288	O	52	PT	Ana Simões	2015	0.08353197523333933	10	70,143	202	66	A	7	PT
Tiago Lisboa	3,41	164,154	296,301	A	52	PT	Filomena Ferreira	2015	0.06671661604777301	29,55	60,65	191	29	O	0	PT
Tulia Redondo	7	91,107,139	274	B	39	PT	Filomena Monteiro	2015	0.16510275824459406	50	150,113	183	71	O	31	PT
Pedro Redondo	40,35	79,72	284,188,300	A	70	PT	Ana Maria Redondo	2015	0.26313825401871394	19,37	177	230,232	49	A	51	PT
Vanessa Rocha	54	132,91,64	186	AB	47	PT	Mariana Rocha	2016	0.14486824528866804	50	63,142	267,301	34	A	26	PT
Tulia Pinto	11,29	174	289	A	53	PT	Teresa Redondo	2016	0.07843173047272023	3,56	88,142	290,255	19	O	5	PT
Mario Reis	35	111	247	A	59	PT	Filomena Reis	2016	0.08095153694461488	3,57	63	255,299	37	A	6	PT
Filipe Castro	25,44,1	97	223	A	51	PT										
Romeu Sobral	15	119	221	A	64	PT	Claudia Sobral	2017	0.07843173047272023	11,42	170	232,247	28	O	5	PT
Vanessa Pereira	23,3	65	232	O	25	PT	Mariana Ferreira	2017	0.2800475952043525	1,7,37	113,154	284	63	O	54	PT
Teresa Fonseca	40	80	179	AB	42	PT	Mario Redondo	2017	0.08887750935508626	23,56	96,143	232	67	O	9	PT
José Guimarães	15,31	101	184,214	A	47	PT	Olivia Azevedo	2017	0.08353197523333933	37	80,141	271	64	B	7	PT

Appendix B

Software installation

PyKPD is an embedded DSL for Kidney Exchange Program management and optimization. It has a module for loading the dataset, or pool, of the registered patients and donors, a module for graph generation and cycle/chain enumeration, and an interface to communicate with a gurobi model that performs the optimization. It is adapted to handle different modalities, such as:

- Altruistic donors and formation of chains
- Inclusion of compatible pairs
- Definition of hierarchical criteria for optimization
- Definition of a ranked, iterative formulation

The source files can be downloaded from

```
https://bitbucket.org/joaoprviana/pykpd/src/master/
```

If you already have Python and pip installed, then after downloading the repository, open a terminal, change to the repository directory and type

```
pip install -r requirements.txt
python setup.py install
```

This will install the package dependencies and the package itself. Afterwards, you can use it as the examples listed in this dissertation.

Bibliography

[12] *Doação Cruzada de Transplantação Renal Nunca Foi Realizada em Portugal*. (Accessed on 15/09/2019). Aug. 22, 2012. URL: `https://lifestyle.sapo.pt/saude/noticias-saude/artigos/doacao-cruzada-de-transplantacao-renal-nunca-foi-realizada-em-portugal`.

[13] *Médicos explicam como foi feito primeiro duplo transplante cruzado de rins com dadores vivos*. `https://www.tsf.pt/portugal/saude/interior/medicos-explicam-como-foi-feito-primeiro-duplo-transplante-cruzado-de-rins-com-dadores-vivos-3174377.html`. (Accessed on 15/09/2019). Apr. 18, 2013.

[ABS07] D. J. Abraham, A. Blum, and T. Sandholm. "Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges". In: *Proceedings of the 8th ACM conference on Electronic commerce*. ACM. 2007, pp. 295–304.

[Ame19] American Red Cross. *Blood Types Explained - A, B, AB and O — Red Cross Blood Services*. `https://www.redcrossblood.org/donate-blood/blood-types.html`. (Accessed on 20/09/2019). 2019.

[AMP] AMPL. *AMPL Website*. `http://www.ampl.com/`. (Accessed on 01/09/2019).

[ANZ] ANZKX. *Australia and New Zealand Kidney Exchange Program*. `https://donatelife.gov.au/about-donation/living-donation/australian-and-new-zealand-paired-kidney-exchange-anzkx-program`. (Accessed on 15/09/2019).

[ATM04] A.E. Roth, T. Sönmez, and M.U. Ünver. "Kidney exchange". In: *Quarterly Journal of Economics* 119.2 (2004), pp. 457–488.

[Bac] Backgrounder. *Kidney Paired Donation program — Canadian Blood Services*. `https://blood.ca/en/news-and-events/media-resources/kidney-paired-donation/backgrounder-kidney-paired-donation-program`. (Accessed on 16/09/2019).

