FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Combining Loan Requests and Investment Offers

Luís Pedro da Cunha Brandão Martinho

FINAL VERSION

Dissertation
Master in Informatics and Computing Engineering

Supervisor: Professor Doutor Luís Paulo Reis

July 2009

Combining Loan Requests and Investments Offers

Luís Pedro da Cunha Brandão Martinho

Dissertation
Master in Informatics and Computing Engineering

Aprovado em provas públicas pelo Júri:

14 de Julho de 2009

Abstract

This dissertation integrates a broader effort to setup a Peer-to-Peer lending community in Portugal. This work focuses on solving the infrastructural problem of combining investment offers from potencial lenders with loan requests from potencial borrowers, an issue which has yet to be given significant consideration within the literature.

The combination process must strive for an optimal result, which pleases lenders and borrowers alike, despite their opposing agendas. Simultaneously the combination result should also benefit the platform's business model, so as to keep it sustainable and profitable.

The text describes how several optimization metaheuristics – algorithms like Hill Climbing, Simulated Annealing, Genetic Algorithms and Particle Swarm Optimization – were applied to explore the solution space of the problem and to find optimal solutions, in light of a proposed utility function. Supporting the metaheuristic guided exploration, was a solution generation mechanism, powered by a constraint programming module.

During the process of building the combination system, a simple framework for reusable heuristics emerged to support the implementation. This framework featured a component-oriented architecture, and displayed modularity.

The results of this approach show how metaheuristic-driven optimization can be successfully applied to Peer-to-Peer lending combination problems.

Resumo

Esta dissertação integra um esforço mais abrangente para estabelecer uma comunidade de crédito colaborativo em Portugal. Este trabalho concentra-se em resolver o problema infra-estrutural de combinar ofertas de investimento de potenciais investidores e pedidos de crédito de potenciais creditados, um assunto que ainda não recebeu atenção significativa na literatura.

O processo de combinação deve procurar um resultado óptimo, que agrade a investidores e creditados, apesar de perseguirem objectivos opostos. Simultaneamente o resultado da combinação deverá também ser benéfico para o modelo de negócio da plataforma, de forma a mantê-la sustentável e rentável.

O texto descreve a forma como várias metaheurísticas de optimização – algoritmos como Subida de Colina, Arrefecimento Simulado, Algoritmos Genéticos e Optimização de Enxame de Partículas – foram aplicadas para explorar o espaço de soluções do problema e para encontrar soluções óptimas, à luz de uma função de utilidade proposta. A suportar a exploração guiada por metaheurísticas, encontra-se um mecanismo de geração de soluções, apoiado por um módulo de programação de restrições.

Durante a construção do sistema de combinação, emergiu uma plataforma simples para heurísticas reutilizáveis para apoiar a implementação. A plataforma possui uma arquitectura orientada por componentes, demonstrando modularidade.

Os resultados desta abordagem demonstram como a optimização usando metaheurísticas pode ser aplicada com sucesso a problemas de combinação em crédito colaborativo.

Acknowledgments

I would like to thank my supervisor, Prof. Luís Paulo Reis, for supporting my work even when I digressed or got stuck, and helping me keep a clear focus on solving the problem at hand. This work would not have been finished without his advice.

I would also like to thank my Roda project partner, Simão Rio, who originally came up with the idea of Peer-to-Peer loans back in 2007, and with which I have worked very hard to build the Roda platform. This work puts us one step closer.

I also thank Hive Solutions Lda., my current employer, not only for the support I received to complete this dissertation, but also for the skill set I have gained while working for the company, which made solving a series of technical issues a lot easier than it was before.

I thank my parents, my sister and Catarina.

Contents

List of Figures

List of Tables

Abbreviations

API	Application Programming Interface
BdP	*Banco de Portugal* (Bank of Portugal)
CO	Combinatorial Optimization
CP	Constraint Programming
CSP	Constraint Satisfaction Problem
EC	Evolutionary Computation
EP	Evolutionary Programming
ES	Evolution Strategies
GA	Genetic Algorithms
HC	Hill-Climbing
IoC	Inversion of Control
OO	Object-Oriented
P2P	Peer-To-Peer
PEP	Python Enhancement Proposal
PRS	Pure Random Search
PSO	Particle Swarm Optimization
SA	Simulated Annealing
SCM	Software Configuration Management
SI	Swarm Intelligence
VCS	Version Control System
WWW	World Wide Web

1 Introduction

Arguably one of the most powerful concepts to emerge from the Internet was that of the social web: a network made not only of machines, but also of people who could now relate directly, no matter how geographically apart. This new interaction paradigm not only challenged existing business models, but also motivated completely new ones. Some of the business models that were most impacted were those that involved intermediation. This was the case with the various forms of employment portals (where employees and employers could meet directly) and auction portals (where buyers and sellers could meet directly). The next step would be direct person-to-person lending or peer-to-peer lending, starting with Zopa in the United Kingdom in February 2005 [Reg05].

This work reports on efforts made from early 2007 to the present day to create an online Social or Peer-To-Peer (P2P) Lending platform, operating in Portugal. P2P lending is the "name given to a certain breed of financial transaction (primarily lending & borrowing, [...] which occurs directly between individuals ("peers") without the participation of a traditional financial institution" [Wik09a].

1.1 Background

The project received the working name "Roda", meaning Circle in Portuguese, and intended to offer a new approach to P2P lending. As all P2P lending efforts, the fundamental aim was to create a simpler, more transparent financial vehicle than those currently available. This new model also intends to be fairer, safer and even more profitable than existing corporate options, e.g.: bank loans. To achieve these goals, the Roda project intends to gather a vibrant community around the lending project, creating additional channels for human interaction in true Web 2.0 style.

1.2 Motivation and Objectives

The success of the project was seen as greatly dependent on the individual satisfaction of both lenders and borrowers, despite their opposing agendas. It was thus required to create a mechanism that could combine loan requests and investment offers in a fashion which pleased the greatest amount of participants, while protecting the interest of the platform operator. It was this key problem that motivated this dissertation.

The main objective of this work was then to build a system capable of successfully finding optimal combinations of loan requests – defined by the amount requested and the maximum rate at which the potential borrower is willing to repay the money – and investment offers – defined by the amount offered by the lender and the minimum interest rate at which the potential lender is willing to receive its money back. Simultaneously, the system should attempt to maximize the amount of money traded, due to the volume based business model of the project.

The construction of a web-based application was also an issue when building an online Social Lending platform, since the majority of customer interaction is to take place through it. This issue, together with further interaction paradigms such as Social Rating, despite relevant for the project as a whole, are considered outside the scope for this work.

1.3 Adopted Conventions and Practices

All references to code are done in `monospaced font`, references to external libraries or products are done in *italic font*. The naming convention used in the actual implementation followed PEP8 (Python Enhancement Proposal) – Style Guide for Python Code [RW09], which most significantly means method names and variables in lower case with words separated with underscores and class names in camel case. All exceptions to these conventions are due to conflicting conventions in third-party code.

To aid in Software Configuration Management (SCM) the *Git* Version Control System (VCS) [Cha09] was put in place to keep track of changes to the source code. The VCS was provided by *GitHub* [Git09a] and is publicly available at the project page [Git09b]. Figure 1.1 shows the "impact" timeline in the VCS, with "impact" defined as the number of lines added plus lines deleted for all non-merge commits during a week period, the data was obtained from [Git09b].

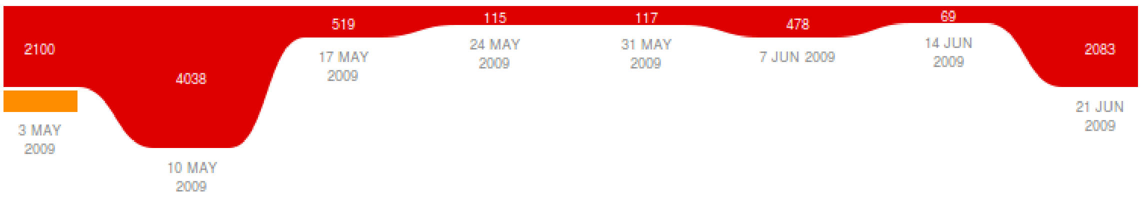

Figure 1.1: Impact timeline in the VCS [Git09b]

1.4 Thesis Structure

This section provides a general introduction to the thesis, which contains 7 additional chapters:

- Chapter 2 and 3 describe the the current environment for this work, in terms of the competitive business landscape in terms of Peer-to-Peer credit and the existing state-of-the-art regarding optimization solutions, respectively.
- Chapter 4 presents a detailed description of the problem which this work addresses.

- Chapter 5 presents an overview of the solution designed for the problem, as well as the rationale behind it.
- Chapter 6 thoroughly describes the solution, providing additional information about the architecture as well as some relevant implementation details.
- Chapter 7, the results analysis, shows how the solution performed under several test scenarios, which intended to simulate potential operational environments.
- Chapter 8 sums up the key contributions from this work, evaluates to what extent were the initial objectives attained and presents possible investigation paths towards achieving better loan combination results.

2 Person-to-Person Lending

This chapter introduces the concept of Person-to-Person lending, from an electronic commerce perspective. Despite being a natural form of human interaction, which can be traced to early human communities, the opportunities created by the advent of modern global communities change the rules of the game and have spawned a number of distinct initiatives.

2.1 Background

A written record of early social lending can be found in a British XVIII century document which states:

"Whereas it has been an Ancient Custom in this Kingdom for Divers Artists to Meet together and unite themselves into Society (But more especially for those who follow any Art or Mystery) to promote Amity and true Christian Charity" (Rules of the Second Mechanics Society, Plymouth, 1794 as cited in [Gor98]).

Friendly societies can be traced back to the first half of the 17th century. Evolving from the ancient gilds and despite being similar to freemasonry societies, members of friendly societies were not as much merchants as they were wage earners or artisans. Throughout the 18th century, friendly societies gained legal support regionally, until being nationally recognized by the Friendly Societies Act of 1793, which aimed to grant various privileges in return for registration ([Gor98]). The 19th century was a period of tremendous growth for these societies with membership surging from from 600,000 in 1793 to 4 million in 1874, representing the most well attended voluntary associations after churches ([Gor98]).

Portuguese social lending history stems from the tradition of an age-old nonprofit sector[1]. According to work regarding the Portuguese nonprofit sector by Raquel Franco [Fra05], by the end of the XV century there were three types of associations, in Portugal, focused on economic concerns, that emerged as a way of providing people with means to face natural disasters, and professional contingencies, and at the same time, promoting solidarity among people from the same professional activity.

1 The Portuguese nonprofit sector is at least as old as the Portuguese nation-state. The origins of the country date back to 1143 and organized charities existed in the territory even before then.

One example of these economic-based civil society associations was the *celeiros comuns* (common granaries). These associations were the result of the initiative of common people who asked the public authorities to provide them with a legal framework covering the solidarity practices already in place. These associations were a way of accumulating stores of grains supplied by all peasants during good times, to be loaned to peasants to ensure the continuity of agricultural activity for those affected by natural disasters or difficult times.

These solidarity principles spread to other peasants' activities such as cattle breeding. In this context, other forms of local solidarity emerged as mutual insurance schemes and mutual credit associations. Their aim was to provide means to those in need through contributions collected from all members of the association. Members contributed goods and money equally to a collective fund that could make low interest loans to members in difficulty.

As Franco describes, by the end of the XVIII century the middle class, namely state officers, liberal professions, and merchants, were also losing status and income. As the corporatist framework and assistance schemes were disappearing, new forms of association among members of the middle class, emerged such as mutuality schemes that sought to provide help in case of illness and provide credit in case of financial difficulties. The associations with an insurance profile were designated as *montepios*, whereas the associations with a credit saving deposits profile were designated as *caixas económicas* .

There are currently around 120 mutualist associations in Portugal, with about 900,000 members and 2 million beneficiaries. The União das Mutualidades Portuguesas (Union of the Portuguese Mutualities), is the federative organization which protects the interests of the associations.

It is from this collective tradition that spawns the first model of online social lending, not surprisingly in Europe. February 2005 marks the start of the UK-based operation Zopa, the first venture into building a P2P Lending Platform ([Reg05]).

2.2 P2P Lending Platforms

Peer-To-Peers lending platforms are online platforms where borrowers place requests for loans online and private lenders bid to fund these. This new approach to social lending enables local communities to grow beyond its geographical borders by recruiting members and connecting with similar groups no matter how distant. For individuals this means having access to business opportunities which were, until now, exclusive to banks and other financial institutions.

The ability to scale along with the customer base that online P2P lending platforms displayed made them an alternative to what had become the standard mechanism for financing: banks. With the increase in comercial trade, society evolved existing person-to-person lending institutions to the modern day commercial banking system. With these new – more sophisticated – institutions, lenders could get a fixed return and delegate risk management to a third-party. Borrowers could now access larger sources of capital, as long as their financial status could be proved to the bank. Intermediary institutions drew their profit from growing economies of scale and from the spread between the rates payed to lenders and the rates charged to borrowers.

Recently the continually growing structures of banks, and respective costs, the lack of trust in the financial system with recent bankruptcies together with structure deficiencies in the current ability to evaluate and upkeep loan contracts, as address in works such as [Par99], has opened the door for alternative financing mechanisms such as P2P lending.

P2P lending presents itself as a disintermediation alternative to financing, with key advantages being the reduced infrastructure that needs to be support, thus reducing overhead costs, and the increased sense of community with aids in differentiating good from bad loans and helps keeps the repayment rate higher, as is described later in this chapter.

The next sections presents some of the current approaches to social lending, that have taking place in the market.

2.3 Lending Models

2.3.1 Pooled Lending

This financing model created by Zopa, was first hinted at in Zopa's patent application [CP00]. The model consists of having a lender lend money to a pool of borrowers with similar credit ratings. In this model the risk of capital and interest for the lender is defaulters in the pool. The risk of capital and interest of the lender is reduced considerably. This model is similar to the traditional bank model and does not allow the lenders to select individual borrowers [Wik09a]. Here the main platform provides a “match-making” service, which joins lenders and borrowers behind the scenes.

Taking the example of Zopa, as described in [Zopa09a], the platform operator analyzes the credit scores of people looking to borrow and determine their credit rating, usually with the help of a third-party rating agency. Lenders make lending offers – stating that they would like to lend a certain amount to a certain type of borrowers for a certain interest rate and during a certain amount of type. Borrowers create loan requests, stating the amount they desire and the rate they intend to pay. The platform takes care of diversifying the investments by spreading the lenders money across multiple borrowers. If the system finds a so called Zone of Partial Agreement (hence the name Zopa) a match is complete, and borrowers enter into legally binding contracts with their lenders. Borrowers repay monthly by direct debit. If any repayments are missed, a collections agency uses the same recovery process that the conventional banks use. Zopa earns money by charging borrowers a transaction fee and lenders an annual servicing fee.

2.3.2 Direct Lending

This model carries a more peer-to-peer sentiment, and consists of having the lender directly pick the borrower to which to lend. The choice is typically based on information provided by the borrower (borrower's identity, occupation, residence, financial status), and validated by the platform which serves as a trust anchor, typically with the support of traditional rating and identity verification institutions.

Potential lenders bid on funding all or portions of loans for specified interest rates, which are typically higher than rates available from depository accounts at financial institutions. Each loan is usually funded with bids by multiple lenders. After an auction closes and a loan is fully bid upon, the borrower receives the requested loan with the interest rate fixed by the platform operator at the lowest rate acceptable to all winning bidders.

An example of this model is the US operator, Prosper Marketplace. At Prosper, individual lenders do not actually lend money directly to the borrower; rather, the borrower receives a loan from a bank with which Prosper has contracted. The interests in that loan are then sold and assigned through Prosper to the lenders, with each lender receiving an individual non-recourse promissory note.

