University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

Student-Project Allocation in the Matching Algorithm Toolkit

Frederik Glitzner

March 24, 2023

Abstract

Many real-world allocation scenarios such as job recruitment, project assignment, or kidney exchanges can be modelled using classical matching problems. The Matching Algorithm Toolkit is a web application making many state-of-the-art matching under preference algorithms available for three different problem classes. However, there are three major shortcomings: there are bugs and minor features that would greatly increase the usability and functionality of the system, there are engineering challenges to be addressed, and the Student-Project Allocation (SPA) problem class is missing completely although it is of great theoretical and practical relevance.

This project addresses all three key issues identified in the existing system. First, we fix bugs and inconsistencies introduced by previous developers to ensure the integrity and reliability of the system. Second, we design and implement a new input method that enables users to input instances more efficiently. Third, we improve the appearance and usability of the random instance generator forms and user manual, making them more intuitive and user-friendly. In addition to these improvements, this project extends the Toolkit to include the SPA problem class. For SPA, we make available five algorithms that find solutions with respect to different optimality criteria such as stability and cost-optimality. Moreover, we implement other essential features for SPA in the Toolkit such as instance readers, random instance generators, and a user-friendly output. All changes are successfully deployed to the live server.

Functional testing and a user study confirm that the system is highly user-friendly and that the changes are well-integrated with the system. Furthermore, the empirical evaluation conducted on more than ten thousand random instances shows how the SPA algorithms vary with respect to matching sizes, costs, and computation times, as the instance size, student preference list lengths, and agent popularity parameters are varied.

Acknowledgements

I would like to thank Professor David Manlove for inspiring and motivating me to work on matching problems, his continuous guidance and support, and the many hours we spent discussing the Toolkit, SPA, and algorithmics.

I would also like to thank Dr William Pettersson for his ad hoc support with the Toolkit and deployment which saved many hours that could be spend working on other parts of the project.

I would also like to thank all participants of my user study for their time, feedback, and suggestions for improvement.

Finally, I am very thankful for the support and motivation of my family and friends throughout this exciting but at times very stressful time.

Education Use Consent

I hereby grant my permission for this project to be stored, distributed and shown to other University of Glasgow students and staff for educational purposes. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Signature: Frederik Glitzner Date: 24 March 2023

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Problem Statement 1

1.3 Outline 1

2 Background **2**

2.1 Matching Problems 2

2.2 Student-Project Allocation 4

2.3 Problem Solvers 6

3 Analysis and Requirements **8**

3.1 Analysis of the Toolkit 8

3.1.1 Back-End 8

3.1.2 Front-End 10

3.1.3 Hosting 10

3.1.4 Different Versions 10

3.1.5 Gaps in the current System 11

3.2 Requirements 11

3.2.1 User Stories 11

3.2.2 Functional Requirements 11

3.2.3 Non-Functional Requirements 12

4 Design **13**

4.1 Student-Project Allocation Problem Class 13

4.1.1 User Interface 13

4.1.2 Abstract Models 14

4.1.3 Instance Generator 15

4.1.4 One-Sided Solvers 16

4.1.5 Two-Sided Solvers 16

4.1.6 Readers and Writers 17

4.2 User Input and General Usability 17

5 Implementation **19**

5.1 Student-Project Allocation Problem Class 19

5.1.1 User Interface 19

5.1.2 Models and Flows 20

5.1.3 Random Instance Generator 22

5.1.4 Solvers and Algorithms 22

5.1.5 Readers and Writers 24
5.1.6 Stability Checker 25
5.2 General Improvements 25
5.2.1 Maximum Popular Matching in Stable Marriage 25
5.2.2 User Input and General Usability 26
5.2.3 Continuous Integration and Development 27
5.2.4 Bugs in the Previous System 27

6 Evaluation **28**
6.1 Functional Testing 28
6.2 Empirical Evaluation 28
6.3 User Study 32

7 Conclusion **34**
7.1 Achievements 34
7.2 Reflection 34
7.3 Future work 35

Appendices **36**

A Appendices **36**
A.1 Application Screenshots 36
A.2 Setup and Quickstart Instructions 39
A.2.1 Folder Structure 39
A.2.2 Requirements and Installation 39
A.2.3 Setup 39
A.2.4 Usage 39
A.2.5 Features and Manual 39
A.2.6 Testing 39
A.3 Adding New Algorithms 40
A.4 General Contribution Guidance 41
A.5 Test Cases 42
A.6 Code Contributions 43
A.6.1 Core Front-End Changes 43
A.6.2 Core Back-End Changes 43
A.7 SPA Stable Algorithm Pseudo-Code 46
A.8 User Study Information Sheet 47
A.9 User Study Task Sheet and Questionnaire 49
A.10 User Study Ethics Checklist 55
A.11 User Study Responses 57

Bibliography **62**

1 Introduction

1.1 Motivation

Matching algorithms are used to find pairwise assignments. In the context of agents, the goal is to organise the agents so that some criteria are satisfied, such as maximum allocation size, capacity restrictions, or optimal preference satisfaction. For matching under preferences, there are many well-studied problem classes that are of complexity-theoretic and real-life relevance. Student-Project Allocation (SPA), for example, is a problem faced by schools and universities all over the world, with hundreds of participants in the Honours project allocation at the University of Glasgow alone. The model is also applicable to other contexts, for example in wireless network engineering. Therefore, it is important to have fair, stable, and efficient algorithms.

For research and demonstration purposes, it is important to provide tools to the community that make these algorithms accessible. The Matching Algorithm Toolkit (Toolkit) is a general-purpose matching web application currently available at `https://matwa.optimalmatching.com/` which has been developed by many different contributors within the University of Glasgow's School of Computing Science. It provides features such as solving the House Allocation, Hospital Resident, and Stable Roommate problems using a range of state-of-the-art algorithms, uploading or randomly generating problem instances, and showing or downloading the results.

1.2 Problem Statement

With almost 500 Java classes in the back-end service alone, the Toolkit has grown to a substantial software project accumulating many matching algorithms not easily available or usable elsewhere. Over time, many inconsistencies and bugs have been introduced, and the general usability and feature set of the application could be improved. Also, recent research related to the SPA problem class, a generalisation of the Hospital Resident model, has yielded many algorithms and complexity results important both in theory and practice. However, the problem class is yet to be implemented and integrated with the Toolkit. Therefore, this project aims to improve and extend the Toolkit, addressing usability and functionality issues and implementing the SPA problem class in front- and back-end with all relevant features that the system should provide.

1.3 Outline

- **Chapter 2:** will provide a basic overview over matchings and popular matching problems, introduce the Student-Project Allocation problem, its variants, and relevant results, and conclude by mentioning different software to solve matching problems under preference.
- **Chapter 3:** will introduce and analyse the Matching Toolkit system before the changes and outline the project requirements.
- **Chapter 4:** will present the design considerations and choices for the changes.
- **Chapter 5:** will show how the changes were implemented in the system.
- **Chapter 6:** will present the set up and results of multiple types of evaluation and testing.
- **Chapter 7:** will conclude the dissertation and outline directions for future work.

2 Background

Lots of research has been conducted in the field of matching problems and algorithms for problems involving preferences. First, an introduction to matching will provided, then a focussed overview of the Student-Project Allocation problem and relevant results presented, and finally an overview over existing related software shown.

2.1 Matching Problems

A graph G is a set of vertices V combined with a set of edges E. G is called bipartite if V can be partitioned into two disjoint and independent sets U, U' so that every edge in E connects a vertex in U with a vertex in U'. A matching in G is a subset $M \subseteq E$ of the edges so that no two edges in M have a vertex in common. A maximum cardinality matching in G is a matching containing the largest possible number of edges. For bipartite graphs, such a matching as in Figure 2.1 can be found in $O(\sqrt{n(n+m)})$ time, where $n = |V|$, $m = |E|$, using the Hopcroft-Karp algorithm (Hopcroft and Karp 1973). For non-bipartite general graphs, such a matching as in Figure 2.2 can be found in the same time complexity using Edmonds' algorithm (Micali and Vazirani 1980).

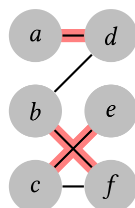


Figure 2.1: *Maximum cardinality matching in a bipartite graph*

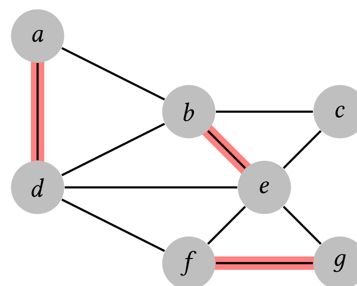


Figure 2.2: *Maximum cardinality matching in a non-bipartite graph*

When dealing with weighted graphs, that is a graph G as above together with an assignment $w : E \to \mathbb{R}$ so that $w(e) \geq 0\ \forall e \in E$, the cost of a matching M is given by $c(M) = \sum_{e \in M} w(e)$. Naturally, the question of a minimum cost matching arises. Using the Hungarian algorithm, a minimum cost maximum cardinality matching can be found in $O(n^3)$ time in a bipartite graph (Kuhn 1955). This problem can also be modelled using s-t flows by constructing an appropriate network from G, for bipartite G. Then, flow algorithms such as the Ford-Fulkerson algorithm or even faster algorithms such as by Orlin (2013) can be applied to find minimum cost maximum flows. These can then be converted into a matching, leading to a time complexity of $O(|E|f)$ time, where f is the flow value of the maximum flow.