[Bir+17] P. Biró et al. *First handbook of the cost action ca15210: European network for collaboration on kidney exchange programmes (enckep)*. 2017.

[Bir+19a] P. Biró et al. "Building kidney exchange programmes in Europe—an overview of exchange practice and activities". In: *Transplantation* 103.7 (2019), pp. 1514–1522.

[Bir+19b] P. Biró et al. "IP solutions for international kidney exchange programmes". In: *arXiv preprint arXiv:1904.07448* (2019).

[Bra+19] M. Bray et al. "KPDGUI: An interactive application for optimization and management of a virtual kidney paired donation program". In: *Computers in biology and medicine* 108 (2019), pp. 345–353.

[Can] Canada's Kidney Donation Program. *Living Donor Kidney Exchange - Kidney Paired Donation (KPD) Program — Professional Education*. https://professionaleducation.blood.ca/en/organs-and-tissues/programs-and-services/kidney-paired-donation-kpd-program. (Accessed on 09/15/2019).

[Cho07] S. Y. Choo. "The HLA system: genetics, immunology, clinical testing, and clinical implications". In: *Yonsei medical journal* 48.1 (2007), pp. 11–23.

[Con+13] M. Constantino et al. "New insights on integer-programming models for the kidney exchange problem". In: *European Journal of Operational Research* 231.1 (2013), pp. 57–68.

[Coo18] H. Cook. *Nation's longest kidney transplant chain surpasses 100 donations*. https://www.beckershospitalreview.com/quality/nation-s-longest-kidney-transplant-chain-surpasses-100-donations.html. (Accessed on 15/09/2019). Aug. 20, 2018.

[Cou] Couchsurfing. *Meet and Stay with Locals All Over the World*. https://www.couchsurfing.com/. (Accessed on 26/09/2019).

[Dic+16] J. P. Dickerson et al. "Position-indexed formulations for kidney exchange". In: *Proceedings of the 2016 ACM Conference on Economics and Computation*. ACM. 2016, pp. 25–42.

[Dja] Django. *The Web framework for perfectionists with deadlines — Django*. https://www.djangoproject.com/. (Accessed on 27/09/2019).

[Ell14] B. Ellison. "A Systematic Review of Kidney Paired Donation: Applying Lessons From Historic and Contemporary Case Studies to Improve the US Model". In: (May 2014).

[ERC] ERCPA. *CKD Facts & Figures*. https://ercpa.eu/index.php/facts-figures/. (Accessed on 27/09/2019).

[Eur] European Network for Collaboration on Kidney Exchange Programmes. *ENCKEP*. http://www.enckep-cost.eu/. (Accessed on 09/15/2019).

[FGK03] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A mathematical programming language*. 2nd ed. Duxbury-Thomson, 2003.

[Fow10] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[FWC09] P. Ferrari, C. Woodroffe, and F. T. Christiansen. "Paired kidney donations to expand the living donor pool: the Western Australian experience". In: *Medical Journal of Australia* 190.12 (2009), pp. 700–703.

[Gen+07] S. E. Gentry et al. "Expanding kidney paired donation through participation by compatible pairs". In: *American Journal of Transplantation* 7.10 (2007), pp. 2361–2370.

[Glo14] K. Glorie. "Clearing Barter Exchange Markets: Kidney Exchange and Beyond". PhD thesis. Erasmus University Rotterdam, Nov. 2014. URL: http://hdl.handle.net/1765/77183.

[Gura] Gurobi. *The fastest solver - Gurobi*. https://www.gurobi.com/. (Accessed on 25/09/2019).

[Gurb] GurobiPy. *Documentation*. https://www.gurobi.com/documentation/8.1/quickstart_mac/the_gurobi_python_interfac.html. (Accessed on 25/09/2019).

[Hom] HomeExchange. *Home exchange*. https://www.homeexchange.com/. (Accessed on 26/09/2019).

[Hud98] P. Hudak. “Domain-Specific Languages”. In: *Handbook of Programming Languages, Vol. III: Little Languages and Tools*. MacMillan Technical Pub, 1998. Chap. 3, pp. 39–60.

[Huh+08] K. H. Huh et al. “Exchange living-donor kidney transplantation: merits and limitations”. In: *Transplantation* 86.3 (2008), pp. 430–435.

[IPS] I. IPST. *Doação e Transplantação Orgãos e Tecidos de Células*. `http://ipst.pt/files/dados_2018.pdf`. (Accessed on 15/09/2019).

[KAV14] X. Klimentova, F. Alvelos, and A. Viana. “A new branch-and-price approach for the kidney exchange problem”. In: *International Conference on Computational Science and Its Applications*. Springer. 2014, pp. 237–252.