2.3.3 Group Lending

The idea of group lending emerged from credit institutions in developing countries which choose to lend to self-selected groups of entrepreneurs, who are jointly liable for a single loan, instead of lending directly to individuals. This approach has gained tremendous attention from economists an innovative way of providing credit to those without access to the formal market ([Wyd99]). The interest in this delivery method came from the notorious success of several group lending institutions, especially that of the Grameen Bank in Bangladesh, founded in 1983 by Bengali economics professor Muhammad Yunus and reported in works such as [Ho88]. What has also fascinated economists is the manner in which group lending appears to be able to exploit social ties and the potential for peer monitoring and social pressure between borrowers in small-scale credit transactions.

It is based on this notion that group lending has entered the realm P2P lending, motivating an approach which emphasizes the importance of building tighter social networks inside the lending environment to create opportunities for peer monitoring and social pressure. This topic has seen little research, as of yet, as knowledge is still kept inside of each P2P lending platform provider's proprietary loop as they experiment around the concept.

2.4 The Role of Social Networks

A relevant attraction in social lending, when compared to traditional investment vehicles, is the inherently humane approach to financing. This provides lenders with a different kind of financial instrument: the ability to lend assistance to those with whom the lenders feel an affinity, despite inferior credit rating. An interview found in [Bog09] accurately illustrates the kind of behavior which arises. The document reports on how a major in the U.S. Marines states that, when faced with someone who has served in the military and maybe made mistakes in the past, stating as example maxing out credit cards, he will fell more apt to invest, even when the borrower does not present a high credit rating. The same document refers a case of a borrower which got a significant part of its funding from lenders in some way connected to its hometown. Cases like this illustrate the significance of human ties in environments where direct P2P lending is made possible.

Social lending platforms have strived to both build communities of their own, within the lending environment, and to connect with other existing social networking vehicles, leveraging existing relationships between members.

2.5 Impact on Microfinance

The sector of microfinance, referenced above in concern to the Grameen Bank experiment, has also witness the development of P2P platforms. These projects allow the general public to directly fund microfinance institutions or even microentrepreneurs, allowing them to tap into good-willed capital from any part of the world.

Kiva.org launched in 2005 and was the first micro-lending Website that enabled an individual to lend money to a micro-entrepreneur in the developing world through a microfinance institution. Kiva states its mission as “(...)to connect people through lending for the sake of alleviating poverty” [Kiv09b]. At the time of writing, Kiva's 119 field partners collaborate with Kiva, dramatically extending its scope and reach. Kiva has raised and lent over 70 million USD.

2.6 Current Research and Future Directions

One of the first, and few, works to address the Peer-to-Peer lending problem is that of Michael Klafft, [Kla08]. The work presents the results from an empirical analysis of 54.077 listings on the Prosper platform. The results demonstrate that verified bank account information and the credit rating are important factors for a listing's success. Additional personal information, such as a photo of the borrower, also has a significant influence on funding, as is shown in Klafft's work. Less important, though still relevant, are peer groups within the online community. The study finds that, similarly to the traditional banking system, interest rates on P2P-lending platforms are primarily determined by credit rating and debt-to-income ratios. As empirical data show, it will mainly be the reliable, AA- or A-rated borrowers who can exploit the opportunity of lower cost loans. High-risk borrowers, however, have serious difficulties in successfully acquiring loans online, as only about 5.5% of their listings are funded.

A report published by Celent [Cel07], regarding the US market, showed that in 2005, there were $118 million of outstanding peer-to-peer loans. In 2006, there were $269 million, and, in 2007, a total of $647 million. The projected amount for 2010 is $5.8 billion, as shown in Figure 2.1. The same work estimated that typical loan amounts range from $8,000 to $20,000; on some sites, multiple lenders may fund a loan, each offering to lend $25 to $200 to a borrower.

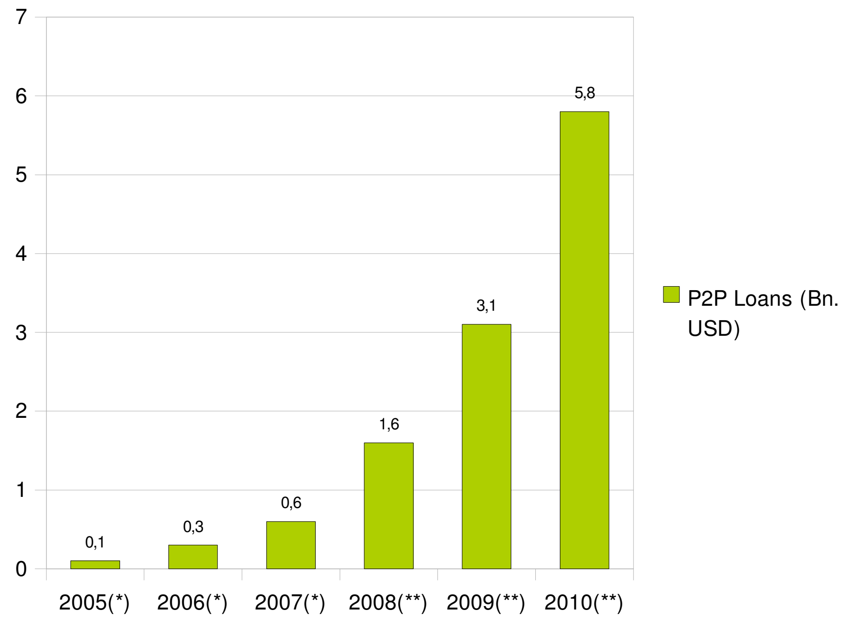

Figure 2.1: Peer-to-peer loans US market value: estimated(*), and projected(**) data from Celent [Cel07].

These numbers are nonetheless from before the height of the Financial crisis of 2007–2009 [Wik09b], which provoked a profound liquidity crisis and caused a decrease in international trade. It is still not clear what will be the impact of the regulation and consumer awareness changes, that are underway across the globe, on the social lending market. Nonetheless some, such as the Uncrunch America initiative and other P2P lending proponents argue that social lending is not only not part of the problem, but might also be an important part of the solution. The official site of the Uncrunch America initiative states that:

"With banks tightening credit, American families and businesses need new alternatives. Two leading social lending companies Lending Club and Virgin Money have teamed up with other consumer-friendly personal finance companies, including On Deck Capital, Credit Karma, Geezeo and ChangeWave to create an awareness campaign called UNCRUNCH AMERICA™, aimed at providing resources to consumers and businesses in need of credit solutions. Its goal is to help resolve the credit crunch and rebuild the economy by delivering consumers secure, trustworthy tools and infrastructure to finance necessary expenses and make critical investments." in [Unc09].

Although still requiring research, P2P may come across as logic answer to the current problem of reduced liquidity, since there are large amounts of money frozen due to lack of investor confidence in traditional institutions. The money flow between lenders and borrowers does not harm reserves or expand public debt, and can have a sustainable positive effect on each nations Gross Domestic Product as pointed out by [Hen09].

According to user edited list in [Wik09a], there are currently 35 social lending platforms across the world, including countries such as the UK, the USA, Canada, France, Germany, Italy, Spain, Switzerland, Poland, Sweden, Japan, China and India, among others.

2.7 Summary

This chapter intended to provide a comprehensive past, present and future presentation of the person-to-person lending topic. The chapter begins with an historical contextualization, by describing some of the social systems which predated the online P2P lending platforms. The text goes on the present a review of different approaches made to this problem. Finally the chapter presents results from the first studies on the subject and also uses work from existing research to project into the future of the P2P lending. With a long history behind, and demonstrating sustainable growth, even in troubled times such as ours, the evidence seems to show that a bright future awaits P2P lending worldwide.

The next chapter provides a technical survey of the environment that surrounds the optimization of problems of the type presented by investment-loan matching in peer-to-peer lending.

3 Optimization Metaheuristics

The quest for solving problems in an efficient and effective way is, by definition, one of the goals of all engineering work and probably one of the most consistent pursuits in human activity. Many of these problems can be defined as a search for the best configuration of a set of variables in order to achieve a certain goal, this field of study is called Optimization.

3.1 Optimization Problems

Optimization problems are usually categorized into two major families: those which solutions come from a spectrum of continuous values and those which solutions come from a discrete range of values.

Optimization problems which involve continuous ranges for parameters, imply the feasible solutions space to be infinite. Methods for solving optimization problems under continuous input parameters are found to be classified as either gradient-based or non-gradient based [SHJ00]. Gradient based methods are the most used ones and three subclasses of these kind of methods can be identified: Gradient Based Search Methods (GBSM), Response Surface Methods (RSM) and Stochastic Approximation Methods (SAM) [Res06].

Due to the discrete nature of the problem's decision variables, these methods lie outside the scope for this work and, as such, are not discussed here.

Inside the second family of problems, that of discrete variable optimization problems, there is a broad class of problems which has been of particular interest for research in the recent years and is relevant to work at hand. This is the class of Combinatorial Optimization (CO) problems. According to [PS82], in CO problems we are looking for an arbitrary object from a finite – or possibly countable infinite – set. This branch of optimization, lies where the set of feasible solutions is discrete or can be reduced to a discrete one, and the goal is to find the best possible solution [Wik09c].

A more formal definition would be, found in [BR03]:

Definition 3.1: Combinatorial Optimization Problem

A Combinatorial Optimization problem $P=(S,f)$ can be defined by:

- a set of decision variables $X=\{x_1,\ldots,x_n\}$;
- variable domains $D_1,\ldots,D_n$;
- a set of constraints $C_1,\ldots,C_n$, which relate the decision variables;
- an objective function *f* to be maximized, where $f:D_1\times\ldots\times D_n\rightarrow\Re^+$.

From which results that the set of valid solutions, or problem space, for a CO problem is thus:

- $S=\{s=\{(x_{1,}v_1),\ldots,(x_n,v_n),\}:v_i\in D_i\wedge s\ \mathit{satisfies}\ C1,\ldots,C_n\}$.

Solving combinatorial optimization problem is then the process of finding a solution $s^*\in S$ with minimum[2] objective function value, i.e., $f(s^*)\leq f(s)\forall S$.

Due to the practical importance of CO problems, a number of algorithms have been developed to approach them. These algorithms fall into two categories: complete and approximate. Complete algorithms are guaranteed to find, for every finite size instance of a CO problem, an optimal solution in bounded time [PS82]. Yet, for CO problems that are NP-hard [GJ79], no polynomial time algorithm exists. Therefore, complete methods might need exponential computation time in the worst-case. This often leads to computation times too high for practical purposes. Thus, the use of approximate methods to solve CO problems has received a growing amount of attention over the years. In approximate methods we sacrifice the guarantee of finding optimal solutions for the sake of getting good solutions in a significantly reduced amount of time.

Classic approximate methods typically lend themselves to classification in constructive methods and local search methods. Constructive methods build a complete solution putting together distinct solution components. According to [BR03], they are thought to be faster but also return lower quality results when compared to local search algorithms. Local search approaches try to systematically cover the solution space by starting at some initial point and then replacing the current solution with a better one, choosing from the set of neighbors. A neighborhood is defined as the set of solution which can be created from applying a determined set of operators which modify the solution only slightly, to yield a new solution. A definition of neighborhood in presented below.

Definition 3.2 – Neighborhood of solution

A neighborhood structure is a function $N:S\rightarrow 2^S$, that assigns to every $s\in S$ a set of neighbors $N(s)\subseteq S$.

A significant amount of work has been put into a new category of approximate algorithm, which tries to wrap basic heuristic knowledge in higher level frameworks, with the aim of efficiently and effectively exploring the search space. These methods are called metaheuristics, and are discussed in the following section.

3.2 Metaheuristics in Optimization

2 As minimization of a function *f* can be seen as maximization of symmetric function *-f*, the chapter focuses only on minimization, without loss of generality.

The term metaheuristic, first introduced by Glover [Glo86], derives from the composition of two Greek words: *heuriskein*, which means "to find", together with the prefix *meta* which means "above" or "beyond". The term describes solution methods that mix higher level strategies with local improvement procedures in order to escape from local optima and to perform a robust search of a solution space. Today it refers to a broad class of strategies for optimization and problem solving. Metaheuristics encompass techniques that range from basic local search procedures to complex learning processes. They are often inspired by analogies (biological, physical, ethological, etc) with real phenomena, but all with the same intent: to guide the search process.

Metaheuristics are defined at a high level of abstraction, making them problem-agnostic, although, at lower levels of abstraction, problem-specific knowledge may be used and is often needed. They are approximate and usually non-deterministic.

Metaheuristics may use forms of memory in order to benefit from acquired search experience. They are, to some extent, stochastic as they incorporate randomized processes in order to promote diversification, to increase robustness and to counter the huge number possibilities.

Despite the fact that even a straightforward implementation and application of a metaheuristic to a given problem instance is usually able to get fairly good results, some degree of customization tends to be required to obtain optimal performance from the methods. Both adding domain specific knowledge to the lower level procedures of the metaheuristic as well as the effective tuning of parameters are thought to be valid strategies to obtain performances that easily improved on canned metaheuristic solutions.

The techniques described here are all based upon the idea of choosing a starting point and then altering one or more variables in an attempt to increase the fitness or reduce the cost. The various approaches have the following two key characteristics [Hop01]:

1. Whether they are based on a single candidate or a population of candidates : Some of the methods to be described, such as hill-climbing, maintain a single "best solution so far" which is refined until no further increase in fitness can be achieved. Genetic algorithms, on the other hand, maintain a population of candidate solution. It is the overall fitness of the population which is improved after each iteration.
2. Whether new candidates can be distant in the search space from the existing ones : Methods such as hill-climbing take small steps from the start point until they reach either a local or global optimum. To guard against missing the global optimum, it is advisable to repeat the process several times, starting from different points in the search space. An alternative approach, adopted in genetic algorithms and simulated annealing, is to begin with the freedom to roam around the whole of the search space in order to find the regions of highest fitness. This initial exploration phase is followed by exploitation, i.e., a detailed search of the best regions of the search space identified during exploration. Methods, such as genetic algorithms, that use a population of candidates rather than just one allow several regions to be explored at the same time.

The next section group the presented metaheuristics into two groups according to the characteristics described above:

- *Trajectory-based metaheuristics*, which usually use a single-candidate solution, but comprises methods which only exploit locally and those which combine the exploration and exploitation referred above;
- *Population-based metaheuristics*, which maintain a set of candidates, and usually explore the solution space freely, in order to escape local optima.

3.3 Trajectory-based Metaheuristics

3.3.1 Pure Random Search

Pure random search is the simplest global random search algorithm. It consists of taking a sample of *n* independent random points and evaluating the fitness function for each of them. Not only very simple to implement, it is often used as a benchmark for comparing properties of other global optimization algorithms [Zhi07].

3.3.2 Hill-Climbing

The name hill-climbing implies that optimization is viewed as the search for a maximum in a fitness landscape. However, the method can equally be applied to a cost landscape, in which case a better name might be valley descent. It is the simplest of the optimization procedures described here. The algorithm is easy to implement, but is inefficient and offers no protection against finding a local minimum rather than the global one. From a randomly selected start point in the search space, i.e., a trial solution, a step is taken in a random direction. If the fitness of the new point is greater than the previous position, it is accepted as the new trial solution. Otherwise the trial solution is unchanged. The process is repeated until the algorithm no longer accepts any steps from the trial solution. At this point the trial solution is assumed to be the optimum. As noted above, one way of guarding against the trap of detecting a local optimum is to repeat the process many times with different starting points.

3.3.3 Simulated Annealing

Originally described by Kirkpatrick in [KGV83], Simulated Annealing (SA) tries to emulate the way in which a metal cools and freezes into a minimum energy crystalline structure (the annealing process) and compares this process to the search for a minimum in a more general system.