Now, if vertices are considered agents, then a matching is an assignment of pairs of agents. In the real world, agents commonly have preferences over their assigned partners. In the case of bipartite sets of agents, preferences can be one-sided, that is members of one set of agents has preferences over members of the other set of agents, or two-sided, meaning that each member has

preferences over members of the other set of agents. Note that these preferences can be complete, meaning that every agent has preferences over every other relevant agent, or incomplete, and that preferences can have ties, that is some agents can have indifferences over some of their rated agents. Complete lists and no ties will be assumed unless specified otherwise. In a matching, these ordinal preferences can be measured by a profile. A profile P is an r-dimensional vector, where r is the maximum length of a preference list, and the first position accounts for the number of first choices achieved in the assignment, the second position accounts for the number of second choices achieved, and so on. The cost of a matching M, in this case, can then be defined by $c(M) = \sum_{i=1}^{r} r * P_r$, the sum of linearly scaled rank positions.

The following are some widely studied matching under preferences problem classes, with information taken from Manlove (2013):

- **House Allocation (HA):** a bipartite matching problem with one-sided preferences. Applicants have preferences over houses, each applicant can be assigned to at most one house, and each house can be assigned to at most one applicant (in the case of CHA, houses also admit capacities larger than one). A real-world motivation: university accommodation where students can rank their potential homes in order of personal preference.
- **Stable Marriage (SM):** a bipartite matching problem with two-sided preferences. Each group of agents has preferences over members of the other group, one group proposes to members of the other group, and each agent can be paired with at most one other agent. A real-world motivation: the dating market where people have preferences over their potential partners but can marry at most one.
- **Hospital Resident (HR):** a bipartite matching problem with two-sided preferences. A generalisation of SM where one set of agents admits capacities larger than one. A real-world motivation: new doctors (residents) apply for graduate jobs at hospitals and have preferences over their applications. Similarly, the hospitals have an upper limit on the number of new doctors they can hire and have preferences over the applicants, for example, based on test results or other qualifications.
- **Student-Project Allocation (SPA):** a bipartite matching problem with one- or two-sided preferences. A generalisation of HR with three sets of agents: students, projects, and lecturers. Lecturers propose any number of projects but have capacities on the number of students they can supervise. Similarly, projects have capacities on the number of students that can work on them. Students have preferences over the projects, and lecturers either have no preferences, preferences over the students, or preferences over the projects they offer. A real-world motivation: Honours project allocation at the University of Glasgow's School of Computing Science.
- **Stable Roommates (SR):** a non-bipartite matching problem with preferences. The agents, all part of a single set, have preferences over each other. A real-world motivation: pairing up roommates in University accommodation.

While cost-optimality seems to be the most intuitive, it might actually be more important in practice to, for example, limit the number of agents that receive their worst choice, or to apply a stability condition. Simplified, a matching is said to be stable if there are no unmatched agents that could be accommodated in the matching and no matched agents that prefer each other over their currently assigned partners (Manlove 2013). Other examples of optimality criteria are rank-maximal matchings in which, simply speaking, the number of happy agents is maximised, or generous maximum matchings in which the number of unhappy agents is minimised (Kwanashie et al. 2015).

Stability applies only to matching problems with two-sided preferences and is important in a game-theoretic sense. Suppose a nontrivial SM matching is not stable but there are no unassigned agents. Then there exists a pair of agents that are not assigned to each other but prefer each other over their currently assigned partners. They, therefore, have an incentive to not abide by the rules of the matching, therefore making their current partners unassigned.

Exactly this logic can be exploited in an algorithmic sense. The original Gale-Shapley algorithm (Gale and Shapley 1962) for SM with complete preference lists operates by starting out with an empty matching and maintaining a valid matching throughout. Members of one set successively propose an assignment to agents of the other set that they have not yet proposed to, and if that agent is unassigned or prefers the proposing one over their current assignment, they get assigned to each other, otherwise that agent remains with their current partner. The algorithm is therefore referred to as a deferred acceptance algorithm and can be implemented to yield a stable matching in time complexity linear to the input size.

2.2 Student-Project Allocation

The SPA problem has been studied in various settings and under numerous optimality criteria. Manlove (2013) describes the problem setup formally as involving three sets of distinct agents: students $S = \{s_1, ..., s_i\}$, projects $P = \{p_1, ..., p_j\}$, and lecturers $L = \{l_1, ..., l_k\}$. Each project $p \in P$ is offered by a single lecturer $l \in L$ and the goal is to find an assignment of students to projects as a subset of $S \times P$ so that every student is assigned to at most one project. A graph example with three students, four projects, and two lecturers is shown in Figure 2.3. Furthermore, projects and lecturers have capacities indicating the maximum number of students that can be assigned to it/them. A project or lecturer is called under-subscribed or full if their capacity is not yet reached or reached, respectively. Finally, students can provide a ranking list of projects, with the first entry being their most favourite project and the last project being their least favourite project. They can choose to rank as many projects as they would like to, with all non-rated projects considered unacceptable. To find such a matching, a range of algorithmic techniques have been investigated, such as integer programming (IP), flow approaches, and deferred acceptance algorithms.

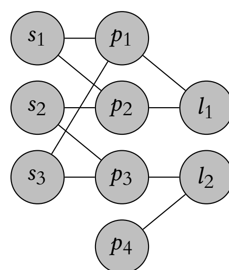


Figure 2.3: *A SPA Graph Example*

Anwar and Bahaj (2003) solve basic one-sided models using IP techniques in which every student $s \in S$ is assigned exactly one project $p \in P$. The objectives considered are to minimise the number of projects supervised by any one staff member and the assignment cost.

Another approach is taken by Lavery (2003). They use a minimum-cost maximum flow algorithm to find a cost-optimal matching after converting the SPA instance to an s-t flow. The work is extended by Kwanashie et al. (2015) who use network flows to also find rank-maximal, greedy, and generous maximum matchings in addition to cost-optimal matchings. A rank-maximal matching is a matching that has lexicographically maximum profile (Irving et al. 2006). Here, the maximum number of students are assigned to their first choice, then the maximum number of students are assigned to their second choice, and continuing. Sizes of rank maximal matchings may vary, however (Manlove 2013). Therefore, they also looked into finding greedy maximum matchings, which are maximum matchings with lexicographically maximum profiles, by optimising with respect to matching size and student preferences. Alternatively, in generous maximum matchings, the minimum number of students are matched to their Rth-choice project (where R is the maximum length of any students' preference list) and then the minimum number of students are

matched to their $(R-1)$th-choice project, and so on. In their general minimum cost maximum flow-based algorithm using principles from Orlin et al. (1993) which can be used to find minimum cost, greedy, and generous maximum matchings using appropriate network constructions, they achieve a worst-case time complexity of $O(m^2\log(n) + mn\log^2(n))$ where $n = |S|$ and m is the sum of all the students' preference list lengths. In their faster algorithms for greedy and generous maximum matchings, they are able to achieve a worst-case time complexity of $O(n^2Rm)$.

Student Preferences	**Lecturer Preferences**	**Project Capacities**
$s_1 : p_1\ p_2$	$l_1 : s_1\ s_2\ s_3$	$p_1 : 1\ p_2 : 2$
$s_2 : p_2\ p_3$	$l_2 : s_2\ s_1\ s_3$	$p_3 : 2\ p_4 : 1$
$s_3 : p_1\ p_3$	**Lecturer Capacities**	$l_1 : 2\ l_2 : 1$
Stable Matching	$\{(s_1, p_1), (s_2, p_2), (s_3, p_3)\}$	

Figure 2.4: *SPA-S Instance and Stable Matching*

The one-sided preference model can be extended to two-sided preferences when lecturers also express preferences over students (SPA-S), the projects they offer (SPA-P), or specific student-project pairs (SPA-(S,P)). In the problem variant SPA-S, every student $s \in S$ provides a strict ranking of a subset of P, each project of which is said to be found acceptable by s. Furthermore, every lecturer $l \in L$ provides a strict ranking of all students that find at least one of l's projects acceptable. A matching M is an assignment such that every student finds their assigned project acceptable and project and lecturer capacities are respected. This is an example of two-sided preferences and here a matching is said to be stable if it does not admit a blocking pair. Manlove (2013) defines a blocking pair as in Definition 1.

Definition 1 (SPA-S Blocking Pair) *A student-project pair $(s_i, p_j) \in (S \times P) \setminus M$ is a blocking pair of a matching M, if:*

1. *s_i finds p_j acceptable;*
2. *either s_i is unassigned in M, or s_i prefers p_j to their assigned project;*
3. *either*
   (a) *p_j is undersubscribed and l_k is undersubscribed, or*
   (b) *p_j is undersubscribed, l_k is full, and either s_i is assigned to a project from l_k or l_k prefers s_i to their worst student assigned in M, or*
   (c) *p_j is full and l_k prefers s_i to the worst student assigned to project p_j,*

where l_k is the lecturer offering p_k.

Figure 2.4 shows such a SPA-S instance with three students, two lecturers, and four projects, as well as a stable matching as a set of student-project pairs that satisfy the capacity constraints. Note that there may exist multiple stable matchings for a given instance. To find such stable matchings, Abraham et al. (2007) present two combinatorial algorithms, a student-oriented (SPA-student) and a lecturer-oriented (SPA-lecturer) one, both running in linear time with respect to the total length of the preference lists.

SPA-student returns a stable matching that is the best possible for the students. Precisely, they arrive at the following theorem: "For a given instance of SPA-S, any execution of Algorithm SPA-student constructs the stable matching in which each assigned student is assigned to the best project that they could obtain in any stable matching, whilst each unassigned student is unassigned in any stable matching" (Abraham et al. 2007). It is an extension of the original Gale-Shapley algorithm for SM and also works on the basis of deferred acceptance.

SPA-lecturer returns a best-possible stable matching M for the lecturers. Specifically, they prove the following: "Each lecturer prefers M to any stable matching in which they have a different set of assigned students" and "Each student is unassigned or is assigned to the worst project he/she has in any stable matching" (Abraham et al. 2007). It is therefore a trade-off to optimise the matching

with respect to the students' or lecturers' preferences. However, SPA-S stable matchings are related, as the Unpopular Projects Theorem (Theorem 1) by Abraham et al. (2007) shows.