[Lej06] C. Lejdfors. “Techniques for implementing embedded domain specific languages in dynamic languages”. PhD thesis. Department of Computer Science, Lund Institute of Technology, Lund University, 2006.

[Luc07] M. Lucan. “Five years of single-center experience with paired kidney exchange transplantation”. In: *Transplantation proceedings*. Vol. 39. 5. Elsevier. 2007, pp. 1371–1375.

[Mat] Mathematica. *Software Web Page*. `http://www.wolfram.com/mathematica/`. (Accessed on 01/09/2019).

[MHS05] M. Mernik, J. Heering, and A. M. Sloane. “When and how to develop domain-specific languages”. In: *ACM computing surveys (CSUR)* 37.4 (2005), pp. 316–344.

[Mie+13] B. Mierzejewska et al. “Current approaches in national kidney paired donation programs”. In: *Ann Transplant* 18 (2013), pp. 112–124.

[MO12] D. F. Manlove and G. O’Malley. “Paired and altruistic kidney donation in the UK: Algorithms and experimentation”. In: *International Symposium on Experimental Algorithms*. Springer. 2012, pp. 271–282.

[Mod+10] P. Modi et al. “Living donor paired-kidney exchange transplantation: A single institution experience”. In: *Indian journal of urology: IJU: journal of the Urological Society of India* 26.4 (2010), p. 511.

[Mon+19] T. Monteiro et al. “A comparison of matching algorithms for Kidney Exchange Programs addressing waiting time”. In: (2019). Submited for publication in Central European Journal of Operations Research on April 15, 2019.

[Num] Numpy. *Official Website*. `https://numpy.org/`. (Accessed on 27/09/2019).

[Pan] Pandas. *Python Data Analysis Library — pandas: Python Data Analysis Library*. `https://pandas.pydata.org/`. (Accessed on 27/09/2019).

[Ped14] J. P. Pedroso. “Maximizing Expectation on Vertex-Disjoint Cycle Packing”. In: *Computational Science and Its Applications – ICCSA 2014*. Ed. by B. Murgante et al. Vol. 8580. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 32–46. DOI: `10.1007/978-3-319-09129-7_3`. URL: `http://dx.doi.org/10.1007/978-3-319-09129-7_3,`.

[PND] PNDRC. *PROGRAMA NACIONAL DE DOAÇÃO RENAL CRUZADA — Portal da Diálise - Insuficiência Renal Crónica*. `https://www.portaldadialise.com/articles/programa-nacional-de-doacao-renal-cruzada-pndrc`. (Accessed on 09/15/2019).

[Pyt] Python. *Welcome to Python.org*. `https://www.python.org/`. (Accessed on 16/09/2019).

[Rap86] F. T. Rapaport. “The case for a living emotionally related international kidney donor exchange registry.” In: *Transplantation proceedings*. Vol. 18. 3 Suppl. 2. 1986, p. 5.

[Reh] Rehash. *Trade Your Clothes Online on a Free Clothes Swapping Website*. `https://www.rehashclothes.com/`. (Accessed on 26/09/2019).

[Sai+06] S. L. Saidman et al. “Increasing the opportunity of live kidney donation by matching for two-and three-way exchanges”. In: *Transplantation* 81.5 (2006), pp. 773–782.

[San+17] N. Santos et al. “Kidney exchange simulation and optimization”. In: *Journal of the Operational Research Society* 68.12 (2017), pp. 1521–1532.

[Swa] P. Swap. *Trade Used Books with PaperBack Swap (the world’s largest Book Club)*. `https://www.paperbackswap.com/index.php`. (Accessed on 26/09/2019).

[Thi+01] G. Thiel et al. “Crossover renal transplantation: hurdles to be cleared!” In: *Transplantation proceedings*. Vol. 1. 33. 2001, pp. 811–816.

[TIO] TIOBE. *TIOBE Index — TIOBE - The Software Quality Company*. `https://www.tiobe.com/tiobe-index/`. (Accessed on 26/09/2019).

[UK] UK Living Kidney Sharing Scheme. *ODT Clinical - NHS Blood and Transplant*. `https://www.odt.nhs.uk/living-donation/uk-living-kidney-sharing-scheme/`. (Accessed on 15/09/2019).

[Uni19] United Network for Organ Sharing. *UNOS — US Organ Transplantation*. `https://unos.org/`. (Accessed on 01/09/2019). 2019.

[Uni98] University of Michigan Medical Center. *HLA Matching, Antibodies, and You*. `https://web.stanford.edu/dept/HPS/transplant/html/hla.html`. (Accessed on 20/09/2019). 1998.

[VKV00] A. Van Deursen, P. Klint, and J. Visser. “Domain-specific languages: An annotated bibliography”. In: *ACM Sigplan Notices* 35.6 (2000), pp. 26–36.