At that time, it was well known in the field of metallurgy that slowly cooling a material (annealing) could relieve stresses and aid in the formation of a perfect crystal lattice. [KGV83] realized the analogy between energy state values and objective function values, creating an algorithm that emulated that process.

The SA algorithm tries to escape local optima by allowing the search to sometimes accept worst solutions with a probability (*p*), which decreases along with the temperature of the system (*t*). In this way, the probability of accepting a solution that resulted in a certain increase in the objective function (Δ*f*), at a certain temperature, would be given by the following formula described in the original paper by [KGV83]:

$$p(\Delta f, t) = \begin{cases} e^{\frac{\Delta f}{t}}, \Delta f \leq 0 \\ 1, \Delta f > 0 \end{cases}$$

Observing the formula, it is clear that downhill transitions are possible, with the probability of them occurring decreasing with height of the hill and inversely related to the temperature of the system.

In order to implement the SA algorithm, the initial temperature of the system, and that temperature is going to be lowered, still has to be decided. The slower the temperature is decreased, the greater the chance an optimal solution is found.

Most times a reasonably good cooling schedule can be achieved by using an initial temperature (T_0), a constant temperature decrement (α) and a fixed number of iterations ate each temperature. These kind of cooling schedules are called fixed schedules. The problem with these schedules is that it is often impractical to calculate the ideal values for T_0 and α .

3.3.4 Tabu Search

Tabu search is based on the premise that problem solving, in order to qualify as intelligent, must incorporate adaptive memory and responsive exploration. An analogy provided by [GL07] is that of mountain climbing, where the climber must selectively remember key elements of the path traveled (using adaptive memory) and must be able to make strategic choices along the way (using responsive exploration).

Algorithm 3.1 shows the outline of a simplified Tabu Search algorithm.

```
P ← GenerateInitialSolution()
s* ← s
TabuList ← Ø
while termination conditions not met do:
Ñ(s) ← $\{z \in N(s) : z \notin TabuList \vee z \text{ is allowed by aspiration}\}$
s ← BestOf(Ñ)
if f(s) < f(s*) then
    s* ← s
UpdateTabuList
end while
```

Algorithm 3.1: Tabu Search algorithmic outline

3.4 Population-based metaheuristics

Population-based metaheuristics deal, in every iteration of the algorithm, with a set (i.e. a population) of solutions, usually dubbed individuals, rather than a single solution. The final outcome of such an approach is also a population which implies that, if the problem has a single optimum, population members should converge to it, but if the problem has multiple optima, a population has a better chance at capturing each of them as is shown is Figure 3.1.

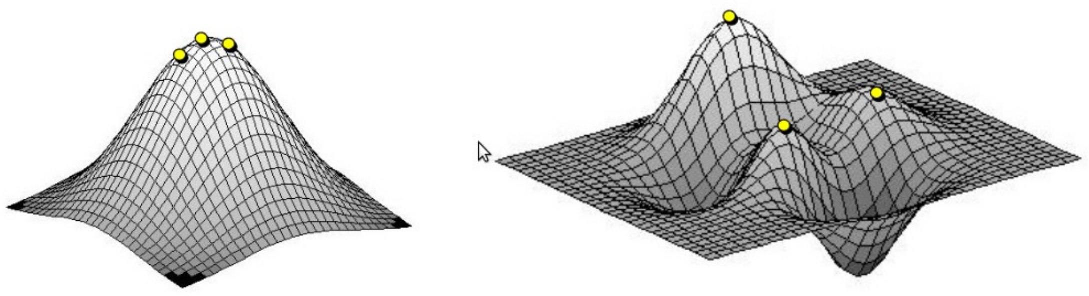

Figure 3.1: Population-based results in single optimum and multiple optima problems [ETH05].

3.4.1 Evolutionary Computation

Evolutionary Computation (EC) is the family of algorithms which share the metaphor of natural evolution, loosely adapted from the field of biology. The underlying concept is that, given an initial population, by selecting only the fittest elements to survive and reproduce, one should expect that each new generation of offspring generates fitter individuals. With this principle in mind a series of methods have been developed, most notably: Genetic Algorithms (GA), Evolution Strategies (ES) and Evolutionary Programming (EP). All of these methods share at the core the following approach: a population of solutions, each one of them having a certain fitness (calculated by evaluating the objective function for each solution), to whom a series of probabilistic operators, like mutations, selections and recombinations, are applied. By using these kinds of stochastic solution manipulation, innovative solutions can be unlocked which explore the solution space in a new way, eventually escaping local optima. Manipulation is generally followed by selection: in which only a subset of the individuals, the ones which exhibit better fitness values, is allowed to continue reproducing.

Algorithm 3.2 shows the outline of a standard EC method ([BR03]).

```
P ← GenerateInitialPopulation()
Evaluate(P)
while termination conditions not met do:
P' ← Recombine(P)
P'' ← Mutate(P)
Evaluate(P'')
P ← Select(P'' ∪ P)
end while
```

Algorithm 3.2: Evolutionary Computation algorithmic outline [BR03]

As has already been stated, the three different approaches to evolutionary computing share the same basic structure. The main differences between them lie in their objectives, the way their population is coded and the way they use the different evolutionary operators.

EP has been initially developed having in mind machine intelligence. Its main particular characteristic is the fact that solutions are represented in a form that is tailored to each problem domain. EP tries to mimic evolution at the level of reproductive populations of species, and recombinations do not occur at this level, so EP algorithms seldom use it.

On the other hand GAs use a more domain independent representation (normally bit strings). The main problem with GA is how to code each solution into meaningful bit strings. Its advantages are that mutation and recombination operators are easily implemented as bit flips (mutations) and string cuts followed by concatenations (recombination).

The main difference between ES and the other two methods just discussed is the fact that selection in ES is deterministic (the worst N solutions are discarded) and that ES uses recombination as opposed to EP.

For the interested reader, a more detailed taxonomy of EC algorithms is available in [CCH99].

3.4.2 Particle swarm optimization

According to its authors in [KE95], particle swarm optimization (PSO) has its roots in two main component methodologies. Perhaps more obvious are ties to artificial life in general and bird flocking, fish schooling and swarming theory in particular. It is also related, however, to evolutionary computation, and has ties to both genetic algorithms and evolutionary programming. The basic underlying concept is the way how social learning influences our beliefs and behaviors.

The particle swarm simulates this kind of social optimization. For a given problem, with a known fitness function a population of individuals, or particles, defined as random solutions for the problem is initialized. An iterative process to improve these candidate solutions is set in motion. The particles iteratively evaluate the fitness of the candidate solutions and remember the location where they had their best success. The individual's best solution is called the particle best or the local best. Each particle makes this information available to their neighbors. They are also able to see where their neighbors have had success. Movements through the search space are guided by these successes, with the population usually converging, by the end of a trial, on a problem solution better than that of non-swarm approach using the same methods [Wik09d].

The main algorithmic characteristic of this method, is that of considering each individual as a particle, with a given position and velocity at each time. It is the velocity that affects its displacement throughout the search space, and also serves as entry point for social influence. At each iteration, the velocity of each particle is influenced by the global optimum (social component) as well as the particle's own local optimum (cognitive component).

Algorithm 3.2 shows the outline of a simple PSO formulation.

```
    $\vec{x}_i$ ← GenerateInitialParticleSolutions()
    $\vec{v}_i$ ← GenerateInitialParticleVelocities()
InitializeGlobalBestSolution( $\vec{\hat{g}}$ )
InitializeLocalBestSolutions( $\vec{\hat{x}}_i$ )
while termination conditions not met do:
    Evaluate( $\vec{x}_i$ )
    $\vec{\hat{g}}$ ← BestOf( $\vec{x}_i$ )
    Update each particle's velocity using:
            $\vec{v}_i = \omega \vec{v}_i + c_{cognitive} \vec{r}_{cognitive} \circ (\vec{\hat{x}}_i - \vec{x}_i) + w_{social} \vec{r}_{social} \circ (\vec{\hat{g}} - x_i)$ ,
    where $\vec{r}_{social}$ and $\vec{r}_{cognitive}$ are random vectors created for each particle and
used to apply the social and cognitive constant.. The $\circ$ operator represents the
Hadamard matrix multiplication operator;
    Update each particle's position, applying the velocity using: $\vec{x}_i = \vec{x}_i + v_i$ ;
return $\vec{\hat{g}}$ .
```

Algorithm 3.3: Particle Swarm Optimization algorithmic outline [BR03]

3.5 Constraint Programming

Constraint Programming is a programming paradigm, where the programming is done by defining variables with relationships in the form of constraints. This differs from traditional programming paradigms, including object oriented programming, which provide little support for specifying constraints among programmer-defined entities. In the traditional programming paradigms it is the role of the programmer to explicitly state and maintain such relationships. Using a Constraint Programming approach the programmer only need to state this kind of relationships once.

The CP paradigm is closely related to Constraint Satisfaction Problems. Using CP the programmer describes the combinatorial aspects of the program as a CSP. The CSP is then solved by a Constraint Solver.

Constraint programming has been successfully applied in numerous domains. Recent applications include computer graphics (to express geometric coherence in the case of scene analysis), natural language processing (construction of efficient parsers), database systems (to ensure and/or restore consistency of the data), operations research problems (like optimization problems), molecular biology (DNA sequencing), business applications (option trading), electrical engineering (to locate faults), circuit design (to compute layouts), etc.

The nature of the matching problem suggests itself the usage of this paradigm for solution generation, as each solution must comply with all the conditions specified by the members. More detail on this can be found in chapter 4 and 5.

3.5.1 Constraint Satisfaction Problems

A finite constraint satisfaction problem is composed by a finite set of variables, each of which with an associated finite domain of possible values and a set of constraints that restricts the values the variables can simultaneously take.

The *domain* of a variable X_i is the set of all possible values that can be assigned to Xi . The domain of X_i will be denoted DX_i .

An *assignment* is a binary relation between a variable and a value in the variable's domain. An assignment of a value to a variable is denoted $<X_i, v_i>$.

A *compound assignment* is a simultaneously assignment of values (possibly empty) to a set of variables. This is denoted ($<X_{1,} v_1>, \ldots, <X_n, v_n>$).

A *constraint* is a restriction of the values that a set of variables can simultaneously be assigned. To denote a constraint on the set of variables S , C_S will be used. Constraints are categorized by the number of subject variables.

A unary constraint is a constraint which has exactly one subject variable e.g. $X_i > 0$.

A binary constraint is a constraints which has exactly two subject variables e.g. $X_i > X_j$.

A *N-ary* constraint is a constraint which has exactly *N* subject variables. Unary and binary constraints can be visualized using undirected graphs.

Satisfies is used as a binary relation between an assignment or compound assignment and a constraint. If the compound assignment *CA* contains a set of variables *S* that are the subject of a constraint C_S , then *CA* satisfies C_S if and only if *CA* is not in conflict with *C*.

A more formal definition of a CSP is now possible:

Definition 3.3: Constraint Satisfaction Problem ([])

A Constraint Satisfaction problem can de defined as a triple (X, D, C)

- ***X*** is the CSP's finite set of variables, i.e., $X = X_{1,} \ldots, X_n$;
- ***D*** is a function which maps every variable in X to a domain of possible values;
- ***C*** is a finite (can be empty) set of constraints on a subset of variables in X. C can be seen as a set of sets of possible compound assignments.

Solving a CSP implies assigning each variable a value of its respective domain, creating a compound assignment containing all the variables X which satisfies all of the constraints C. A solution tuple for a CSP is a compound assignment containing all the variables of the problem, where all the constraints are satisfied. A CSP is said to be *satisfiable* if a solution tuple exists for it.

A common approach to solving CSPs, and one that has motivated significant research work is that of search-based methods [Tsa93]. Many of the search algorithms discovered derive from the same base algorithm called Backtracking. Techniques include constraints propagation and local search. These techniques can be used on problems with nonlinear constraint. Variable elimination and the Simplex algorithm are used for solving linear and polynomial equations and inequalities [Wik09e]. For further background on CSP solving methods please refer to [RBW06] and [Bar99].

3.5.2 Constraint Programming Libraries

The constraint programming paradigm can be used together with a number of other paradigm. It is common to embed the ability to declare constraints in an host language. According to [Wik09e] the first host languages used were logic programming languages, so the field was initially called constraint logic programming. The two paradigms still share many important features, like logical variables and backtracking. Today most Prolog implementations include one or more libraries for constraint logic programming.

Nonetheless, the need to easily integrate the matching system produced by this work, with a separate Web application suggested the need for a language that was dynamic, to allow the necessary interactive experimenting, had a strong base library to facilitate integration with the Web system so that the overall development time was shortened. The choice was narrowed down to Ruby, since the existing Web application had been built using Ruby on Rails [TH09] and Python due to its very mature base library and extensive multi-paradigm support (which would make for smoother connection between the object-oriented and the constraint programming paradigms used).

The most significant available constraint programming solutions for these dynamic languages were [Wik09e]:

- Cassowary constraint solver, open source project for constraint satisfaction (accessible from C, Java, Python and other languages);
- logilab-constraint, open source constraint solver written in pure Python with constraint propagation algorithms;
- python-constraint (Python library, GPL);
- Gecode, an open-source portable written in C++ developed as a production-quality and highly efficient implementation of a complete theoretical background. Bindings available for Ruby via Gecode/R;
- Conssolv, a simple constraint solver in pure ruby;

The choice was python-constraint essentially due to its pure Python nature, its small size (circa 1400 library lines of code, including comments and whitespace) and the documentation available in [Nie05]. These characteristics improve the learning experience due to the high readability of the code base and the quality of the documentation, which together make the learning curve less steep, and open the way to future extension.

3.6 Summary

This chapter presented various works related to the subject of discrete variable optimization. A formal definition of this kind of optimization problem was shown, which can later be applied to the optimization problem at hand. A series of optimization techniques, known as metaheuristics, which provide reusable strategies for solving optimization problems were also presented.

The chapter goes on to describe a programming paradigm which is of particular interest for the kind of problem at hand: Constraint Programming (CP). The text discusses what is CP, presents the underlying concept of Constraint Satisfaction Problems and presents technologies to solved them in the context of the current work.

The next chapters will use the concepts of Optimization and CP and apply them to the problem at hand.

4 Loan Request and Investment Offer Combination

Despite the main focus of this work is the optimization component, some aspects regarding the overall project must be detailed to allow complete understanding of the problem. The next section provides further information about the operating model for the platform.

Building the foundation for an online social lending community, requires work in several areas namely:

- building a match optimization system, the primary focus of this work;
- developing a web application, with the associated user experience concerns;
- legal issues arising from establishing this kind of social platform under the local law.

4.1 Platform Operation

The platform is operated by a for-profit organization, which intends to provide alternative financial services, as much as possible, outside the modern banking system. The underlying principle is direct person-to-person lending. The existence of an online hub, which allows people to meet and establish social and financial connections, solves the scale issue with traditional credit communities. As people can join together to provide money for a single borrower, it is possible to finance larger loans without making each individual lender have lend a significant amount of money to the same borrower. This collaborative concept give each investor the opportunity to diversify its investments by choosing different loan types, and only investing the amount of money intended. The illustration in Figure 4.1 depicts final art from the Web application that represents the kind of collaborative and social sentiment that is intended for the platform.

Figure 4.1: Miscellaneous final art from Web platform front-end [Rod09].

4.2 Lenders

Lenders or investors provide the money for the loans, and received the agreed interest through the loan life cycle. Lenders submit investment offers specifying the bid as:

- maximum amount invested;
- minimum interest rate offered.

Investment offers target specific lending markets. Lending markets are groups of similar loan requests, regarding borrower risk and loan duration. Investors are advised to carefully pick their loan markets, according to their investment strategy and to diversify as much as possible.