Theorem 1 (Unpopular Projects Theorem) *For a given SPA instance, the following holds.*

1. *Each lecturer has the same number of students in all stable matchings.*
2. *Exactly the same students are unassigned in all stable matchings.*
3. *A project offered by an under-subscribed lecturer has the same number of students in all stable matchings.*

Complete and strict preference lists in SPA-S are not always suitable in practice though. SPA-ST is the extension allowing ties in the preference lists. Here, a matching can be weakly stable, strongly stable, super stable, or none as adapted from other stable matching problems by Abraham et al. (2007). Although a weakly stable matching always exists, Cooper and Manlove (2018) show that finding a maximum size weakly stable matching is NP-hard and provide an approximation algorithm achieving a size of at least $\frac{2}{3}$ of the optimal based on approximation techniques from Király (2013). For strong and super stability, Olaosebikan and Manlove (2019) and Olaosebikan and Manlove (2018) provide the first polynomial time algorithms to find such matchings optimally or report that none exist. Olaosebikan (2020) also contributes IP formulations for these problems for experimental purposes and evaluates the existence and sizes of stable matchings empirically on random and real-world data. Furthermore, they extend Theorem 1 to SPA-ST.

A natural extension is to introduce lower quotas on the projects, lecturers, or both. Cooper (2020) investigates a similar setting extending SPA-ST to lecturer targets indicating a preferred number of allocations and calls the problem SPA-STL. The author provides polynomial time algorithms when optimizing for various target objectives without stability, proves that combined with stability the problem becomes intractable, and presents IP formulations for the problems. Similarly, SPA with lower quotas and project closure was studied by Monte and Tumennasan (2013) and extended by Kamiyama (2013) to the case where student preference lists may be incomplete, but both only consider one-sided preferences of students over projects. Biró et al. (2010) studies the college admissions with lower and common quotas problem (CA-LQ) which, structurally, is closely related to SPA-S with lower quotas. They find that the problem is NP-hard in general, but tractable under tight conditions, and that a stable matching may not exist at all. Other papers such as by Arulselvan et al. (2018) also investigate related structural and algorithmic properties of matching with lower quotas, but without considering stability and focussing on cardinal utilities rather than ordinal preferences.

Other applications of the SPA model outside of student allocation can be found, for example, in user and channel assignments in multi-cell networks such as presented by Baidas et al. (2019). Another special case of the SPA model is discussed by Elviwani et al. (2018), who assign workers to posts in institutions based on performances and preferences, which could be extended to department segmentation for large institutions when using the full SPA model.

2.3 Problem Solvers

Various software tools such as in Table 2.1 have been developed to solve matching problems.

The online tool by Oozeer (2019) (henceforth referred to as A) visualises matching algorithms, but does not take preferences into account. An online tool by Technical University of Munich (2016) (B) lets the user apply general graph matching algorithms such as the Hopcroft-Karp algorithm (Hopcroft and Karp 1973), as well as letting the user find matchings in weighted bipartite graphs using the Hungarian method (Kuhn 1955). Similarly, Halim (2011) (C) also lets the user find matchings in weighted graphs with their tool, but also does not account for two-sided preferences.

Tool	Problem Classes	Upload Instances	Random Instances	Web Interface	Export Matching	Multi-Algorithm
A	Graphs	No	No	Yes	No	No
B	Weighted Graphs	Yes	Yes	Yes	Yes	No
C	Weighted Graphs	Yes	Yes	Yes	No	Yes
D	SM, HR	Yes	Yes	Yes	No	Yes
E	HA, SM, HR	No	Yes	Yes	No	No
F	SPA	Yes	No	No	Yes	No
G	SPA	Yes	No	Yes	Yes	No
H, I, J	SPA	Yes	No	No	Yes	No
Toolkit	HA, SM, HR, SR	Yes	Yes	Yes	Yes	Yes

Table 2.1: *Comparison of Existing Solvers*

The Matching Algorithm Visualiser (Lau 2021) (D) can solve stable marriage and hospital resident instances and visualises the algorithm in a user-friendly way. Similarly, Ferris and Hosseini (2020) (E) also provide stable matching algorithms for stable marriage, hospital resident, and house allocation in their tool, but does not allow the user to upload instances or download results.

Regarding SPA, ProjectsGeek (2017) (F) provides an offline software tool promising to solve SPA instances, but the input, algorithms, and output are unclear. Morey (2021) (G) provides a special-purpose web application for SPA-S that lets the user upload instances in separate files and claims to use an implementation of the student-oriented stable matching algorithm by Abraham et al. (2007) (H). As previously noted, Lavery (2003) (I) and Kwanashie et al. (2015) (J) implemented SPA-student and one-sided profile- and cost-optimal algorithms in Java, but neither is easily accessible without setting up the special-purpose code locally.

Finally, the Matching Algorithm Toolkit (University of Glasgow 2023) (Toolkit) is a general-purpose matching web application developed over time within the University of Glasgow's School of Computing Science by at least 18 past contributors. It lets users find matchings for instances of the house allocation, hospital resident, and stable roommate problem classes using a range of algorithms for each, upload or randomly generate instances, and show or download the results. There have been at least six BSc, MSc, and MSci projects dedicated to developing parts of the system, many still in use, some discontinued, and some redeveloped. With almost 500 Java classes in the back-end code alone, this project has grown to a substantial software project accumulating many state-of-the-art matching algorithms not easily available elsewhere.

The project started out as a command-line version in an effort to make matching algorithms available in a more standardised way. Remta (2010) then designed and implemented a web back-end to bring everything together and originally considered SPA but did not integrate it. Lazarov (2018) then built a front-end web application and extended the existing back-end. Several other students and researchers have extended the application over time, for example by adding more algorithm implementations and integrating graphs and visualisations. However, although Kwanashie (2015) has already implemented suitable one-sided SPA algorithms and Lavery (2003) and Zhang (2019) already implemented student- and lecturer-oriented two-sided stable SPA-S algorithms, respectively, SPA is not a part of the Toolkit so far. Another problem is that in terms of version control, there are many different offline versions of the codebase used by the online version and many extensions such as maximum popular matching for SM by Yang (2022) that are not actually reflected in the live version.

It is assumed that other internal special-purpose student allocation applications exist across universities, but to the best of the authors knowledge, no public ones follow the SPA-S model and none are general-purpose online matching web applications.

3 Analysis and Requirements

The background section has shown that there are no general purpose web application that provides a selection of SPA algorithms. However, the Toolkit is a user friendly web application that already provides other matching problem classes and is therefore a suitable base to extend to SPA. Furthermore, the Toolkit also has potential to be extended in other ways, and needs some crucial steps to comply with general software engineering best practices and to increase the overall usability, as this section will show.

3.1 Analysis of the Toolkit

The Toolkit follows the classic client–server model and consists of a front-end (Matching Algorithm Toolkit Web Application, short MATWA) and a back-end (Matching Algorithm Toolkit Web Service, short MATWS). The current available problem classes are House Allocation, Hospital Residents, Stable Marriage, and Stable Roommates and the user is able to choose between uploading their own problem instance or generating problem instances randomly.

Figure 3.1 shows a typical interaction between the user in the front-end and the server in the back-end when requesting to solve a randomly generated instance. After the user selects a problem class and chooses to generate a random instance and its relevant parameters, the back-end reads and verifies the instance parameters, generates the problem instance model randomly, converts the model to a string and verifies it, determines which algorithms are applicable to this instance, and returns the instance string and list of available algorithms. The user then selects which algorithms to run, which triggers the back-end to convert the instance string back to an instance model, generates solver objects for each requested algorithm, solves the instance, calculates the matching statistics, and generates relevant results. The user is then displayed the results in the front-end. Note that these are common steps between all problem classes.

3.1.1 Back-End

The MATWS is a standalone REST API that provides three public endpoints, `FileCheck`, `ParameterCheck`, and `AlgorithmRunner`. A REST API is stateless and provides endpoints in form of exposed URLs whose input and output format is of a specified format, usually specified as JSON or XML.

- `FileCheck` is called when the user chose to upload their own instances. It takes the problem class and instance strings as input and returns a status, instance strings, and list of applicable algorithms.
- `ParameterCheck` is called when the user chose to generate random instances. It takes the problem class and relevant parameters as input and returns a status, instance strings, and list of applicable algorithms just like `FileCheck`.
- `AlgorithmRunner` is called when the user chose a subset of the applicable algorithms to their instances. It takes the problem class, algorithm name, and instance strings as input, and returns a status, algorithm name, number of matchings found, list of those matchings, statistics, and graphs (if applicable).

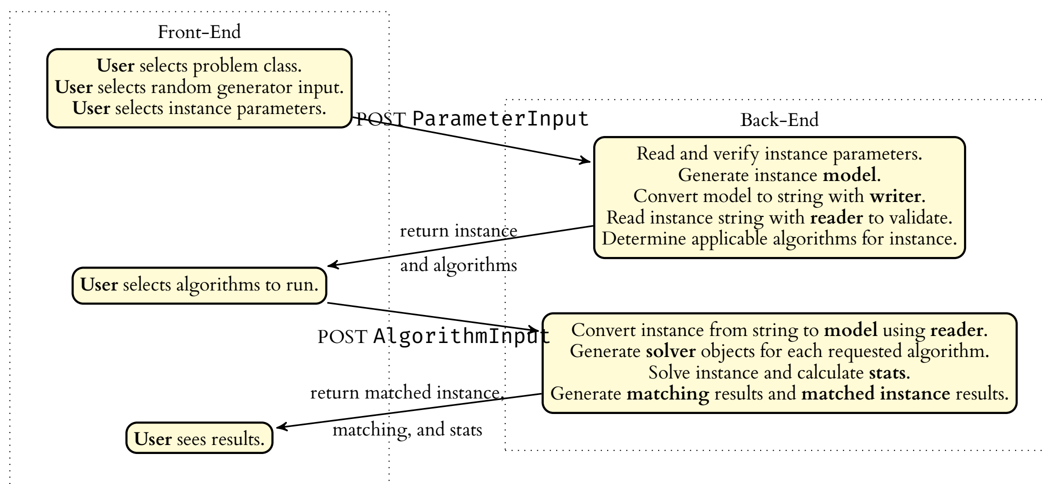


Figure 3.1: *Typical interaction when solving a random instance with the Toolkit*

The service is built on top of Spring Boot, a Spring-based Java framework that simplifies the configuration overhead when building a REST API. Naturally, the back-end software is written in Java, tested using JUnit, and organised with Maven for dependency management and lifecycle processes.

Many different object-oriented software engineering patterns are followed. For example, most objects are created from abstract factories and builders (creational patterns). From the structural patterns, the adapter pattern is used to port between the core service logic and different legacy implementations of the algorithms, and the composite pattern is leveraged to to deal with different problem classes, as each is treated the same in the API logic flow but requires different reader and writer classes, for example. From the behavioural patterns, the state pattern allows objects to change their behaviour based on the specific problem instance parameters, for example the self-validation method of problem instance models based on whether it has one- or two-sided preferences.

Note that this service is a substantial piece of software, with almost 500 java classes implemented. The overall organisation can be summarised as follows:

- **API:** Abstract interfaces for every implemented class, following creational engineering patterns. This is not to be confused with the web service API.
- **Legacy:** Various algorithm implementations from different engineers for all problem classes, many with their own utilities such as readers, writers, and further algorithms and models they depend on.
- **Matchings:** Implementations of a generic matching, matching stats, and a respective factory.
- **Generators:** Implementations of generators and instance parameters for all problem classes.
- **Models:** Implementations of models for agents, preferences, instances, networks, etc.
- **Readers:** Readers to convert from an instance string to an instance model for every problem class as well as abstract readers, factories, and utilities.
- **Writers:** Writers to convert model objects to strings, such as for instances, matchings, matched instances, and statistics. For each type, there exist problem-class-specific implementations, as well as legacy-code specific implementations for some problem classes, and factories. Furthermore, for each implementation, there generally exists a legacy GUI version and an HTML version which is used for the current web service.

- **Solvers:** Problem-class specific solver classes that inheret from abstract solvers and are generated from factories. Each legacy algorithm has its own adapter class to convert instances and matchings appropriately.
- **Webservice:** Request and response formats of the web service and its core logic (`SpringModel`).

3.1.2 Front-End

The MATWA is a standalone Django application serving HTML and static files (CSS, JavaScript) to the user. It is written in Python and uses jQuery and Bootstrap components and furthermore relies on JavaScript dependencies for features such as file input, result zipping, and graph visualisation.

The main functionalities are available on a single page, which makes several different API calls depending on the user's actions. The main tabs within this page open dynamically and are problem class selection, input, parameters, algorithms, and results, which lines up with Figure 3.1. They open sequentially and the user cannot jump ahead and skip a step, but can go back and modify previous steps. At any time, there is at most one tab expanded to the user. Other features are opening the user manual on a separate page, saving the instances, and saving the results.

The design of the front-end is kept minimal, with a simple grey-green color scheme. User input is handled through mouse clicks and number fields only.

The project is organised following standard Django guidance. Note that the parameter forms for random instance generation are problem-class-specific and are handled using separate HTML forms that are dynamically imported into the main page.

3.1.3 Hosting

The front- and back-end services are both hosted on the same physical machine, Mithril, an Ubuntu server located in the School of Computing Science at the University of Glasgow. Mithril hosts various projects associated with Professor David Manlove, each of which is run in a separate virtual machine (*matwa* for MATWA and MATWS), and runs nginx as a common front-end for all web apps to serve static files. The URLs `matwa.optimalmatching.com` and `matws.optimalmatching.com` are being redirected to ports 8008 and 9115 on *matwa*, respectively. These are served from *matwa* using a Django production server and the compiled Spring Boot application, both started automatically on boot from systemmd service files.

Currently, there are no continuous integration and deployment procedures in place for the Toolkit. The server is being updated manually by connecting to the campus network, either physically or using a VPN connection, then connecting to the school's sibu network, again, either physically or using an SSH connection, and finally connecting to Mithril using an SSH connection. Then, the source code stored at */opt/boris/* can be changed directly. HTML files are synced automatically, other static files need to be synced with nginx outside of *matwa*, and the back-end needs to be recompiled and the service restarted. Access to developers is given by current maintainers on request.

3.1.4 Different Versions

Similarly to the deployment, there is no version control system in place for the Toolkit. As mentioned in the background section, there have been at least 18 contributors on the project, with no strict project management or software engineering requirements. Over time, some inconsistencies and bugs came along due to the complexity of the project. Similarly, there are multiple different offline codebases of the Toolkit, each with differences ranging from minor fixes, to major algorithm contributions. For example, one MSc student implemented a maximum popular matching algorithm for the SM problem and attempted to integrate it in the codebase,

but the changes were never synced with the live version. Also, the command line toolkit has some more algorithms, which are not integrated with its GUI or the Toolkit.

3.1.5 Gaps in the current System

The major gaps in the current system can be summarised in three blocks: the SPA problem class is missing completely, there are bugs and minor features that, if addressed, would greatly increase the usability and correctness of the system, and there are engineering challenges such as version control and deployment. One goal is to combine the most significant features and extensions into a single version, set up a proper git-based version control system, and extend the web application with usability-increasing features and the SPA problem class.

3.2 Requirements

The requirements derived from the above analysis combined with the user stories below are grouped into functional and non-functional requirements and prioritised using the MoSCoW method.

3.2.1 User Stories

As a student
I want to solve random instances of the SPA problem
So that I understand what a possible outcome of the algorithm looks like.

As a project coordinator
I want to solve custom instances of the SPA problem
So that I can find a stable matching for my instance or report that none could be found.

As a researcher
I want to try out different parameters when generating random instances of the SPA problem
So that I can run matching algorithms on them.

As a developer
I want to contribute to the codebase using version control
So that software engineering best practices can be followed.

As the application owner
I want to have a universal version of the application
So that all developed features are available in this version.

As a user
I want to see the most current stable version of the application
So that I can use all the available features.

As a user
I want to have generous guidance from the system
So that I know how to use it appropriately.

3.2.2 Functional Requirements

Must haves:

- The system must allow the user to generate random instances of SPA.
- The system must allow the user to solve the SPA problem.
- The system must be able to display the results of the SPA matching algorithm to the user.

Should haves:

- The system should be able to check whether the SPA output matching is stable.
- The system should be able to output the SPA matchings to a file.
- The codebase should be hosted using git for version control.
- For each of the four problem classes, the system should allow the user to input custom instance data directly without uploading a file.

Could haves:

- The resulting codebase could be merging previous extensions of the project into a single version.
- The resulting codebase could address and fix some known bugs.

Would be nice to haves:

- It would be nice to have a continuous deployment procedure that integrates with the version control system.
- It would be nice to provide the user with more guidance on what everything means, the formatting standards and input requirements.

Amended would be nice to haves (agreed after project start):

- It would be nice if the system could work with SPA instances with one-sided and two-sided (SPA-S) preferences.
- It would be nice if the system could find both student-optimal and lecturer-optimal stable matchings of suitable SPA-S instances.
- It would be nice if the system could find minimum cost, greedy, and generous matchings.
- It would be nice to allow instances with or without IDs for all problem classes.

3.2.3 Non-Functional Requirements

- The extensions should integrate well with the existing versions of the project and act as a single source of truth for the most up-to-date version and available features.
- The extensions should run reasonably fast (instances with less than 50 agents should be solvable in less than one second) on the existing infrastructure.
- Any adaptions to the interface should remain intuitive.

4 Design

The requirements were translated into design choices that can be summarised under the SPA extension and some general improvements.

4.1 Student-Project Allocation Problem Class

The design and the user experience of the SPA problem class were chosen to be as close as possible to the existing problem classes to keep the application and its appearance coherent. Therefore, there were generally no significant wireframes created before the changes were made. All final designs for a SPA example flow can be seen in Figures A.1 - A.7.

4.1.1 User Interface

The user experience flow can be seen in Figure 4.1; note that the user can also go back to any previous step at any point.

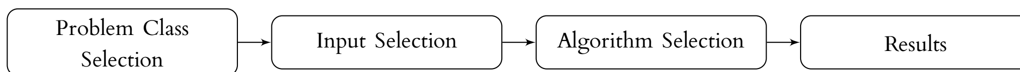


Figure 4.1: *SPA User Experience Flow*

All user choices, except the parameter selection in case of random instance generation, are made through simple square buttons with rounded corners. The algorithm selection step allows the user to select multiple algorithms if more than one is applicable to the instance. The results are then displayed using a browser-like tab design, where each tab contains the results of the respective algorithm.

The parameter selection lets the users choose attributes of the random instance to be generated by the system. Specifically, the system allows the user to choose:

- **Number of Agents:** the number of students, projects, and lecturers.
- **Total Capacities:** the total capacities of the projects and lecturers. These are distributed randomly by the system.
- **Even Distribution:** when this checkbox is ticked, the total capacities are distributed evenly across the projects and lecturers, up to off-by-one errors due to integer divisibility.
- **Student Preference List Lengths:** student preferences can be restricted to a fixed number or a range of projects. In the case of a range, the lengths of individual student preferences in a generated instance vary randomly between the chosen bounds.
- **One-Sided Preferences:** here, the generator does not generate lecturer preferences over students if ticked.
- **Skewness Factors:** these influence the difference in popularity between the least and most popular projects and, if not one-sided, students. These can be controlled separately for the agent groups.