4.3 Borrowers

Borrowers receive money from loans, and pay it back with the agreed interest along the loan life cycle. Borrowers submit loan requests specifying the loan as:

- minimum amount required;
- maximum interest rate accepted;
- loan duration.

Borrowers are categorized according to the associated credit risk. The credit risk associated with each borrower will be initially determined according to information provided by the member, after validation and evaluation by a third-party rating agency. Other models for risk management are proposed in the Future work section.

4.4 Security and Fraud prevention

Due to the nature of the project, an identity management strategy is required. To make the platform a safe means of interaction for everybody, users are consider to have different access levels:

- *Member* (basic): signed up for the platform, and confirmed e-mail address.
- *Lender* (verified identity) : Same as member, and provided additional personal data (full name, address, tax number, etc.) which should be verified by a specialized third-party.
- *Borrower* (credit rated): Same as lender, and provided additional financial data (housing conditions, existing debt, etc.) which should also be verified and evaluated by a rating agency

4.5 Business Model

The business model of the platform relies on two fundamental sources of income:

- An opening fee, which corresponds to charging for the successful matching of lender and borrower. This fee is charged to borrowers only.
- A commission over the interest received by investors, which corresponds to charging for the processing of borrower payments. This fee is charged to lenders only.
- An expense fee for withdrawing money for the Member's Private Account.

Despite being a for-profit initiative the business model is thought to be value-driven, in that it charges only for the services which actually deliver value to members (the remaining usage of the site, bid placement, account management is free of charge), and is at the same time transparent, in that it charges for the real expenses that the platform operators have to deal with.

Due to business and legal implications, the platform operator itself does not intend to continuously supply money directly or otherwise invest in loans inside the system in the long term, but still may need to it initially to achieve critical mass. In case the platform operators do invest, they will operate under the exact same conditions as lenders.

4.6 Legal issues

The Portuguese law is particularly restrictive in terms of financial operations. Both paying back interest for money deposits, and charging interest for loaned money is restricted to very small amounts outside which a different regulation altogether applies: the "Regime Geral das Instituições de Crédito e Sociedades Financeiras" (General Regime of Credit Institutions and Financial Societies, loosely translated), regulates the conditions under which Banks and other Financial Institutions operate [BdP04]. The legal document specifies the activities which are reserved for each kind of institution, none of which can be performed by other types of organizations.

One of the alternatives considered as the legal framework for the project was creating a new financial institution dedicated to operate the Person-To-Person lending platform. Nonetheless, the stringent conditions imposed to these kinds of institutions in the regulation and the number of subjective criteria used in approving the creation of a new bank or other credit institution made it extremely hard to approach the alternative.

Another model was then designed which consisted of taking the platform operator out of the way, allowing members to directly borrow and lend to each other. In this model the entities involved in the loan contracts are the exclusively members and not the platform operator. In turn, the platform operator and the members are bound by the Terms and Conditions, modeled upon the Zopa Principles [Zopa09b] and the Prosper Terms of Use [Pros09]. These terms essentially specify how the users move money to and from the platform, and how they accept being charged for the matching service done by the system. The terms also clarify the procedures upon payment default by any borrower, and the steps which the platform operators take, from the initial notifications, to the assignment of a credit recovery agency to the collect the missing amount in the name of the lenders.

4.7 Summary

This chapter presented a comprehensive description of the problem, intending to describe the particular formulation of P2P credit, in the context of the project which this work relates to. The chapter defined the core concepts required to understand the loan request and investment offer combination platform, and presented them from different perspectives, ranging from the more operational platform management issues to more legal-oriented considerations.

The next chapter introduces a formal definition of the presented problem, and proposes a corresponding solution.

5 Proposed Solution

5.1 Formal Problem Definition

The problem of combining loan requests and investment offers, can be considered as an optimization problem, taking the decision variables as the rates at which loans are matched, together with the amounts involved. The constraints would be set by the conditions specified by the members, when placing their terms for intended rates and amounts. The objective function would take into account the stated goals of the platform: to maximize the satisfaction of both borrowers and lenders, while contributing for the platform's profitability.

More formally the problem $P=(S,f)$ can be defined as a generic optimization problem by specifying:

- the set of parameters:
  - N is the number of lenders participating;
  - M is the number of borrowers participating;
  - $Rmin_i$ is the minimum rate at which lender i is willing to lend its money;
  - $Rmax_j$ is the maximum rate at which borrower j is willing to borrow money;
  - $Amax_i, Amin_i$ are the maximum and minimum amounts of money, lender i is willing to lend;
  - $Amax_j, Amin_j$ are the maximum and minimum amounts of money, borrower j wants to borrow.
- the set of the decision variables $X=\{r_{11}, a_{11}, r_{12,} a_{12,} \dots, r_{ij}, a_{ij}, \dots r_{NM}, a_{NM}\}$, where:
  - r_{ij} represents the rate at which lender i lends to borrower j and
  - a_{ij} represents the amount of money lender i lends to borrower j;

- the variable domains defined for each instantiation of the matcher where:
  - $D(r_{ij})$ is the interval $[0,1]$, where the maximum allowed rate is 100%, due to business considerations;
  - $D(a_{ij})$ is the interval $[0,A_j]$, where A_j represents the maximum amount requested by borrower j.
- the constraints which result from the members terms, which consist of:
  - C1: $\frac{\sum_{i=1}^{N} a_{ij} * r_{ij}}{\sum_{i=1}^{N} a_{ij}} \leq Rmax_j \forall j \in \{1,\ldots,M\}$, i.e., the overall loan rate for borrower j must be less than or equal to the proposed maximum rate;
  - C2: $\frac{\sum_{j=1}^{M} a_{ij} * r_{ij}}{\sum_{j=1}^{M} a_{ij}} \geq Rmin_i \forall i \in \{1,\ldots,N\}$, i.e., the overall investment rate for lender i must be greater than or equal to the proposed minimum rate;
  - C3: $Amin_i \leq \sum_{j=1}^{M} a_{ij} \leq Amax_i \forall i \in \{1,\ldots,N\}$, i.e., the total amount invested by lender i must be less than or equal to the amount offered;
  - C4: $Amin_j \leq \sum_{i=1}^{N} a_{ij} \leq Amax_j \forall j \in \{1,\ldots,M\}$, i.e., the total amount received by borrower j must be less than or equal to the amount requested;
- a function $f: D(r_{11}) \times D(a_{11}) \times \ldots \times D(r_{ij}) \times D(a_{ij}) \times \ldots \times D(r_{NM}) \times D(a_{NM}) \rightarrow \Re$, which computes the utility to maximize, defined as mapping the decision variable's domains into a decimal value.

The solution space is then the set of valid solutions, or:

- $S = \{ s = \{\ldots(r_{ij}, R_{ij}), (a_{ij}, R_{ij}), \ldots\} : R_{ij} \in D(r_{ij}) \wedge A_{ij} \in D(a_{ij}) \wedge s \text{ satisfies all the constraints} \}$.

5.2 Utility functions

Defining an utility function is a sensible issue and has a great impact on the overall business quality of the solutions. Nonetheless validation of such a function presents a non-trivial challenge, which may fit better the field of efficient markets or an even more social science domain, rather than the field of computer science. This is why the comparative study of different utility functions was deemed out of scope for this work, and although two utility functions are presented, for illustration purposes, only the latter will be subject to experiments.

5.2.1 Dissatisfaction cost

A simple approach for evaluating solutions was first proposed, which tried to make sure that all the members were as close as possible to their specified terms. The underlying belief is that, for one, people state their limits within their comfort zone (are comfortable with a deal closing on the limit) and that, on the other hand, as everyone nears the limit of what they specified, they will be accepting deals which are better for the community as a whole, since they are being more competitive. This is obtained considering the sum of the squared margins obtained by each member, and minimizing it.

$$DissastisfactionCost=\sum_{i=1}^{N}\sum_{j=1}^{M}(r_{ij}-Rmin_i)^2+(Rmax_j-r_{ij})^2$$

5.2.2 Tight margin utility

The *tight margin utility* function tries to reward profitable and fair solutions. Its base rationale is that people specify their limits as their baseline, from which they expect some sort of profit margin. To achieve this, the function takes into account three key factors:

- member profit: how much did the members benefit from the solution, what was their margin;
- fairness: how evenly distributed are these benefits;
- demands satisfaction: how well where the demands fulfilled in the solution, this is particularly significant for the platform's profitability which comes from the volume of money successfully matched.

More formally the tight margin utility is computed by:

$$TightMarginUtility=(k_{MemberMargin}*MemberMargin)*(k_{Tightness}*Tightness)*(k_{FulfillmentRate}*FulfillmentRate)$$

where:

- Member Margin represents the sum of margins obtained by each member. Here margin is understood as the difference between the initial terms the member specified and the actual deal conditions proposed by the solution. This can be computed as:

$$MemberMargin=\sum_{i=1}^{N}(r_{ij}-Rmin_i)+\sum_{j=1}^{M}(Rmax_j-r_{ij})$$

- Tightness represents how close together are the individual gains of each member, a solution where some member make are highly favored in the detriment of others exhibits low tightness where a solution where margins are homogeneous is considered tight. Tightness can then be computed as:

$$Tightness=\frac{1}{\sigma(\{(r_{ij}-Rmin_i):i\in[1,N],j\in[1,M]\}\vee\{(Rmax_j-r_{ij}):i\in[1,N],j\in[1,M]\})}$$

- Fulfillment Rate indicates the relevance of the amount that was matched, in the light of the total amounts of money which were in stake. A fulfillment rate of 1 or 100% means that all the available money was invested and all the money requested was fulfilled. This can be computed as:

$$FulfillmentRate = \frac{2 * TotalMatchedAmount}{(TotalOfferedAmount + TotalRequestedAmount)}$$

- Each of the factors is weighed according to a specific parameter ($k_{MemberMargin}$, $k_{Tightness}$ and $k_{FullfilmentRate}$), which allows to fine tune the utility function for the kind of results intended.

5.3 Proposed Solution

Formulating the problem as a generic optimization problem as stated in 5.1, allows the application of a number of meta-heuristics, and to directly compare results in order to propose an approach for the final platform matching system.

The designed system consists of three components: a constraint solving tool; the optimization suite and an adapter for the suite that uses the constraint solving library to solve the specific problem. Figure 5.1 illustrates these blocks and shows their high-level interactions.

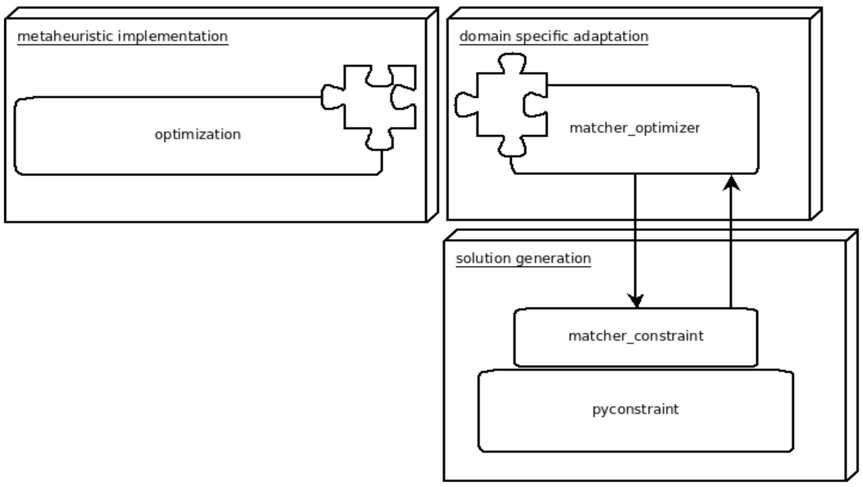

Figure 5.1: Proposed Solution Architecture Overview

5.3.1 Constraint library extensions

The first problem to address was that of efficiently producing feasible solutions, complying with all the existing constraints. As discussed in section 3.5, the constraint programming paradigm is a natural choice for this class of problems. The *python-constraint* library, described in section 3.5.2, was chosen and extended to match the specific system requirements. The extensions made to the base library intend both to provide additional constraints missing from the original library and to add functionality required by a large number of meta-heuristics which would depend on the solution generation mechanism to explore the solution space. The added constraints where necessary for the correct specification of the optimization problem, as formulated earlier. This was the case with providing constraints on weighted averages, as required by the member rate terms, both for lenders and borrowers, which are weighed by the amount involved in each contract. From the optimization standpoint, there were low-level functionalities that required close interaction with the solution generation strategy, yet were relevant for solution space exploration. This was the case with a an API method for retrieving the valid neighborhood for a given solution, as well a way for obtaining the closest feasible solution to a specified solution, even if invalid. These operations could not be efficiently implemented at a higher-level or would otherwise require replicating a large part of the solution generator behavior in the optimization adapter layer.

5.3.2 Optimization framework

The optimizer framework provides strategies for solving optimization problems. An Object-Oriented (OO) framework is a reusable design together with an implementation [JF88]. The design represents a model of an application domain or a pertinent aspect thereof, and the implementation defines how this model can be executed, at least partially [Rie00]. Although the concept may be similar to object-oriented libraries, the key difference relies in the control of program flow, which is not imposed by the caller, as with libraries, but by the framework itself [Wik09f, Rie00]. It is commonplace to find frameworks which use Inversion of Control (IoC) [Fow04] techniques to define the basic operation flow, providing well-defined entry-points for the use-case specific code.

The optimizer framework specifies the abstract behavior of an optimizer object, and provides concrete implementations of popular search meta-heuristics such as those described in chapter 3. The problem specific use-client code needs only to take care of what is really domain specific: the solution generation, solution evaluation and solution visualization. The components which provide these adaptation services are then injected into the optimizer object which wraps the meta-heuristic algorithm. This dependency injection design decision aims to reduce coupling, allowing to easily apply the same problem-specific adapters to other meta-heuristic optimization objects and to reuse the same meta-heuristic optimization objects on different problem-specific adapters. The optimization classes can also be extended to provide new meta-heuristic techniques, which will work with existing problem adapters as long as the framework contract is kept: typically only the core optimization routine needs to be implemented as the abstract Optimizer class already provides flow-control tools such as termination criteria detection, which can also be extended.

5.3.3 Matcher optimization use-client

The problem-specific adapter bridges the two components described above. An optimization adapter is placed inside the IoC framework, providing solution generation, evaluation and visualization services to the Optimizer classes. The adapter produced for the current problem uses the solution generation infrastructure introduced above, to model the problem, as it was formulated, offering wrapping in generic solution generation and manipulation services. The adapter also provides a solution evaluation service, which implements the utility function; and a simple character-based solution output service to aid development. Although the solution generator component and the underlying constraint solving libraries and necessarily coupled, it would be easy to replace it with another generator using any other technique. This holds true for the remaining components of the adapter, which give it a degree of flexibility.

5.4 Summary

This chapter builds on the qualitative presentation of the matching problem, provided by earlier chapters, and describes a quantitative approach to solving it. A formal definition is proposed, specifying all of the typical components of an optimization problem. Having adequately formulated the problem, a solution design is presented which includes a reusable framework for optimization problems, and a problem specific use-client adapter for this particular matching problem. The next chapter will present how this design was executed, and how the components operate and interact, with additional detail.

6 Implementation

This chapter presents the low-level details relating to the solution which was implemented to address the problem discussed in the previous chapters.

The diagram in figure 6.1 provides a detailed vision of the solution architecture, with additional detail on the components discussed in section 5.3. The figure depicts the developed components in white background, and re-used components in grey background. Below is a breakdown of the responsibility of each of the classes involved and the main interactions that take place in each component.