- **Probabilities of Ties:** both student preferences and, if not one-sided, lecturer preferences can have ties. The probability factor here determines the expected number of ties in the preferences, controllable for either agent group individually.
- **Number of Instances:** generally, the user can choose to generate multiple instances at once. Note that the main results overview will not apply in this case, as the user will only be presented with summary statistics. The remaining section will therefore focus on single instances.

Although the design and layout of the results tab are adapted from the other problem classes, multiple changes were made. Figure A.6 shows an example results tab with the relevant matching statistics, the matching itself as an expandable section, and the instance itself with assigned agents highlighted in the preference lists. Noteworthy are the cost and profile distinctions between different agent groups, which is something the system did not do previously. Furthermore, the statistics are shown in a sensible order, whereas previously they were presented in random order due to specific implementation details. Apart from this, the matching itself is presented to the user in text form, whereas the other problem classes just present agent-agent pairs using IDs. This was chosen to increase the usability of the system, as pairs would be confusing given that the problem involves three sets of agents. It could be unclear to new users whether the system shows student-project or student-lecturer pairs.

4.1.2 Abstract Models

As outlined in the analysis of the existing Toolkit, the back-end codebase abstracts most classes and operations through high-level models in form of interfaces and abstract classes. Some core models to consider in the integration of the SPA problem class are:

- **Problem Model:** In order to integrate SPA, it requires its own problem class and instance model that extends and implements the general problem model. Some of the features needed, for example, are storing the student, lecturer, and project agents with their respective links and capacities, providing the ability to retrieve them in an indexed format, returning summary statistics such as the total number of students, lecturers, projects, and capacities, and finally providing the capabilities to check the model and making it consistent. When checking the model, parameters about the existence of ties, completeness, and one-sidedness should be detected and set within the model. When making the model consistent, on the student side, all projects should be removed from the student's preference list if the student is not on the supervising lecturer's preference list, i.e. is considered unacceptable by the lecturer. On the lecturer side, all students that do not find any of the projects they offer acceptable should be removed from their preference lists. This ensures efficient and correct computations.
- **Agent Model:** The agent model is an abstraction that represents either a student, lecturer, or project in the system. Although not every information is applicable to every type of agent, the model captures, in particular, the agent name and ID, their preference list, their original and remaining capacities, their engagements and number of links (for example projects supervised by a lecturer), and provides functionality to add any of this information, get the first and last agents on their preference list, get the preference list index of a specific agent, and many more that are required by the algorithms.
- **Instance Parameter Model:** The instance parameters capture all of the information required to fully determine the settings of the SPA random instance generator and the model provides the functionality to get and set those values and check if present and of the appropriate type. The parameters are mainly the ones described in the user interface section.
- **Matching Model:** The matching model captures a matching in the system. It is a store of agent-agent mappings, indexed on one type of agent - in SPA it is indexed on students

and contains student-project pairs. When querying the model, one can retrieve the whole matching, check whether a specific student is matched, and if so, which project they are matched with.

- **Matching Stats Model:** The statistics model contains information about a specific matching and can compute and provide these as requested. In particular, it stores the total costs, sizes, and profiles of the matching.

4.1.3 Instance Generator

Before designing the instance generator itself, a suitable instance format was required. Historically, instances have been stored in text files, so a text-based schema was the go-to option. In the existing Toolkit, HA and SR use similar schemata, while HR uses a slightly different schema with more flexibility due to agent IDs and capacities. Therefore, the HR format was chosen as a basis, and extended to have an additional section dealing with the project capacities and supervisors.

```
#Students #Projects #Lecturers
StuId: Pref1 Pref2 (...)
StuId: Pref1 Pref2 (...)
StuId: Pref1 Pref2 (...)
LecId: LecCapacity: Pref1 Pref2 (...)
LecId: LecCapacity: Pref1 Pref2 (...)
ProjId: ProjCapacity: LecId
ProjId: ProjCapacity: LecId
ProjId: ProjCapacity: LecId
ProjId: ProjCapacity: LecId
```

Figure 4.2: *Toolkit Format*

```
3 4 2
1: 1 2
2: 2 3
3: 1 3
1: 2: 1 2 3
2: 1: 2 1 3
1: 1: 1
2: 2: 1
3: 2: 2
4: 1: 2
```

Figure 4.3: *Instance Example*

```
3 4 2
1: (1 2)
2: 2 3
3: (1 3)
1: 2:
2: 1:
1: 1: 1
2: 2: 1
3: 2: 2
4: 1: 2
```

Figure 4.4: *Ties Example*

The resulting format can be seen in Figure 4.2, where the first line specifies three numbers x, y, z separated by a space. x, specifies the number of students, y, specifies the number of projects, and z specifies the number of lecturers. The following x lines are for the preference lists of the students. In each, the student ID is directly followed by a colon and a space, and the preferences follow as a space-separated list. Ties are represented by putting brackets around the preference entries. The following y lines are for the preference lists of the lecturers. In each, the lecturer ID is directly followed by a colon and a space, then the lecturer's capacity, a colon and a space, and finally the preferences as a space-separated list. Here, too, ties are represented by putting brackets around the preference entries. If the instance only has one-sided preferences, all lecturer preference lists are empty. Finally, the last z lines capture the project information. Each line starts with the project ID followed by a colon and space, then the project capacity followed by a colon and space and finally the lecturer ID who supervises the project.

Figure 4.3 presents a conversion of the previously shown SPA instance in Figure 2.4 to the Toolkit instance format and Figure 4.4 represents the same underlying instance but with one-sided preferences and ties (students 1 and 3 find their ranked projects equally good).

The random instance generator itself takes the parameters outlined in the User Interface section. Depending on whether the capacities should be distributed equally, the algorithm tries to do so or otherwise assign them randomly to the lecturers and projects. The core logic of the generator lies in the creation of the preference lists. For every student and lecturer's preference list, after its length is randomly determined in the specified range, project and student IDs, respectively are picked randomly. The probability distributions follow a linear function determined by skewness factors k_p, k_s, with the probabilities of picking the most popular project and student being k_p and k_s times as high as the probabilities of picking the least popular project and student. This results in skewed preference lists, where popular projects and students are more likely to appear in the first rank positions than less popular projects and students, and more likely to appear at all in the case of incomplete preference lists.

4.1.4 One-Sided Solvers

The one-sided solvers will follow the minimum cost maximum flow-based algorithms as presented in Kwanashie et al. (2015) to find cost-optimal, greedy, and generous maximum matchings. The corresponding graph $G = (V, E)$ is constructed by taking the vertex set $V = S \cup P \cup L \cup \{s, t\}$ and setting up edges e_{s,s_i} from source s to students s_i with cost zero and capacity one, ensuring that every student is matched to at most one project. Then let edges e_{s_i,p_j} denote the preferences from students over projects, with the cost determined based on the optimality criterion and capacity one. Furthermore, let edges e_{p_j,l_k} denote the project-lecturer links, with every project being linked to exactly one lecturer and the capacity determined by the project's capacity. Finally, let edges $e_{l_k,t}$ denote the lecturer links to the sink, with cost zero and the capacity determined by the lecturer's capacity. The last two capacities enforce the capacity restrictions. The student-project cost in the case of cost optimality is simply the rank position r of the project in the student's preference list, whereas for greedy matchings it is $n^{R-1} - n^{R-r}$ for where R is the maximum length of any students' preference list, and for generous matchings it is n^{r-1}. The authors argue why their respective cost functions ensure optimal criteria satisfaction when the minimum cost maximum flow in the respective network is converted to a matching by taking all $M = \{(s_i, p_j)$ such that $\text{flow}(e_{s_i,p_j}) = 1\}$. Figure 4.5 shows such a network with edge labels $(cost, capacity)$, constructed from the instance shown in Figure 4.4 and with a cost-optimal flow highlighted. Specifically, this results in the matching $M = \{(s_1, p_1), (s_2, p_2), (s_3, p_3)\}$ with cost 4.

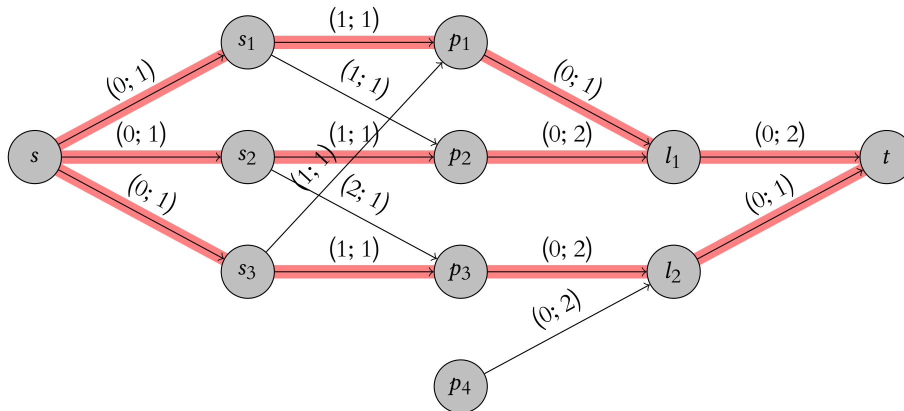


Figure 4.5: *A Cost-optimal Flow*

The one-sided solver in the Toolkit deals with this by taking a SPA instance in form of the system's model, converting it to an s-t network as described, running the algorithms from the authors which are treated as legacy code in the system, and finally converting the flow into the system's matching model using the procedure described.

4.1.5 Two-Sided Solvers

For the two-sided solvers, the two classic SPA-S stable matching algorithms SPA-student and SPA-lecturer by Abraham et al. (2007) that allow incomplete student preferences but no ties were chosen due to their efficiency and universality. Here, too, legacy code is used in the Toolkit.

The pseudocodes for the algorithms can be found in Figures A.8, A.9. Let the projected preference list of a project be the preference list of its supervising lecturer, restricted to the students that find this project acceptable. On a high level, the student-oriented algorithms starts with all students unassigned. Next, while there are unassigned students with non-empty preference lists, they apply to projects on their list, leading to provisional assignments. These assignments will be broken if project or lecturer capacity constraints are violated, removing the worst student

provisionally assigned to the project or lecturer, respectively. If a project or lecturer becomes full, preference list entries are removed from the students and projected preference lists to ensure there will be no applicants to this project or lecturer, respectively, that are worse than its worst current assignee.

Similarly, the lecturer-oriented algorithm also starts with all students unassigned. Now, while there is some lecturer with free capacity that offers a project with free capacity and there is a student on that project's projected preference list that is not provisionally assigned to this project, then this student is either free or prefers a project with free capacity offered by this lecturer to their current assignment. In this case, the first such student in the lecturer's preference list will be provisionally assigned to the first such project in that student's preference list, breaking their provisional assignment if necessary. Then, all less preferred projects from the student's preference list and the student from the respective projects' projected preference lists are removed. This ensures that no project is subsequently offered to that student that is worse than their current provisional assignment.

Both algorithms terminate with the final provisional assignments being stable matchings. On a high level, the software flow is the following: the SPA instance in form of the system's model is converted to the respective legacy code models, solved using the algorithm, returned, and converted back to the Toolkit's matching model.

4.1.6 Readers and Writers

The Toolkit requires an instance reader for SPA which takes as input an instance string and outputs a problem class model, as well as a writer for each of matched instance, matching, and matching statistics which take as input the respective internal models and output strings to be presented by the front-end, displaying the respective information in a structured format.

The design of the instance reader is fully determined by the input format outlined in the instance generator design section and works by breaking down the instance string into the respective agents and their information, converting them into agent models, and capturing the instance as a SPA problem class instance model.

The design of the writers follows the expected output format outlined in the results part of the user interface design section. The matched instance presents the agents and their preferences lists, with the matched agents highlighted. The matching presents the student-project pairs themselves with the associated supervising lecturers. The matching statistics present the size, costs, and profiles of the agents. Note that the output format is adjusted dynamically based on whether the Toolkit is dealing with a one-sided or a two-sided instance.

4.2 User Input and General Usability

Text Box The file upload proved not flexible enough for past users of the Toolkit. A core contribution on top of the newly implemented SPA problem class is a new way for users to input their problem instances. In the input type selection, the user can now choose to input from a text box as shown in Figure 4.6, in addition to the existing file input and random instance generation. This text box is available for all problem classes and lets the users copy and paste their own instances without uploading a file. Furthermore, the user can choose to dynamically add and remove agent IDs at the beginning of the preference lines.

Flexibility Another contribution takes away some of the restrictions on the uploaded and pasted instances to increase usability. For example, the system previously let users upload text files created on Linux machines but broke when users tried to upload text files created on Windows machines. Furthermore, the input format requires a very strict format, for example imposing empty space characters as separators and disallowing tab spaces. A major inconsistency in the Toolkit was that

Figure 4.6: Text Box Input Example for an HR Instance

the input format for HA and SR instances required no agent IDs in the preference lines, but did require these for HR and the first SPA design. The different types of input were incompatible between the different problem classes for no reason, as the number of agents is given in the first input line either way. The new design addresses all of these issues, both in the file uploading and the new text box input features. Specifically, instances can now be entered with or without IDs in the preference lines for all problem classes.

Approachability The previous user input design for the random instance generators used text boxes for all number input fields, except for a checkbox for the even capacity distributions. This led to confusion for first-time users, as the number of parameters to configure was overwhelming and required a good understanding of how they affect the problem instance. Therefore, a new design was developed, combining number fields with checkboxes and range slider fields as shown in Figure 4.7 that let the user drag the slider from left to right to choose a value, with the exact value indicated in red. This restricts the input domain to a valid range by default, but on top of that, fields with restrictions on the domains were extended by a range indicator that tells the user where the input number must lie. Furthermore, default values were added to, for example, the number of agents fields, which the user can modify manually, to reduce the initial complexity.

Figure 4.7: SPA Parameter Input Design

User Manual Finally, the user manual, which is a page in the Toolkit, was improved by making it more readable and updating it to include all up-to-date information. Specifically, large paragraphs of text were partitioned into logical sections with headers and spacing and improved through concise and clear language. Furthermore, a consistent writing and design style was imposed. Also, a parameter guide was written and added that explains all non-trivial parameters of the random generator tab for all problem classes. The information itself was updated to any new changes made to the system and gives credit to all other contributors.

5 Implementation

To set up the Toolkit and start developing, a copy of codebase hosted on the live server was received, as well as copies of student extensions of the codebase and other student implementations unrelated to the Toolkit. To mirror the system environment, a terminal-only virtual machine with an Ubuntu Live Server 18.04.6 image was created using VirtualBox with bridged network mode, and the setup instructions for the front- and back-end services by Lazarov (2018) followed, including installing relevant dependencies as outlined in the new setup instructions in the Appendix and mentioned in the Toolkit Analysis. For development, VS Code was installed on the main Windows OS machine, using the SSH extension to connect to the virtual machine. On the virtual machine, Python 2 and pip2, as well as relevant requirements, were installed, and a new Python environment was created.

When following the existing setup instructions, there was a compiler error due to a false Java Developer Kit version, which turned out to be a minor fix. After changing the web endpoints according to the respective locally assigned IPs, both MATWA and MATWS loaded successfully on the virtual machine and could be reached from a local browser on the Windows OS.

Note that in this section, relevant class, function and variable names will be in `typewriter font`.

5.1 Student-Project Allocation Problem Class

5.1.1 User Interface

The implementation of the user experience with the SPA problem class was, similar to the design, chosen to be as close as possible to the existing problem classes to keep the application coherent.

In MATWA, the `index.html` file was extended by adding a new problem class section using bootstrap components. Furthermore, the bootstrap layout of the existing components had to be adjusted to allow enough space. Then, a new form was added that captures all of the input fields for the SPA random instance generator, using a mix of number, check box, and range slider input fields to be consistent with existing input types in the application. Every field is fitted in the bootstrap structure, is assigned an id and name, set to be required, and provided with an explanation label and a default value. Additionally, some inputs have an associated invisible help block which can be populated dynamically when the user enters an invalid number.

Next, the existing jQuery logic was adjusted to include SPA. This entails adding additional user flows and navigating to the right steps on user action. Additionally, in the `AppClass` JavaScript logic, the parameters of the SPA form are validated to be of the right type, and some logic checking is performed directly such as whether the project number exceeds the project capacity. In these cases, the user is then prompted to enter a valid number, after being provided a message detailing why the current one is not allowed, using the parameter's help block.

It was decided that in the results section, it would be best to show the student and lecturer costs and profiles individually, as well as a combined cost and profile. After implementing this through existing logic, however, it displayed these statistics in random order because they are generated from a Java HashMap in back-end which returns unordered key-value pairs. In order to show

the statistics in a structured and readable manner, the back-end information was split and sorted based on the statistic names.

5.1.2 Models and Flows

With almost 500 Java classes implemented in the back-end codebase due to various extensions, special cases, and a high level of abstraction, implementing a new problem class turned out to be on the one side highly non-trivial and on the other side not requiring a complete redevelopment of the codebase, but rather natural extensions of existing flows and models.

There are three main parts to the back-end implementation. One, there are abstract models implemented through Java interfaces for almost every component, aiming at making the varying implementations between problem classes behave similarly in the program flow. Second, there is the core implementation logic inherited and extended from the original command-line toolkit which provides all of the functionality such as generating and solving instances. Finally, there is the implementation of the web service itself which handles requests and provides corresponding responses by handling the internal orchestration of the core logic through program flows. Each part required separate adjustments, with each of them going hand in hand with the others.

Before talking about the core implementation, it is worth outlining the internal flow of the programs based on the request orchestration. The requests from the front-end service are either of type `FileCheck`, `ParameterCheck`, or `AlgorithmRunner`, with each of them broadly explained previously in the Analysis section. The orchestration is happening inside the `SpringModel` class, with almost 2000 lines of code. The program flow for SPA is closely modelled after the other problem classes, and every problem class is dynamically handled within the same function.

`FileCheck` parses the instance string passed in, verifies it, sets relevant internal flags, and returns all available algorithms to the instance based on these flags and algorithm filtering. Specifically, the implementation of the program can be summarised as follows, with some relevant class names pointed out:

1. Parse Input
   (a) Populate an object of abstract type `InstanceReader` with a new `DefaultSPAReader` object generated from a factory class.
   (b) Generate and populate a `Model` object of the problem using the newly generated reader and make it consistent.
2. Set flags
   (a) Check the `Model` and determine the instance parameters such as presence of ties, one-sidedness, and complete preference lists.
3. Filter available Algorithms
   (a) Based on the problem class, filter the list of available algorithms based on whether they apply to instances with the given parameters.
   (b) Populate `Tooltips`, i.e. high-level information on the algorithms, and return all.

The `ParameterCheck` operates similarly, but in addition, verifies the parameters and generates the random instances. Specifically,

1. Process Parameters
   (a) Parse the parameters passed from the front-end and put them in a Java HashMap while converting them to the correct data types (e.g. Integer, Float, or Boolean).
   (b) Create an `InstanceParameterSPA.Builder` object with relevant attributes to build an `InstanceParameterSPA` object from it.
2. Generate Instances
   (a) Populate an object of abstract `Generator` type with a new `GeneratorSPANew` object using a factory class.