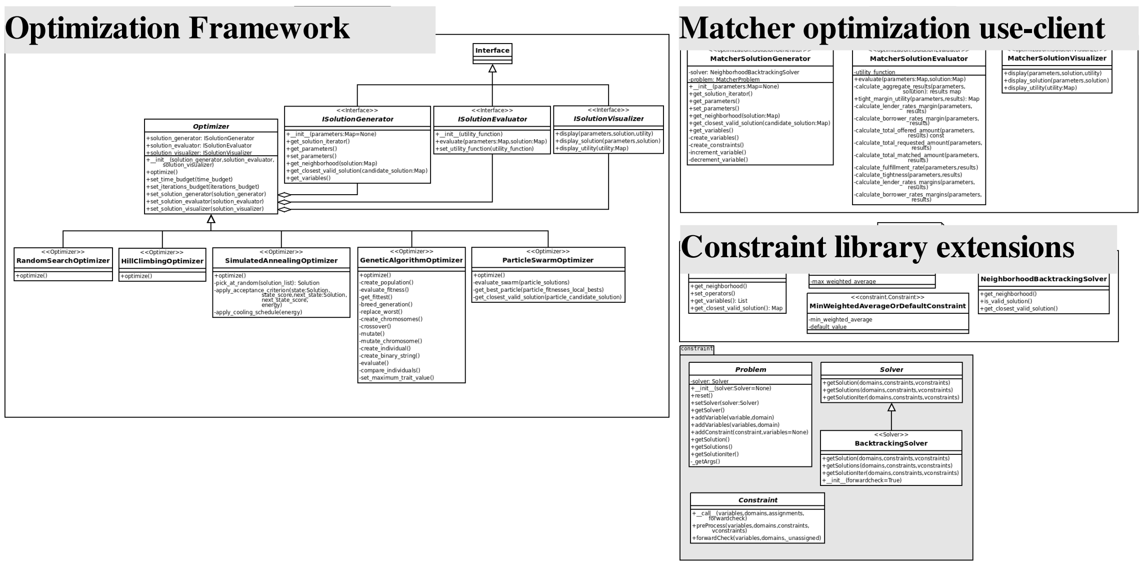

Figure 6.1: Proposed Solution Detailed Architecture.

6.1 Constraint library extensions

Figure 6.2 illustrates how the constraint library extensions module extends the *python-constraint* library, depicted in grey background. This module provides additional functionality which is relevant for the current problem domain. The features include neighborhood generation, useful for several search meta-heuristics, getting approximate valid solutions from an initial solution, whether or not valid, and new constraints to enforce the members' terms as formulated in section 5.1.

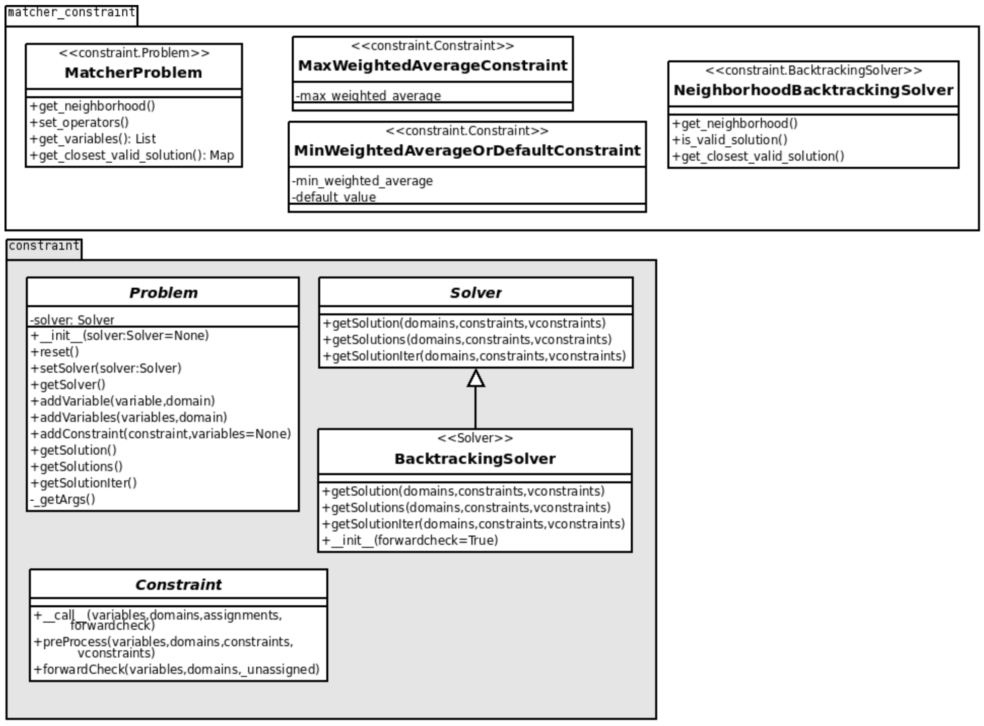
matcher_constraint
<<constraint.Problem>>
MatcherProblem
+get_neighborhood()
+set_operators()
+get_variables(): List
+get_closest_valid_solution(): Map
<<constraint.Constraint>>
MaxWeightedAverageConstraint
-max_weighted_average
<<constraint.Constraint>>
MinWeightedAverageOrDefaultConstraint
-min_weighted_average
-default_value
<<constraint.BacktrackingSolver>>
NeighborhoodBacktrackingSolver
+get_neighborhood()
+is_valid_solution()
+get_closest_valid_solution()
constraint
Problem
-solver: Solver
+__init__(solver:Solver=None)
+reset()
+setSolver(solver:Solver)
+getSolver()
+addVariable(variable,domain)
+addVariables(variables,domain)
+addConstraint(constraint,variables=None)
+getSolution()
+getSolutions()
+getSolutionIter()
-_getArgs()
Solver
+getSolution(domains,constraints,vconstraints)
+getSolutions(domains,constraints,vconstraints)
+getSolutionIter(domains,constraints,vconstraints)
<<Solver>>
BacktrackingSolver
+getSolution(domains,constraints,vconstraints)
+getSolutions(domains,constraints,vconstraints)
+getSolutionIter(domains,constraints,vconstraints)
+__init__(forwardcheck=True)
Constraint
+__call__(variables,domains,assignments, forwardcheck)
+preProcess(variables,domains,constraints, vconstraints)
+forwardCheck(variables,domains,_unassigned)

Figure 6.2: Constraint library and extensions architecture.

Below is a description of the main object classes introduced, and the purpose they serve.

6.1.1 Problem wrapper

The `MatcherProblem` class extends `constraint.Problem`, the *python-constraint* façade class used to define a problem and retrieve solutions from a solver [Nie05].

- Introduces the `getNeighborhood` method which accepts a solution, in the form used throughout the constraint library, and returns a list of the adjacent solutions produced using the `Solver` object's `getNeighborhood` and the `MatcherProblem`'s set of operators. Operators are functions which generate new solutions from existing ones, allowing to navigate within the solution space.
- The problem wrapper also provides the `getClosestValidSolution` method which returns a valid solution, closest to a specified solution, whether or not valid. The strategy consists of search for adjacent solutions which are valid, using the operators, and if not found by gradually replacing values in each of the variables, always trying to change as little as possible. This feature is relevant for search meta-heuristic that potentially generate invalid solutions which must be converted to valid ones, this is the case with evolutionary approaches such as genetic algorithms.

6.1.2 Solver

A new solver was developed that provided new solution space exploration capabilities, The `NeighborhoodBacktrackingSolver`: extends the `constraint.BacktrackingSolver` provided by *python-constraint*, which provides a constraint solver with backtracking capabilities. This subclass introduces the `getNeighborhood`, the `isValidSolution` and the `getClosestValidSolution` methods used by the new `MatcherProblem`:

- `getNeighborhood` returns the neighborhood of solutions, for a specified solution. It does so by applying the provided operators to all the existing variables, to generate all the adjacent solutions, in the solution space. After each operator is applied, each solution is tested for validity, according to the existing constraints. If the solution is not valid, an alternate solution is generated by adjusting one of the variables. The set of valid generated solutions is returned as a list, corresponding to the initial solution's neighborhood. This method uses an internal `getNeighborhoodGenerator` function, which returns a python generator object [Sche01], that can be used to generate new solutions on demand.
- `isValidSolution` checks the provided solution against all the existing constraints, verifying if all hold.
- `getClosestValidSolution` returns a valid solution which is obtainable by applying the smallest number of transformations to the given solution, whether or not valid. The method leverages the `getNeighborhoodGenerator` method and simply returns the first solution yield by the generator.

6.1.3 Constraints

In the addition to maximum and minimum sum constraints, two new constraints where developed for the constraint library: maximum weighted average constraint and minimum weighted average constraints. Both of these constraints are relevant for restricting the average rate a member gets in the final match result, since the rate is weighed by the amount of money in each match. Below is a detailed description of each constraint:

- `MaxWeightedAverageConstraint`: concrete subclass of `constraint.Constraint`, the *python-constraint* abstract class for constraints [Nie05]. Specifies a new constructor which accepts the value of the maximum weighted average a set of variables in the problem can assume. Provides an implementation of the `__call__` which allows instances of this class to be called as functions [PSF09a], and is used to implement the constraint checking code. The checking algorithm is described below.

```
1. Initialize running weighted average and running weight sum;
2. Group the variables list, in $\{w_1, v_1, w_2, v_2, \ldots, w_N, v_N\}$ order, into a list pairs, in the
   order $\{(w_1, v_1), (w_2, v_2), \ldots, (w_N, v_N)\}$ , where $w_i$ is the i-th weight variable and
   $v_i$ is the i-th value variable.
3. For each weight-value pair:
    1. If both weight and value variables have been assigned:
```

1. *Update the running weighted average:* $\bar{v}_i = \dfrac{(\sum_{j=0}^{i-1} w_j * \bar{v}_{i-1} + w_i * v_i)}{\sum_{j=0}^{i-1} w_j + w_i}$;
2. *Update the running weight sum* $w_i = \sum_{j=0}^{i-1} w_j + w_i$;

4. *If the running weighted average is below or equal to the maximum specified value, the constraint holds;*
5. *Else the constraint does not hold, and that is signaled to the calling function.*

- `MinWeightedAverageOrDefaultConstraint`: another concrete subclass of `constraint.Constraint`, similar to `MaxWeightedAverageConstraint`. The difference relies in the final constraint check, which in this case checks if the running weighted average, at the end of the iteration, is above or equal to the initially defined minimum weighted average or to an initially defined default value, such as 0, as will be shown in the matcher problem.

6.2 Optimization framework

The optimization framework provides a standard design for building solutions for optimization problems as well as actual implementations of optimization strategies. The framework provides a core abstract `Optimizer` class, which defines the behavior of a generic optimizer, as well as concrete implementations of meta-heuristics wrapped in `Optimizer` subclasses. The framework also includes a set of interfaces[3] for the use-client objects to implement, that specify and document the API contract for handling solution generation, solution evaluation and solution visualization.

3 The term *interface* is used here in the broad context of an abstraction that an entity provides of itself to the outside [Wik09g], not a standard Python construct. Presently there seems to be no consensus in the Python community about a common standard for interfaces, with previous attempts at standardization rejected [Pel09]. Therefore, this work adopted a popular, despite non-standard approach to this problem, which takes from the Zope web application framework [Zope09].

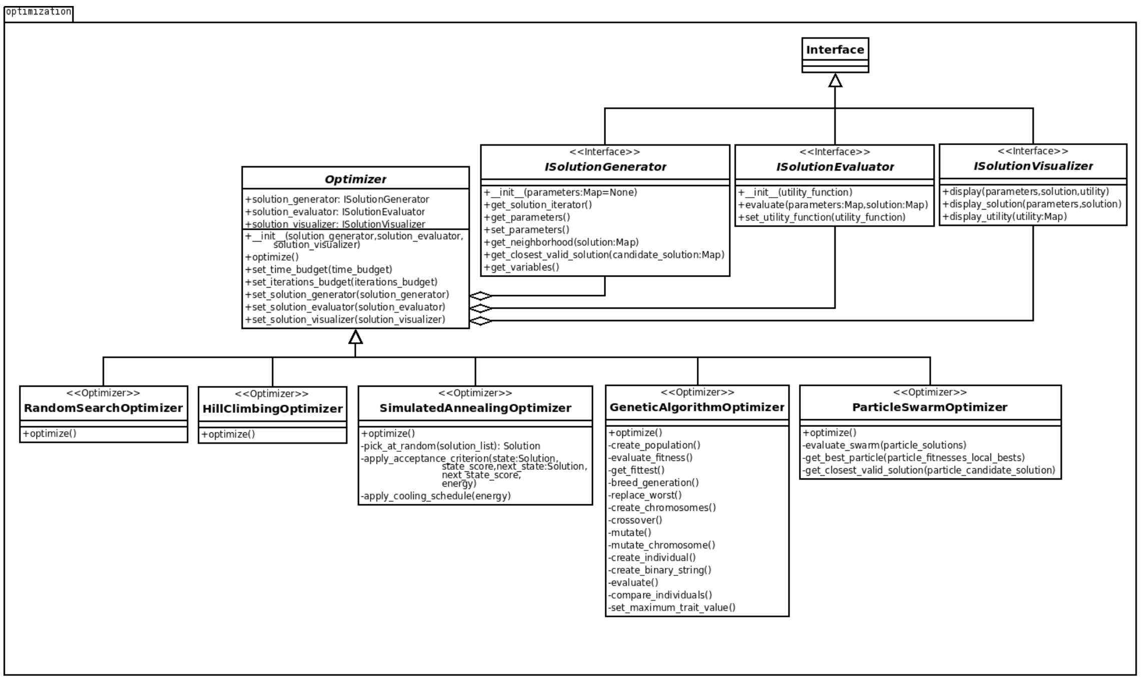
optimization
Interface
<<Interface>>
ISolutionGenerator
+__init__(parameters:Map=None)
+get_solution_iterator()
+get_parameters()
+set_parameters()
+get_neighborhood(solution:Map)
+get_closest_valid_solution(candidate_solution:Map)
+get_variables()
<<Interface>>
ISolutionEvaluator
+__init__(utility_function)
+evaluate(parameters:Map,solution:Map)
+set_utility_function(utility_function)
<<Interface>>
ISolutionVisualizer
+display(parameters,solution,utility)
+display_solution(parameters,solution)
+display_utility(utility:Map)
Optimizer
+solution_generator: ISolutionGenerator
+solution_evaluator: ISolutionEvaluator
+solution_visualizer: ISolutionVisualizer
+__init__(solution_generator,solution_evaluator, solution_visualizer)
+optimize()
+set_time_budget(time_budget)
+set_iterations_budget(iterations_budget)
+set_solution_generator(solution_generator)
+set_solution_evaluator(solution_evaluator)
+set_solution_visualizer(solution_visualizer)
<<Optimizer>>
RandomSearchOptimizer
+optimize()
<<Optimizer>>
HillClimbingOptimizer
+optimize()
<<Optimizer>>
SimulatedAnnealingOptimizer
+optimize()
-pick_at_random(solution_list): Solution
-apply_acceptance_criterion(state:Solution, state_score,next_state:Solution, next_state_score, energy)
-apply_cooling_schedule(energy)
<<Optimizer>>
GeneticAlgorithmOptimizer
+optimize()
-create_population()
-evaluate_fitness()
-get_fittest()
-breed_generation()
-replace_worst()
-create_chromosomes()
-crossover()
-mutate()
-mutate_chromosome()
-create_individual()
-create_binary_string()
-evaluate()
-compare_individuals()
-set_maximum_trait_value()
<<Optimizer>>
ParticleSwarmOptimizer
+optimize()
-evaluate_swarm(particle_solutions)
-get_best_particle(particle_fitnesses_local_bests)
-get_closest_valid_solution(particle_candidate_solution)

Figure 6.3: Optimization framework architecture.