   (b) Generate a `Model` object using the generator with the previously parsed parameters.
3. *(The next steps are the same as in* `FileCheck`*...)*

Finally, `AlgorithmRunner` initialises various variables, creates the solvers for the algorithms, solves the instances, and accumulates the matchings and matching statistics:

1. Initialise
   (a) For each algorithm, create `Solver`, `Output`, `Matching`, and `StatsAccumulator` objects.
2. Create Solvers
   (a) Populate the abstract `SPAFactory` with an object of class `DefaultSPAFactory`.
   (b) Populate the previously initialised `Solver` objects for each desired `Algorithm` object created from the `SPAFactory` using a general factory for solvers.
3. Solve
   (a) Populate an object of abstract type `MatchingWriterFactory` with an object of class `DefaultMatchingWriterFactory` and generate a `DefaultStatsWriter` from it.
   (b) Adjust to a `SPAHTMLStatsWriter` object from the `GUIMatchingWriterFactory` class.
   (c) Populate the `MatchedInstanceWriter` with a `SPAHTMLWriter` object from a respective factory class.
   (d) Populate an object of abstract type `StatsAccumulatorFactory` with an object of class `StatsAccumulatorFactoryImpl`.
   (e) Solve the instance, capturing `Matching`, `Stats`, `MatchedInstance`, and return all.

One of the essential model classes for the SPA computations is the SPA problem model itself. As previously mentioned in the design section, it extends and implements the general problem model, stores the problem instance information, and provides many different features. By starting the implementation assuming only SPA-S instances, it was easy to incorporate one-sided instances and algorithms by adding an additional flag attribute in the SPA problem model that is true if there are no lecturer preferences, and false if there are. Note that SPA-S instances are compatible with the one-sided algorithms and the lecturer preferences are simply ignored, but clearly one-sided instances are not compatible with the two-sided algorithms.

```
method makeConsistent():
    if (instance is not onesided) and (incomplete preference lists):
        for each student:
            for each acceptable project of the student:
                if student not acceptable to supervising lecturer:
                    remove project from preferenceList

        for each lecturer:
            for each acceptable student of the lecturer:
                if student does not find any project offered by the lecturer acceptable:
                    remove student from preferenceList
```

Figure 5.1: *Pseudocode for making a SPA-S Model Consistent*

One challenge was implementing the method making the model consistent, with the pseudocode shown in 5.1. This was achieved by treating the preference lists as linked lists. Furthermore, checking whether the student finds a project of the lecturer acceptable was achieved by getting all projects linked to the lecturer and checking whether any of them occur in the student's preference list.

The method checking the model is simply going through every agent's preference list of the instance, and if any of them have ties or incomplete lists (checked by comparing the number of agents against the length of their preference list), setting the appropriate internal flags.

5.1.3 Random Instance Generator

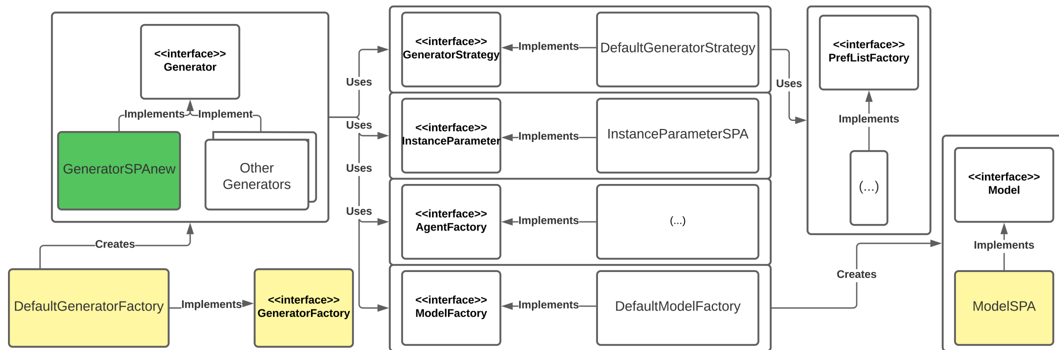


Figure 5.2: SPA Generator Class Diagram (green: new, yellow: modified)

The random instance generation process is highly dependent on legacy implementations and makes use of many abstract types and factories as can be seen in Figure 5.2. Implementing the SPA generator here was a natural extension of existing code for the other problem classes and required the modification of multiple existing classes as well as the implementation of some new ones, with the logic being very close to the HR instance generator logic.

An object of abstract type `Generator` can be populated with a `GeneratorSPAnew` object using a `GeneratorFactory`. The generator has a method to generate an instance based on an `InstanceParameterSPA` object of abstract type `InstanceParameter`. Here, first, arrays of `Agent` objects are initialised for every agent type, namely students, lecturers, and projects. Then, a `GeneratorStrategy` object is used to distribute the capacities among the projects and lecturers, which are also stored in arrays. For each agent, the `AgentFactory` is then used to create a new agent object with sequential unique ids and the generated capacities. Afterwards, each student gets attached a random `PreferenceList`. If the instance is not to be one-sided, then the same is done for the lecturers. Then, the projects are distributed among the lecturers, again using a `GeneratorStrategy` feature and storing the results in an array. Lastly, based on this distribution array, the lecturer objects are linked to their respective project objects, and the projects are set to be in a bond to their respective lecturers. Finally, the `ModelFactory` object is used to create and return an instance of a `ModelSPA` based on the students, lecturers, and agent object arrays.

5.1.4 Solvers and Algorithms

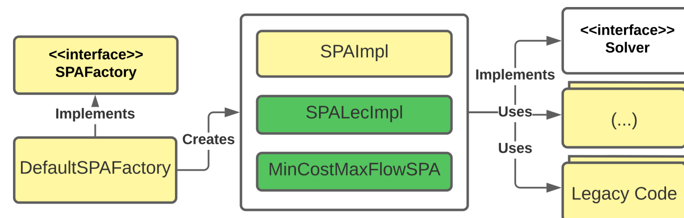


Figure 5.3: SPA Solvers Class Diagram (green: new, yellow: modified)

The implementations of algorithms to find SPA matchings were largely treated as legacy code, given that they are all accompanied by papers and extensive testing. However, each of the three legacy codebases required a varying level of effort in order to be integrated into the Toolkit, with the one-sided algorithms requiring the least work and the lecturer-optimal stable algorithm requiring the most work.

As can be seen in Figure 5.3, for each algorithm, the Toolkit has a solver class, and for each problem class, the Toolkit has a solver factory class. To fit in with the existing system, each of these has an abstract interface model as well as a specific class implementation. The solver factory class simply provides methods to generate solver objects for the available algorithms in the `SpringModel`. Specifically, for SPA, the methods `getSPA()`, `getSPALect()`, and `getSPAOneSided(AlgorithmType type)` were implemented to return implementations for the student-optimal, lecturer-optimal, and one-sided implementations respectively.

One-Sided Solvers As can be seen in the declaration of the method `getSPAOneSided`, the three one-sided algorithms are made available through the same solver class. This was possible because the only algorithmic difference is the cost function for student-project edges as noted previously. Therefore, the `AlgorithmType` has states `Greedy`, `Generous`, and `Cost` and the solver dynamically adjusts the costs.

The `MinCostMaxFlowSPA` class is taken almost directly from the legacy codebase, made to work with dynamic switching based on the type value and adjusted to fit in with the factory and Toolkit codebase. For this, six new utility classes for the s-t network that came with the legacy codebase such as `Edge` and `Network` were placed in a dedicated network model folder. The broad implementation logic when the solver is tasked to find a matching for an instance is that the instance is converted to a weighted network, checked to be valid, a residual network is generated, and a greedy algorithm is used to find maximum augmenting paths and to regenerate the residual network. Finally, when no augmenting path can be found anymore, the flow is converted into a matching using a `MatchingFactory`. Overall, there were no major issues when porting the legacy code to the Toolkit.

Student-Optimal Solver The student-optimal algorithm legacy implementation comes with its own instance format, reader, and utility classes. To avoid extensive refactoring and take advantage of the correctness testing previously conducted for the implementation, it is therefore treated as a legacy code black box to port Toolkit instances to and convert back into a Toolkit matching, if found, which aligns with legacy implementations of previously integrated algorithms.

In terms of implementation logic, the corresponding `SPAImpl` class in the Toolkit creates an instance writer object that outputs the problem `Model` in the expected string format, reads in the instance again with the legacy reader, runs the legacy matching algorithm, and iterates through the matching results while adding found pairs to a new matching model object created from the `MatchingFactory`. Potential errors and incompatibility issues are handled using classic Java error-handling techniques and custom exception classes.

As a start was previously made by Remta (2010) to integrate this algorithm in an early version of a matching API, some ideas could be adapted and required working out specific implementation details. Overall, there were also no major issues here when porting this legacy code to the Toolkit.

Lecturer-Optimal Solver The lecturer-optimal algorithm legacy implementation is also a standalone codebase. Here, the integration into the Toolkit was non-trivial, as most of the logic was implemented in a God class that did everything from instance reading from file, constructing the instance, populating the models and data structures, the solver algorithm, and writing the output to a file, in a single method. Therefore, significant refactoring was required to handle these features separately. Then, a new instance writer was implemented as a wrapper from the Toolkit format to the input expected by the legacy code, together with a new instance writer factory to align with the Toolkit practices.

The `SPALecImpl` class implementation follows the same logic as the student-optimal solver class, with some adjustments to account for implementation-specific details of the legacy code.

5.1.5 Readers and Writers

When the back-end writes SPA information, it must differentiate between writing a SPA instance, SPA matching, SPA matched instance, or SPA matching stats. Each is implemented in a separate class. Figure 5.4 shows how the instance writer is connected, but the other readers and writers follow a very similar logic.