The following sections present the specifics on each of these components.

6.2.1 Abstract Optimizer

The `Optimizer` class holds the search strategy used to optimize the problem, using the provided generator and evaluator. Subclasses of this abstract optimizer template provide the concrete implementations of each meta-heuristic. The abstract class already defines basic termination management code which can be reused or augmented by child classes. Below is a breakdown of the most relevant methods:

- abstract `optimize` method where concrete subclasses start the optimization loop and typically contains the following generic steps:
    1. *Resetting termination conditions;*
    2. *Obtaining parameters from the generators;*
    3. *Initializing the meta-heuristic specific structures;*
    4. *While termination conditions have not been met:*
        1. *Searching for a new solution;*
        2. *Evaluating the new solution;*
        3. *Update the iteration count and the best fitness so far (for termination detection);*
    5. *Return the best solution found.*

- `termination_conditions_met` predicate method indicates if the configured termination conditions have been reached. The abstract `Optimizer` class offers support for the most common termination conditions: time limit (maximum interval since the execution start), iterations limit (defining a maximum number of iterations for the search process) and fitness target (defining a value of the utility function considered satisfactory).
- `reset_termination_conditions` re-initializes the structures used to keep track of the status of the optimization execution which are consulted by the `termination_conditions_met` method.

6.2.2 Random Search Optimizer

`The RandomSearchOptimizer` class implements the pure random search meta-heuristic described in section 3.3.1. Implements the optimize method, using the solution generator object to get each possible solution in the search space. Evaluates each solution, updating the best at each iteration, until termination conditions have been met

6.2.3 Hill Climbing Optimizer

The `HillClimbingOptimizer` class implements the hill-climbing meta-heuristic, described in section 3.3.2. Starts with the first solution retrieved from the generator object, and calculates its neighborhood. Evaluates each node in the neighborhood and picks the best. If the best neighbor node is worse than the current node returns, else calculates the best neighbor nodes neighborhood and enters the next iteration. This procedure is repeated until the termination conditions have been met.

6.2.4 Simulated Annealing Optimizer

The `SimulatedAnnealingOptimizer` class implements the simulated annealing meta-heuristic, relies on the standard solution generator functionality and contains additional helper methods to aid in the overall meta-heuristic execution.

- concrete method `optimize` follows a standard simulated-annealing algorithm, detailed below:

```
1. Reset termination conditions;
2. Initialize state and energy;
3. Initialize best solution;
4. While termination conditions not met:
    1. Pick a neighbor;
    2. Compute the neighbor's score;
    3. If the neighbor is a new best move to it, and store it as best solution;
    4. Else, determine if the solution should still be accepted using
       apply_acceptance_criterion;
    5. Update energy with apply_cooling_schedule;
5. Return the best solution found.
```

- `apply_acceptance_criterion` method implements the stochastic component of the meta-heuristic, accepting values which are not better than the current best but still exploring them for search space comprehension. The probability p of given solution s, is replaced by solution s' in a system with energy E is determined by: $p(s,s',E)=e^{\frac{f(s)-f(s')}{E}}$.
- `apply_cooling_schedule` implements the cooling schedule of the meta-heuristic which, in this implementation, consists of applying a constant factor between 0 and 1 to the current energy.

6.2.5 Genetic Algorithm Optimizer

The `GeneticAlgorithmOptimizer` class implements a variant of this popular evolutionary algorithm, presented in 3.4.1. Below is a description of the flow of the main `optimize` method, and the detailed responsibilities of the auxiliary functions developed.

- The concrete `optimize` method articulates the optimization flow, using the following simplified algorithm (phrasing adapted from [Wik09h]):

```
1. Reset the termination conditions;
2. Choose initial population (create_population);
3. Evaluate the fitness of each individual in the population (evaluate_fitness);
4. While termination conditions not met:
    1. Select best-ranking individuals to reproduce (get_fittest);
    2. Breed new generation through crossover and/or mutation (genetic operations) and give birth to offspring (breed_generation);
    3. Evaluate the individual fitnesses of the offspring (evaluate_fitness);
    4. Replace worst ranked part of population with offspring (replace_worst);
5. Return the best element in the population.
```

- `create_population` uses the optimizer's solution generator to create a *N*-sized list of initial solutions, where *N* is the initial population size parameter.
- `evaluate_fitness` takes in a population list and computes the fitness function for all individuals, returning an evaluated population list, i.e., a list of (individual, fitness score) pairs.
- `get_fittest` takes in an evaluated population list, sorts it according by ascending fitness, and returns the top *K* individuals, where *K* is the reproduction sample size;
- The `breed_generation` takes in an evaluated population list, and generates a set of offspring produced by crossover and mutation operations. The main breeding algorithm is as follows:

1. *Choose the fittest member of the population to pair up with an eventual lonely individual. This is required whenever the K, the reproduction sample size, is odd, since the breeding mechanism used is more strictly nature inspired and uses combinations of two individuals to reproduce;*
2. *For all the pairs of individuals, or couples:*
    1. *Calculate the genetic representation for each individual (`create_chromosomes`);*
    2. *Apply the crossover operator (`crossover`) obtaining the two children's genetic representation;*
    3. *Mutate each child's chromosomes (`mutate`);*
    4. *Create the children individuals from the chromosomes (`create_individual`);*
    5. *Use the solution generator to convert each child into a valid problem solution (genetic manipulation operations tend to generate invalid solutions), by finding the closest valid solution for each child.*
    6. *If no valid solution can be reached by the generator, for the current child, one of the original individuals is used as replacement in the offspring list.*

- `replace_worst` *replaces the bottom n elements of the population, with* $n \in \{1, \ldots, P\}$ *, with the top n elements of the offspring;*
- `create_chromosomes` creates a genetic representation of a solution individual. The approach create a map of binary strings, where each key represents a trait of the individual, in this case a variable of the solution. Each binary string is obtained with the Python `bin` built-in function, recently introduced with Python 2.6.2 [PSF09b], which converts an integer number to a binary string. This binary string is then padded for the standard size. The standard chromosome length is computed for each problem, depending on the domain of each of the decision variables;
- `crossover` combines the chromosomes belonging to two individuals into two new individuals, which composed of different parts of the genetic material of the parent individuals. In this implementation there is a single crossover-point, exactly at the middle of the each individuals' chromosomes.
- `mutate` uses the bit-flipping approach to each of the individual's trait, picking a random bit to flip on each of the solution variable's representation.
- `create_individual` generates an individual based on its genetic material, i.e., creates a solution by converting the binary string representation of the value of each of its variables.

6.2.6 Particle Swarm Optimizer

The `ParticleSwarmOptimizer` class provides a sample implementation of the particle swarm meta-heuristic. As described in section 3.4.2, this population-based algorithm takes advantage of the simulated social interaction between individuals, searching for the best solution while communicating its achievements with a group of neighboring individuals. In this sample implementation the core methods are:

- The concrete `optimize` method, which coordinates the execution of the particle swarm meta-heuristic implements the following algorithm:
    1. *Reset the termination conditions;*
    2. *Initialize the key parameters for the algorithm:*
        1. N *, swarm size, i.e., the number of particles;*
        2. ω *, an inertia constant;*
        3. $c_{social}, c_{cognitive}$ *, constants use to weigh-in the importance give to both the social and cognitive components of the each particle;*
    3. *Initialize the* $\vec{x}_i$ *particle solutions, using the provided solution generator;*
    4. *Initialize the* $\vec{v}_i$ *particle velocities, with the corresponding null-vectors;*
    5. *Initialize the* $\vec{\hat{g}}$ *global best solution, and the global best score;*
    6. *Initialize the* $\vec{\hat{x}}_i$ *local best solutions and the local best scores for each particle;*
    7. *While termination conditions not met:*
        1. *Evaluate the swarm: calculate each particle's fitness using the provided solutions evaluator;*
        2. *Update the local bests for all the swarm;*
        3. *Pick the best performing particle and update the global best solution and respective fitness;*
        4. *Update each particle's velocity using:* $\vec{v}_i = \omega \vec{v}_i + c_{cognitive} \vec{r}_{cognitive} \circ (\vec{\hat{x}}_i - \vec{x}_i) + w_{social} \vec{r}_{social} \circ (\vec{\hat{g}} - x_i)$ *, where* $\vec{r}_{social}$ *and* $\vec{r}_{cognitive}$ *are random vectors created for each particle and used to apply the social and cognitive constants. The* $\circ$ *operator represents the Hadamard matrix multiplication operator;*
        5. *Update each particle's position, applying the velocity using:* $\vec{x}_i = \vec{x}_i + v_i$ *;*
    8. *Return* $\vec{\hat{g}}$ *.*
- Helper methods include `evaluate_swarm` and `get_best_particle`, which aid in manipulating the underlying data structures used to manage particles and particle fitnesses.

6.2.7 Framework Interfaces

Below are the framework's set of interfaces for the use-client objects to implement, as mentioned earlier:

- `ISolutionGenerator`: Implementors of this interface generate solutions inside the search space of the specific problem, holding all the required domain specific knowledge for navigation therein. Implementors must provide the `get_parameters`, `get_solution`, `get_neighborhood` and `get_closest_valid_solution` methods.

- `ISolutionEvaluator`: Implementors wrap the utility function from the problem domain and must implement the `evaluate` method, which receives the problem initial parameters and the current solution and evaluates it in context. The method must return a dictionary with a “score” key which resolves to the float value attributed to the evaluated solution. The resulting dictionary may also be used to transport additional meta-data about the solution evaluation, that can be used, for example, in the problem's solution visualizer.
- `ISolutionVisualizer`: Implementors provide mechanisms to display the specified solution, along with the problem's parameters. This entry-point for use-client code was included in the framework to allow real-time update of a potencial visualizer. The framework's optimizers are prepared to call upon the visualizer, to update its representation on every iteration if configured to do so. The relevant method is `display`, which expects the problem's parameters, the solution itself and the solution's utility for display purposes.

6.2.8 Helper functions

The `timed_optimizer_run` auxiliary function is part of the optimization framework's module and is to be used in conjunction with Python's decorator feature [Smi08], which allow for function or method transformation. In this particular usage it measures the time expended by a method call to an Optimizer subclass object, relying on a `_set_last_run_duration` method to store the value in the instance. It is used to decorate the `optimize` method.

6.3 Matcher optimization use-client

This component provides the adapters required to use the optimization framework in the context of loan request and investment offer combination. It leverages the constraint library extensions to build use-client code that can fit in the framework's entry points. This adapter also serves as an example implementation of a component for the optimization framework, as new adapters for different domain problems, would have some structural likeness due to the imposed API contract by the framework. Nonetheless, the internal details of the adapter classes presented here are specific for the solution manipulation mechanism used by this component and for this problem domain itself.

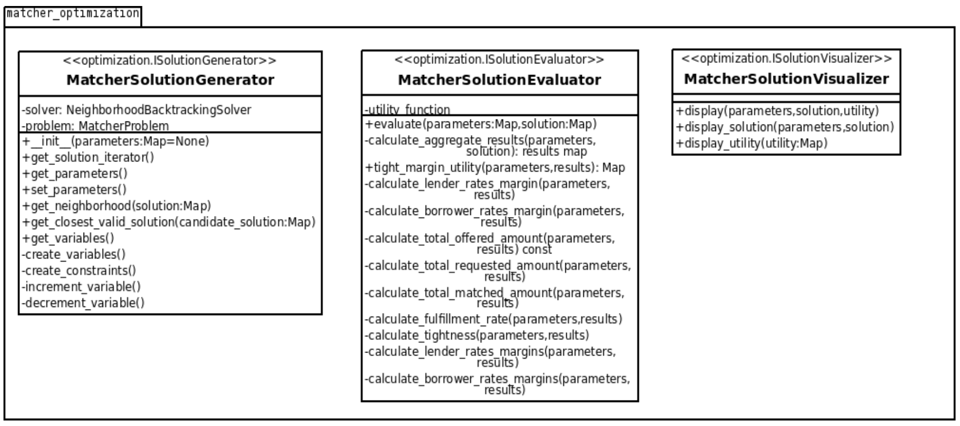
matcher_optimization
<<optimization.ISolutionGenerator>>
MatcherSolutionGenerator
-solver: NeighborhoodBacktrackingSolver
-problem: MatcherProblem
+__init__(parameters:Map=None)
+get_solution_iterator()
+get_parameters()
+set_parameters()
+get_neighborhood(solution:Map)
+get_closest_valid_solution(candidate_solution:Map)
+get_variables()
-create_variables()
-create_constraints()
-increment_variable()
-decrement_variable()
<<optimization.ISolutionEvaluator>>
MatcherSolutionEvaluator
-utility_function
+evaluate(parameters:Map,solution:Map)
-calculate_aggregate_results(parameters, solution): results map
+tight_margin_utility(parameters,results): Map
-calculate_lender_rates_margin(parameters, results)
-calculate_borrower_rates_margin(parameters, results)
-calculate_total_offered_amount(parameters, results) const
-calculate_total_requested_amount(parameters, results)
-calculate_total_matched_amount(parameters, results)
-calculate_fulfillment_rate(parameters,results)
-calculate_tightness(parameters,results)
-calculate_lender_rates_margins(parameters, results)
-calculate_borrower_rates_margins(parameters, results)
<<optimization.ISolutionVisualizer>>
MatcherSolutionVisualizer
+display(parameters,solution,utility)
+display_solution(parameters,solution)
+display_utility(utility:Map)

Figure 6.4: Matcher optimization use-client architecture.

6.3.1 Solution Generator

The `MatcherSolutionGenerator` class generates matcher solutions for the configured problem, according to a defined parameter specification for credit matching problems. It uses the matcher constraint library as the solution generation mechanism.

- The `set_parameters` method receives the map with the problem parameters, as presented in section 5.1, stores them in an instance variable and starts the generator initialization process (`initialize_generator`).
- The `initialize_generator` method creates all the objects required for solution generation, initializes them according to available parameters and connects them together to prepare for solution generation. The initialization steps are as follows:
  1. *Create the solver, instantiating* `NeighborhoodBacktrackingProblem`*;*
  2. *Create the problem, instantiating* `MatcherProblem` *with the solver and the operators (increment and decrement variable);*
  3. *Create the decision variables using* `create_variables`*;*
  4. *Constrain the variables according to the lenders' and borrowers' terms, using* `create_constraints`*;*
  5. *Create a new generator object using the problem object's* `getSolutionIter`*.*
- The `create_variables` adds the required decision variables to the problem. It adds a rate variable and an amount variable for each lender borrower combination in the format *rate_i_j* and *amount_i_j*, respectively. These string identifiers for the constraint library correspond to r_{ij} and a_{ij} from the earlier formal definition found in section 5.1. Due to the finite-domain nature of the constraint library, the range for the variables is necessarily integer. Due to the need to represent rates with fixed point precision of 2 decimal places, the strategy adopted consisted of ranging the rate variables from 0 to 10000, where 10000 corresponds to 100,00% times 10000. The original parameter values are converted into this range and back when the solution is presented. The amount variable is ranged from 0 (which means no deal) to the maximum amount offered of the lender (which would implied the lender gives all its money to the same borrower). The precision for amount computation is integer, with no decimal places.
- The `create_constraints` method applies the members' terms, as defined in section 5.1, to the decision variables in the form of constraints from the constraint library extensions built for the matcher. The step followed are:
  1. *Add lender constraints; for each lender:*
     1. *Build a list with all the amount variables, involving the current lender;*
     2. *Add a maximum sum constraint to the problem, based on the lender's maximum offered amount parameter, and apply it to the list of lender amount variables (C3);*
     3. *Add a minimum sum constraint to the problem, based on the lender's minimum offered amount parameter, and apply it to the list of lender amount variables (C3);*
     4. *Build an auxiliary list with all the rate variables, involving the current lender.*

        5. *Build a list of pairs, combining the amount and rate variables to use as input to the weighted average constraint. Using this format will enable the constraint to consider the amount variable as the weight and the rate variable as the value.*
        6. *Add a minimum weighted average or default constraint, using the lender minimum rate and 0 as the default value, and apply it to the list of amount-rate variable pairs (C2).*
    2. *Add borrower constraints; for each borrower:*
        1. *Build a list with all the amount variables, involving the current borrower;*
        2. *Add a maximum sum constraint to the problem, based on the borrower's maximum requested amount parameter, and apply it to the list of borrower amount variables (C4);*
        3. *Add a minimum sum constraint to the problem, based on the borrower's minimum requested amount parameter, and apply it to the list of borrower amount variables (C4);*
        4. *Build an auxiliary list with all the rate variables, involving the current borrower.*
        5. *Build a list of pairs, combining the amount and rate variables to use as input to the weighted average constraint.*
        6. *Add a maximum weighted average, based on the borrower maximum rate, and apply it to the list of amount-rate variable pairs (C1).*

- The `increment_variable` and `decrement_variable` are the operators made available for traversing the solution space. These simple operators add or subtract one unit from the specified variable in the specified solution. The resulting solutions are not guaranteed to be valid, but can be generated in an efficient manner and allow traversing the complete solution space.
- The `get_solution` and the `get_solution_iterator` methods, required by `ISolutionGenerator`, are direct solution retrieval methods, rely on the internal solution iterator, which is a Python generator object, which can be recurrently call to yield further solutions.
- The `get_neighborhood` method, required by `ISolutionGenerator`, is used to explore the solution space by retrieving solutions adjacent to a specified solution.
- The `get_closest_valid_solution,` also required by `ISolutionGenerator`, is useful for converting solutions obtained by an outside mechanism into valid ones, for which all the constraint apply.
- The `get_parameters` and `get_variables` methods, are useful for retrieving metadata about the solution generator itself, which can then be provided to a visualizer or used for other purposes.

6.3.2 Solution Evaluator

The solution evaluator built for the matcher adapter is `MatcherSolutionEvaluator`. It evaluates solutions according to a selected utility function. Below is a description of the most relevant methods:

- `evaluate`, required by `ISolutionEvaluator`, aggregates the standard format solution into a final result map, similar to the problem parameters, using calculate_aggregate_results. It then evaluates the configured utility function, specifying the problem's parameters and the resulting matches for the specified solution;
- `calculate_aggregate_results` takes in the problem's parameters and the standard format solution and calculates the final aggregate rates and amounts for each member, according to the values of r_{ij} and a_{ij}, as described below:
  - For each lender i: $r_i = \frac{\sum_{j=0}^{M} a_{ij} r_{ij}}{\sum_{j=0}^{M} a_{ij}}$, $a_i = \sum_{j=0}^{M} a_{ij}$;
  - For each borrower j: $r_j = \frac{\sum_{i=0}^{N} a_{ij} r_{ij}}{\sum_{i=0}^{N} a_{ij}}$, $a_j = \sum_{i=0}^{M} a_{ij}$;
- The `tight_margin_utility` function is a straightforward implementation of the formula presented in 5.2.2, which makes uses of several smaller auxiliary functions included in the `MatcherSolutionEvaluator`. All of these methods were extracted from the main utility function to improve readability and to allow their reuse in additional utility functions[4].

6.3.3 Solution Visualizer

This component provides simple visualization services for displaying the solutions during development. It uses a character-based representation of the member rates and amounts in a tabular fashion. Its `display` method, as required by `ISolutionVisualizer`, wraps its two inner display methods: `display_solution` and `display_utility`.

6.4 Summary

This chapter covered the implementation details on the solution which was previously presented. The text includes a detailed description of the responsibilities of each class involved, together with its methods, providing pseudo-code whenever relevant and trying to explain all the interactions that take place between all the system components.

4 This is the case with: `calculate_lender_rates_margin, calculate_borrower_rates_margin, calculate_total_offered_amount, calculate_total_requested_amount, calculate_total_matched_amount, calculate_fulfillment_rate, calculate_lender_rates_margins and calculate_borrower_rates_margins methods.`

The next chapter will demonstrate how this solution behaved in different scenarios which intended to simulate real operational settings.

7 Results Interpretation

A series of experiments where designed to test and validate the solution described in the previous chapters. This chapter describes how the experiments where assembled, and how the solution behaved in different scenarios and using different internal strategies.

7.1 Experimental Scenarios

A key concern while testing the solution was to have a sufficient amount of quality data that could be used to exercise the solution under different conditions, but maintaining a set of constant parameters. The solution was to design a simple data generator which would receive a high-level specification of the test scenario, and would then, stochastically, create a complete dataset to use as input for the matcher system.

To cater to the different types of settings, the following parameters stood out as relevant:

- N and M, the number of lenders and borrowers, respectively;
- $\hat{r}_i$, the mean lender rate;
- σ_{ri}, the lender rate standard deviation;
- $\hat{a}_i$, the mean lender amount;
- σ_{ai}, the lender amount standard deviation;
- $\hat{r}_j$, the mean borrower rate;
- σ_{rj}, the mean borrower standard deviation;
- $\hat{a}_j$, the mean borrower amount;
- σ_{aj}, the borrower amount standard deviation.

This information comprehensively describes a scenario, i.e., a template containing a high-level definition of the environment parameters. Using this type of templates, with the help of a stochastic environment generator – which was developed on top of a statistics module for Python [AS07] – it is possible to generate different matching environments each time a run is made.

The two scenarios which define the input for the different experiments were as follows:

1. *Tight market*: lenders offer lower interest rates, with borrowers allowing higher rates when finding a loan. Small deviation for both lenders and borrowers, keeping the market homogeneous.
2. *Loose market*: offer lower interest rates, with borrowers allowing higher rates when finding a loan. Larger deviation in rates distribution for both lenders and borrowers, providing some diversification and heterogeneity.

Table 7.1 show the exact settings for each of the referenced scenarios.

Table 7.1: Detailed scenario settings

	Scenario parameters	
Parameter	*Tight market*	*Loose market*
N	5	5
M	5	5
$\hat{r}_i$	5,00%	5,00%
σ_{ri}	1,00%	5,00%
$\hat{a}_i$	1.000,00 €	1.000,00 €
σ_{ai}	100,00 €	100,00 €
$\hat{r}_j$	20,00%	20,00%
σ_{rj}	1,00%	5,00%
$\hat{a}_j$	1.000,00 €	1.000,00 €
σ_{aj}	100,00 €	100,00 €

The scenarios were used as background for the experiments conducted, and allowed to understand how the different strategies and settings behaved under different environment conditions.

7.2 Metaheuristic parameter optimization

A set of undocumented experiments were empirically performed to fine tune each of the metaheuristic for the main utility function. Each section briefly describes the parameters which are supported by each metaheuristic implementation, and provides some considerations about the tuning that was performed.

7.2.1 Random Search

The metaheuristic was used only as baseline, representing uninformed search. As such no parameters were available to control the search strategy and no tuning was performed. For results for this algorithm, and how it compared with others, please refer to section 7.3.

7.2.2 Hill Climbing

The provided implementation of the Hill Climbing metaheuristic did not offer tuning capabilities, which results from the simplicity of the algorithm.

7.2.3 Simulated Annealing

The standard specification of the SA algorithm includes two key parameters: the initial energy and the cooling alpha constant. Some experimentation explored these parameters, but also a new parameter proposed by the created implementation was used with significant results: the neighborhood sample. This parameter allows to pick not just a random neighbor, but to gather a sample of the neighborhood for a given solution, and to pick the best member. Best values values concentrated around a 5% sample of each neighborhood, with much better results than just picking a random neighbor.

7.2.4 Genetic Algorithms

Key parameters in GAs include the initial population size, the number of individuals picked for reproduction, the size of the population to keep throughout the algorithm runs and the number of replacements to make in each generation. Successful results tended to appear around a 10 element population, with 5 individuals used for reproduction and substitution in each generation. Also relevant to the GA performance was going from a one-point crossover to a two-point crossover approach, as described in [Wik09i] and performing the genetic coding on the concatenated genetic string, instead of separate strings for each trait/variable.

7.2.5 Particle Swarm Optimization

The PSO algorithm reference specification considers 4 core parameters: the number of particles in the swarm, the inertial constant, the cognitive weight and the social weight. Together with the manipulation of these values, and additional optimization took place by applying the particle velocity not to all the variables (which resulted in very low solution feasibility) but to just one random variable per iteration, per particle (which returned solutions which were either valid or could be converted to valid ones by neighborhood exploration).

7.3 Metaheuristic analysis

The performance of each metaheuristic on the utility function was measured by successive executions, under different environments as mentioned above. Using the prepared scenario templates, an environment was generated for each type. Under each scenario, the metaheuristics were executed with a fixed 1000 iteration budget. The result of the utility function in each run was then recorded, and the experiment repeated using the same settings to record the mean utility score.

In the following section, the results of the successive executions are presented and analyzed.

7.3.1 Scenario 1: *Tight market*

The graph in figure 7.1 presents the results for the utility function under optimization at each iteration during the algorithm runs. The utility values are plotted for each of the implemented metaheuristics:

- Pure Random Search (PRS);
- Hill-Climbing (HC);
- Simulated Annealing (SA);
- Genetic Algorithm (GA);
- Particle Swarm Optimization (PSO).

The results suggest that the most efficient method was GA, converging towards better results considerably faster than the other algorithms and finishing with a better overall result after the all the runs were completed. The evolution is nonetheless irregular, exhibiting relevant breakthroughs at certain iterations, suggesting the importance of certain exceptional mutations and crossovers, as opposed to a continuous evolutionary improvement. With a more regular progression is the PSO algorithm, with strong results appearing late (by iteration 400) but keeping a consistent progression until stagnation at a local optima. The HC algorithm displays an interesting, near linear, improvement of the utility score as iterations increase. The PRS approach had very bad results, with the solution space exploration mechanisms trapping the metaheuristic in a region with bad solutions. The PRS did not offer a relevant contribution to the specific scenario, as it didn't provide a reasonable baseline. The results indicate that the baseline could be considered to be the SA approach due to its below average performance in this environment.

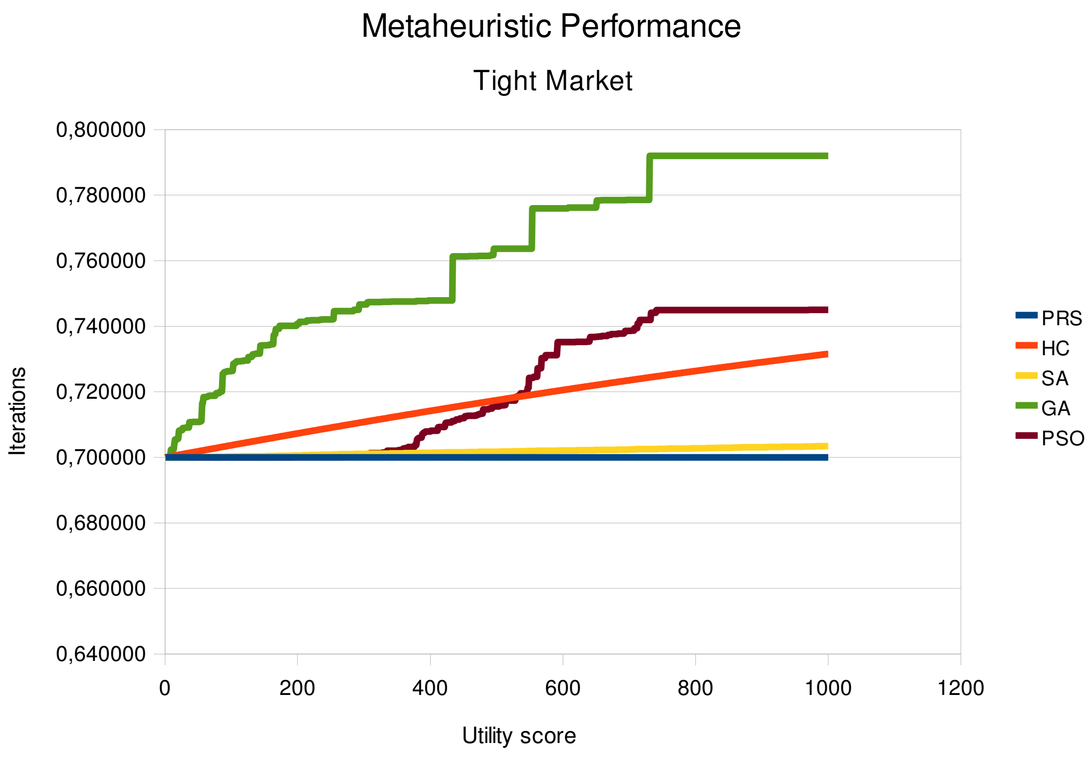

Figure 7.1: Scenario 1 – Plot of utility function value against iterations (by metaheuristic)

7.3.2 *Scenario 2: Loose market*

The graph in figure 7.2 presents the results for the utility function under optimization at each iteration during the algorithm runs. The utility values are plotted for each of the implemented metaheuristics. Notably the best performance also belongs to GA, although, in this setup, the HC and even the PRS have interesting performances. Near the end of the iteration budget, these two trajectory based algorithms surpass the results of the PSO implementation.

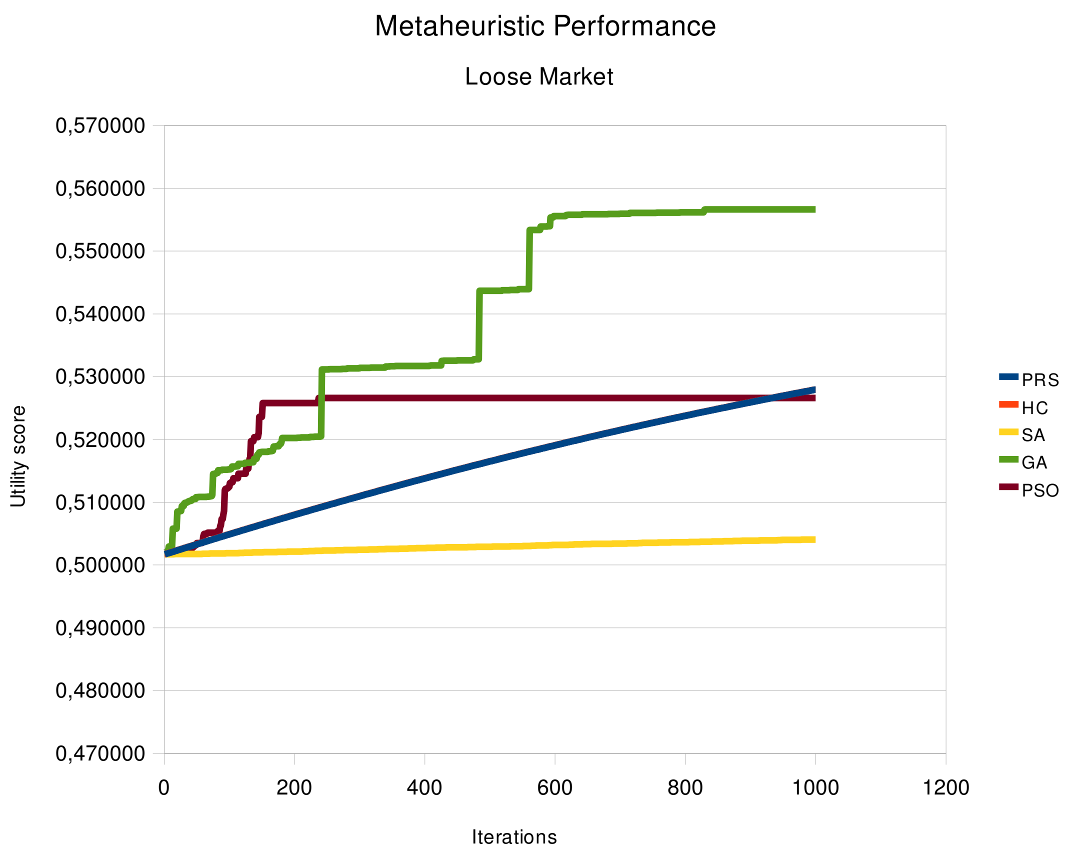

Figure 7.2: Scenario 2 – Plot of utility function value against iterations (by metaheuristic)

8 Summary Conclusions

This chapter synthesis the work described in this dissertation, evaluates if the initial goals where attained and proposes directions for further work.

8.1 Summary

Chapter 1 Introduces the concept of Peer-to-Peer lending, describes the context for this work and defines the goals to reach at the end of the project.

Chapter 2 Provides detailed context on the topic of P2P lending and describes the state-of-the-art in the field, reviewing practices and trends.

Chapter 3 Provides a description of existing techniques in the fields of Optimization and Constraint Programming that will be used throughout the project.

Chapter 4 Describes the specific problem at hand, in light of the previously presented concepts on P2P lending platforms.

Chapter 5 Formalizes the problem described in Chapter 4, and presents an architecture for the solution to implement to solve it.

Chapter 6 Presents the implementation details of the problem, providing the rationale behind key implementation decisions.

Chapter 7 Describes the experimental settings used to validate the system which was built, and provides a critical analysis of the experimental results.

8.2 Conclusions

From the experiments presented in Chapter 7 , it results that the system is effectively performing successful matches, and it is possible to pick the best performing metaheuristics in each scenario to use in a production environment. To that extent the goal of building a system capable of successfully finding optimal combinations of loans requests and investment offers, is believed to have been reached. The optimization approach worked and was successful in exploring the solution space considering the provided utility function. Further work could be done in experimenting with different utility functions, as is discussed in section 8.3.1.

Together with the successful results of the matching system, which was the main goal of this work, the optimization framework is seen as of value in itself. The framework was flexible enough to support multiple metaheuristics, gradually added as the work progressed, and was adequately decoupled from the problem to be used in solving other test problems. This level of modularity also allowed the problem specific code to be evolved by itself, while keeping the API contract, without breaking the optimizer as a whole.

8.3 Future Work

8.3.1 Utility Alternatives

Having seen the various optimization strategies applied to the utility function, it seems that the solution is effectively verified, i.e., solving the problem right. Nonetheless the question remains if the current solution solves the right business problem, i.e., a question of validation. This issue would naturally require a different approach, that analyzed the results from a business perspective, taking the members' and the platform's best interest into account. Relevant and important as it would be, this study would still be closer to the subject of Efficient Markets in the field of Economics, and was determined out of scope for this work.

8.3.2 Distributed Framework

Despite carefully designed and effectively reusable, the optimization framework left out of its scope a significant topic in optimization, which is distribution. Work on distributed optimization solutions is not new and distributed frameworks with similar goals to that produced in this work have been build in past [Res06]. Future extensions to this work, in what concerns the optimization architecture, should prepare the existing framework for parallelized work to allow distribution across multiple cores or even multiple nodes. The way the framework is designed should support evolving to a distributed architecture, but still there are communication and synchronization mechanisms that need to be put in place and may involve relevant work. An interesting direction for this work, could be leveraging existing distribution strategies such as MapReduce (see [DG04] for an introduction to MapReduce and [NMS07] a PSO implementation based on the MapReduce programming model). Complying with a popular programming model could significantly simplify the deployment to a cloud computing host, enabling access to virtually unlimited source of computational resources at a reasonable implementation and hardware cost.

8.3.3 Multi-Agent System Testbed and Matcher

Another approach which could bring significant insight into the problem, would be that of multi-agent systems. For one, building a multi-agent system that simulated the problem, with agents modeling lenders and borrowers with distinct profiles, would provide an interesting testbed for the existing matching system. Additionally, if negotiation mechanisms were to be added to the multi-agent platform, one could expect to achieve an alternative matching system with several advantages: not only would it solve the distribution issues with its naturally distributed nature, but also it would mimic more closely the underlying phenomena of Peer-to-Peer lending.

8.3.4 Social Rating

The third and final point to explore, would be that of trust management in the Peer-to-Peer lending community. As was reviewed during this work, existing P2P lending platforms rely on external trust anchors to determine the reliability of members. This is, by nature, a centralized approach which does not honor the social emphasis of a P2P network, and implies additional costs for both the platform operators and the users.

The trust management angle to this solution would imply designing a social rating scheme, based on work done in the fields of collaborative filtering and trust management, which allowed users to get involved in evaluating other users they knew and trusted. The system should be self-regulating, equipped with an effective reward and punishment system which protected the member's and the platform's interests.

Future work in this particular area seems the most promising, would positively influence the success of the Person-to-Person project in course.

8.4 Final Remarks

The final solution that results from this work may not be of production quality, due to the considerations presented earlier, but still, significant headway has been made into creating the infrastructure for an innovative community driven project in Portugal. As the overall goals have been reached, despite its limited resources, one cannot help but think that this project was indeed successful, and that its usefulness will be proven in the near future.

References

[AS07] István Albert, Gary Strangman. python-statlib - Google Code, 2007. http://code.google.com/p/python-statlib/, retrieved June 09 retrieved in June 2009.

[Bar99] Roman Barták. Constraint Programming: In Pursuit of the Holy Grail. In Proceedings of WDS99 , pp. 10. MATFYZPRESS, 1999.

[BdP04] Banco de Portugal. Regime Geral das Instituições de Crédito e Sociedades Financeiras, 2004. www.bportugal.pt/publish/legisl/RGICSF2004_p.pdf retrieved in June 2009.

[Bog09] David Bogoslaw. Peer-to-Peer Lending: Problems and Promise – BusinessWeek, 2009. http://www.businessweek.com/investor/content/apr2009/pi2009043_811816.htm retrieved in June 2009.

[BR03] Christian Blum, Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. In , pp. . ACM Computing Surveys, Vol. 35, No. 3, September 2003, pp. 268–308., 2003.

[CCH99] Patrice Cálegary, Giovanni Coray, Alain Hertz, Daniel Kobler, and Pierre Kuonen. A taxonomy of evolutionary algorithms in combinatorial optimization. In , pp. 145–158. Journal of Heuristics, 1999.

[Cel07] Celent. Up Close and Personal with Online Lending. In Celent Report , pp. n/a. Celent LLC, 2007.

[Cha09] Scott Chacon. Git - Fast Version Control System, 2009. http://git-scm.com/ retrieved in June 2009.

[CP00] Eric Collins, Alan Price. Apparatus and Method for Facilitating Agreement Over a Network . In United States Patent Application 20020007362 , pp. n/a. United States Patent and Trademark Office, 2000.

[DG04] Jeffrey Dean, Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In Proceedings of the 6th Symposium on Operating Systems Design and Implementation , pp. 107-113. , 2004.

[ETH05] ETH Zürich. Population Based Methods, 2005. www.tik.ee.ethz.ch/tik/education/lectures/.../population_based.pdf .

[Fow04] Martin Fowler. Inversion of Control Containers and the Dependency Injection pattern, 2004. http://www.martinfowler.com/articles/injection.html retrieved in May 2009.

[Fra05] Raquel Campos Franco. Defining the Nonprofit Sector: Portugal. In Working Papers of the Johns Hopkins Comparative Nonprofit Sector Project, no. 43 , pp. n/a. The Johns Hopkins Center for Civil Society Studies, 2005.

[Git09a] GitHub. Secure source code hosting and collaborative development - GitHub, 2009. http://github.com/ retrieved in June 2009.

[Git09b] GitHub, Luís Martinho. lmartinho's collaborative_credit_matcher at master - GitHub, 2009. http://github.com/lmartinho/collaborative_credit_matcher/ retrieved in June 2009.

[GJ79] Michael R. Garey, David S. Johnson. *Computers and Intractability*. W. H. Freeman, , 1979.

[GL07] Fred Glover, Manuel Laguna. *Tabu Search*. Springer, First Edition, 1997.

[Glo86] Fred Glover. Future paths for integer programming and links to artificial intelligence. In Computers and Operations Research , pp. 533-549. , 1986.

[Gor98] Martin Gorsky. The Growth and Distribution of English Friendly Societies in the Early Nineteenth Century. In Economic History Review , pp. 489-511. Economic History Society, 1998.

[Hen09] Colin Henderson. The right idea for the times | UncrunchAmerica | Fast Company. In , pp. . , 2009.

[Ho88] Mahabub Hossain. *Credit for Alleviation of Rural Poverty: The Grameen Bank in Bangladesh*. Bangladesh Institute of Development Studies, First Edition, 1988.

[Hop01] Adrian A. Hopgood. *Intelligent Systems for Engineers and Scientists*. CRC Press, Second Edition, 2001.

[JF88] Ralph E. Johnson and Brian Foote. Designing Reusable Classes. In *Designing Reusable Classes*, pp. 22-35. , June/July 1988.

[KE95] James Kennedy, Russel Eberhart. Particle Swarm Optimization. In Proceedings of IEEE International Conference on Neural Networks, Piscataway, NJ , pp. 1942-1948. IEEE, 1995.

[KGV83] S. Kirkpatrick , C. D. Gelatt , Jr. , M. P. Vecchi. Optimization by Simulated Annealing. In Science , pp. 671-680. New Series, Vol. 220, No. 4598, 1983.

[Kiv09b] Kiva. Kiva - What Is Kiva?, 2009. http://www.kiva.org/about/what/ retrieved in June 2009.

[Kla08] Michael Klafft. Peer to Peer Lending: Auctioning Microcredits over the Internet. In Proceedings of the International Conference on Information Systems, Technology and Management , pp. n/a. A. Agarwal, R. Khurana eds., IMT, Dubai, 2008.

[Nie05] Gustavo Niemeyer. API Documentation (python-constraint), 2005. http://labix.org/doc/constraint/ retrieved in May 2009.

[NMS07] Andrew W. McNabb, Christopher K. Monson, Kevin D. Seppi. MRPSO: MapReduce particle swarm optimization. In Proceedings of the 9th annual conference on Genetic and evolutionary computation , pp. 177-177. ACM New York, NY, USA, 2007.

[Par99] Partnoy, Frank. The Siskel and Ebert of Financial Markets: Two Thumbs Down for the Credit Rating Agencies. In Washington University Law Quarterly, Vol 77 , pp. 619-712. Washington University, 1999.

[Pel09] Michel Pelletier. PEP 245 -- Python Interface Syntax, 2009. http://www.python.org/dev/peps/pep-0245/ retrieved in June 2009.

[Pros09] Prosper Marketplace CA, Inc.. Terms Of Use - Prosper Loans Marketplace, 2009. http://www.prosper.com/legal/terms_of_use.aspx retrieved in June 2009.

[PS82] Christos H. Papadimitriou, Kenneth Steiglitz. *Combinatorial optimization: Algorithms and Complexity*. Dover Publications, Inc., New York, , 1982.

[PSF09a] Python Software Foundation. The Python Language Reference, 2009. http://docs.python.org/reference/ retrieved in June 2009.

[PSF09b] Python Software Foundation. The Python Standard Library — Python v2.6.2 documentation, 2009. http://docs.python.org/library/ retrieved in June 2009.

[RBW06] Francesca Rossi, Peter van Beek, Toby Walsh. *Handbook of Constraint Programming*. Elsevier, , 2006.

[Reg05] The Register. Zopa – the bank that likes to say 'Hello There!' - The Register, 2005. http://www.theregister.co.uk/2005/07/29/zopa_p2p_banking/ retrieved in June 2009.

[Res06] André Restivo. *Dynamic Scenario Simulation Optimization*. Master Thesis, Universidade do Porto, Faculdade de Engenharia, 2006.

[Rie00] Dirk Riehle. *Framework Design: A Role Modeling Approach*. Ph.D. Thesis, No. 13509, Switzerland, ETH Zürich, 2000.

[Rod09] Roda Crédito Colaborativo. roda.cc - Roda Crédito Colaborativo, 2009. http://www.roda.cc/ retrieved in June 2009.

[RW09] Guido van Rossum, Barry Warsaw. PEP8 - Style Guide for Python Code Style Guide for Python Code, 2009. http://www.python.org/dev/peps/pep-0008/ retrieved in May 2009.

[Sche01] Neil Schemenauer, Tim Peters, Magnus Lie Hetland. PEP 255 -- Simple Generators, 2001. http://www.python.org/dev/peps/pep-0255/ retrieved in June 2009.

[SHJ00] J. Swisher, P. Hyden, S. Jacobson and L. Scruben. A survey of simulation optimization techniques and procedures. In *Proceedings of the 2000 Winter Simulation Conference*, pp. . K. Kang J.A. Joines R.R. Barton and P.A. Fishwick, editeurs, 2000.

[Smi08] Kevin D. Smith, Jim J. Jewett, Skip Montanaro, Anthony Baxter. PEP 318 -- Decorators for Functions and Methods, 2008. http://www.python.org/dev/peps/pep-0318/ retrieved in June 2009.

[TH09] Dave Thomas, David Heinemeier Hansson. *Agile Web Development with Rails*. The Pragmatic Programmers, Third Edition, 2009.

[Tsa93] Edward Tsang. *Foundation of Constraint Satisfaction*. Academic Press, Department of Computer Science, University of Essex, UK, , 1993.

[Unc09] Uncrunch America. About UNCRUNCH AMERICA™ - Uncrunch America, . http://www.uncrunch.org/uncrunchAmerica/about.action .

[Wik09a] Wikipedia. Person-to-person lending, 2009. http://en.wikipedia.org/wiki/Peer-to-peer_lending retrieved in June 2009.

[Wik09b] Wikipedia. Financial crisis of 2007–2009 - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Financial_crisis_of_2007-2009 retrieved in June 2009.

[Wik09c] Wikipedia. Combinatorial optimization - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Combinatorial_optimization retrieved in June 2009.

[Wik09d] Wikipedia. Particle swarm optimization - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Particle_Swarm_Optimization, retrieved in June 2009 retrieved in June 2009.

[Wik09e] Wikipedia. Constraint satisfaction - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Constraint_satisfaction, retrieved June 2009 retrieved in June 2009.

[Wik09f] Wikipedia. Software framework - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Software_framework retrieve in June 2009.

[Wik09g] Wikipedia. Interface (computer science) - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Interface_(computer_science) retrieved in June 2009.

[Wik09h] Wikipedia. Genetic algorithm - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Genetic_algorithm retrieved in June 2009.

[Wik09i] Wikipedia. Crossover (genetic algorithm) - Wikipedia, the free encyclopedia, 2009. http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm) retrieved in June 2009.

[Wyd99] Bruce Wydick. Can Social Cohesion be Harnessed to Repair Market Failures? Evidence from Group Lending in Guatemala. In The Economic Journal, Vol. 109, No. 457 , pp. 463-475. Blackwell Publishers, 1999.

[Zhi07] Anatoly Zhigljavsky. *Stochastic Global Optimization*. Springer, , 2007.

[Zopa09a] Zopa Ltd.. How it works, 2009. http://uk.zopa.com/ZopaWeb/public/about-zopa/how-it-works.html retrieved in May 2009.

[Zopa09b] Zopa Ltd.. FAQs - What are the Zopa Principles?, 2009. http://uk.zopa.com/zopaweb/public/help/help-faqs-interested.html#principles retrieved in June 2009.

[Zope09] Zope 3 project team. Zope 3 API Documentation, 2009. http://apidoc.zope.org/++apidoc++/ retrieved in June 2009.