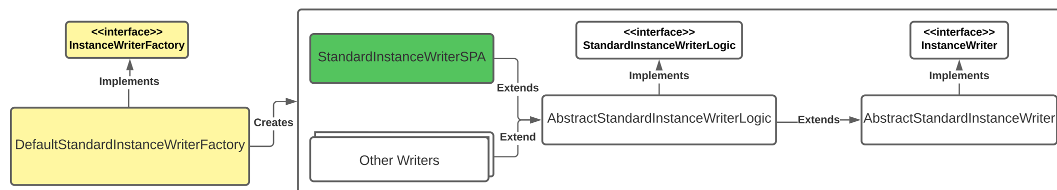


Figure 5.4: *SPA Instance Writer Class Diagram (green: new, yellow: modified)*

- `SPA Instance Writer`: The `DefaultStandardInstanceWriterFactory` is used to generate a `StandardInstanceWriterSPA` object of abstract type `InstanceWriter` which extends some abstract classes. The main method of the new writer loops through the agents and prints the information in the desired instance format by making use of existing functions to generate, for example, preference list lines.
- `SPA Matched Instance Writer`: As the matched instance writer is an extension of the instance writer, the set up is similar. In order to comply with the existing architecture of the Toolkit, an empty `MatchedInstanceWriterSPA` was set up, to be extended by the `MatchedInstanceHTMLWriterSPA`. The latter uses existing code to generate the output line by line and to deal with agent types, capacities, and IDs dynamically. Furthermore, the IDs of agents in the preferences that are matched with that agent are highlighted for the HTML output.
- `SPA Matching Stats Writer`: Here too, were HTML and a non-HTML writers implemented, specifically the `SPAMatchingStatsWriter` and the `SPAHTMLMatchingStatsWriter`. While the former does not provide any major logic, the latter initiates the calculations of matching size, student and lecturer costs, and student and lecturer profiles. These are then appended to the output in appropriate string form. Furthermore, the matching for HTML output is stored in a textual form saying which student is matched with which project, as well as the corresponding supervisor. This is implemented through a loop through the matching.
- `SPA Matching Writer`: To output a SPA matching, the existing `StandardMatchingWriterImpl` could be used without extension due to integration with the abstract matching model.

While the student profile could be easily computed using existing code, the lecturer profile had to be calculated manually. This is implemented together with the SPA cost calculations in a `Util` class, which differs from the other problem classes due to the split between student and lecturer costs. For the lecturer profile, a Java HashMap from integer to integer is created to store the counts of index positions in the matching. Then, by nature of the matching type indexing, the program loops through the matched students, gets the lecturer of their projects, queries the profile position this student has in the lecturer's preference list, and increases the respective counter in the HashMap. The HashMap is then converted back into an integer array of counts.

When the back-end reads a SPA instance, the `DefaultStandardReaderFactory` is used to generate a `StandardReaderSPA` object of abstract type `InstanceReader`. The new reader class extends the abstract class `AbstractStandardReaderLogic` which again extends the abstract class `AbstractTextReader`. This enables an abstraction so that the main method of the SPA reader just parses the first line of the input which contains the number of agent lines following, then loops

through these lines and converts them into `Agent` objects and `PrefList` objects and links them according to the input.

5.1.6 Stability Checker

Note that while the legacy code of the one-sided solvers is highly compatible with the Toolkit, the stable algorithms are treated as black box algorithms and need appropriate porting through readers and writers. To verify that this is done correctly, a stability checker is a suitable tool. Specifically, the checker should check whether given a problem instance and a matching, the matching does not admit a blocking pair. This was designed to be a simple procedure that directly follows the definition of a SPA-S blocking pair (Definition 1).

```
function isStable(model, matching):
    for each student s:
        for each acceptable project p:
            l = p.supervisor
            if (s is unmatched) or (s prefers p over s.project):
                if p is undersubscribed:
                    if l is undersubscribed:
                        return False
                    if (l is full) and (s is matched with another project from l or l prefers s to the worst student in
                        l.assigned):
                        return False
                if (p is full) and (l prefers s to the worst student in p.assigned):
                    return False
    return True
```

Figure 5.5: *SPA-S Stability Checker Routine*

Although the pseudo-code in Figure 5.5 is straightforward, the implementation turned out to be more challenging. This is because the matching is only indexed on students but not on lecturers or projects, and thus finding, for example, whether a student is matched with another project from the lecturer, or finding the worst-rated assigned student to a lecturer, are not operations implemented in the models. It, therefore, turned out to be necessary to create Java HashMap objects storing project-to-student links, lecturer-to-student links, the number of students assigned to a project, and the number of students assigned to a lecturer. Together with helper functions to find the worst ranked assigned student to either a lecturer or a project and a method to check which of two agents is preferred by a third agent, enabled the implementation of the referenced pseudo-code without any major adjustments.

5.2 General Improvements

In addition to the extension to the SPA problem class, some general extensions and improvements to the web app spanning all problem classes were implemented.

5.2.1 Maximum Popular Matching in Stable Marriage

Yang (2022) contributed a maximum popular matching algorithm implementation for the SM problem with incomplete lists, found as a special case of HR in the Toolkit, to be integrated into the Toolkit. This, however, was never integrated with the live codebase. As this was purely an algorithm addition, no front-end changes were made, and no core logic in the back-end was changed. Upon further inspection though, it turned out that by accepting the code changes

proposed by the author, the Toolkit codebase did not compile and run fully correctly anymore. Specifically, existing Unit Tests failed due to unexpected output, and a model change was needed in order for the correct algorithms to be returned. Also, some scripts were removed and loaded through an insecure http connection, which most current browsers disallow.

Overall, this was a minor adjustment to the live codebase that makes a previously implemented algorithm available to the users.

5.2.2 User Input and General Usability

Text Box The text box was implemented in such a way that no back-end changes were required. Using a classic HTML text area field in the main page, styled using the same bootstrap components that are used for the file input, the user can paste an instance in the right format directly into the text box. The user experience flow is handled fully in jQuery, with the respective tabs to open the text area and continue on to the algorithm selection tab being done through on-click handlers on respective buttons to be consistent with the existing design. Adding and removing the agent IDs is done through jQuery and JavaScript functions modifying the text area content directly. The functions first check whether IDs are currently present, and are active or not based on this. If IDs are not present but are requested by the user, the number of agents is parsed and each agent line is given a sequential numerical ID starting at one for each agent type, as well as the appropriate seperator. Similarly, if they are present but are requested not to be by the user, the agent preference lines are parsed and the ID and seperater are removed. The user input is then passed to the back-end as if it was the content of a file, using previously available features.

Flexibility In order to make the input from file and the new text box more flexible, there were some front- and back-end changes required. Previously, HA and SR admitted instances without Agent IDs, and HR, originally, SPA only admitted instances with Agent IDs. This major inconsistency in the input types was overcome by investigating the reader logic and how they interact with the agent and preference list object constructors. After making some adjustments to each of the readers and fixing a bug in the agent constructor, the system is now able to read in instances with and without IDs for all problem classes. Regarding the Windows file format error, after some investigation, it turned out to be a problem with end-of-line characters. Windows adds "\r\n" characters for return and newline, whereas Linux just adds "\n" for newline. The input reader splits the lines based on the backslash n character and was unable to deal with the remaining backslash r characters. Once discovered, this was a simple fix by converting the end-of-line characters appropriately through a regular expression in JavaScript before passing to back-end. Equally, tabs were not accepted as spaces. As tabs are not generally used in the input format, here, too, a regular expression converting tabs to regular spaces fixed the problem.

Generator Forms Default values in the random instance generator parameter forms were achieved through simply initialising the number fields with values, making them appear more user friendly than empty fields. The sliders shown in Figure 4.7 were implemented using HTML range slider input types with minimum and maximum values, as well as specifying the starting value and step size. In order to show which value is currently selected, an HTML span entry that is dynamically updated using JavaScript was created above each range slider, next to the slider label. Lastly, the range indicators for each numerical parameter were achieved by simply specifying the theoretical parameter bounds statically within the parameter label, see Figure 4.7.

User Manual Finally, the user manual relies mainly on pure HTML, with section opening and closing achieved through jQuery. The implementation details of the manual restructuring and rephrasing are restricted to introducing new sections in the HTML files and some simple CSS styling properties.

5.2.3 Continuous Integration and Development

As a first step, the live version of the codebase as hosted on the server was uploaded to a private GitLab repository that was newly created in the School's matching group's GitLab. After configuring the GitLab and adding necessary users and permissions for stakeholders, the codebase was cleaned up to only include relevant source code and not the compiled Java classes. Furthermore, a Readme and other relevant documentation was created.

Next, the deployment on the live server was investigated, following the written guide of a previous contributor to the Toolkit. Some steps were adjusted as needed, and after understanding the setup, clarifying materials were created. Furthermore, a start was made on a continuous deployment procedure, which could be done, for example, through GitLab CI/CD. However, the potential maintenance overhead through version updates and authentication issues was traded against the manual steps involved. Therefore, an efficient manual process to update the live server was developed for future contributors (see Appendix).

5.2.4 Bugs in the Previous System

As previously mentioned, as the complexity of the Toolkit system and the number of different contributors increased over time, some inconsistencies and bugs were introduced. These were kept in a list of known bugs by the maintainer and addressed by this project:

- **Input formatting:** Windows text files and tab spacing not recognised as valid input. This was already addressed in the section on input and usability.
- **SR algorithms offered for SRI:** This should not be the case for algorithms such as `All Stable Pairs`, `All Stable Matchings`, and `Egalitarian Stable Matching`. This was addressed by changing the internal algorithm selection logic in the Webservice's `SpringModel` to exclude these algorithms in the presence of ties.
- **Incorrect SR Costs:** This bug was noticed previously and upon further investigation, it turned out that the cost displayed in the SR algorithm results is double what it should be for instances with an even number of roommates larger than three. The bug was fixed on this assumption by the cost accordingly in the cost calculator utility.
- **House Capacities are not shown in Matched Instances of CHA:** Here, the matched instance writer for the Housing Allocation problem class was adjusted to show the house capacities as a mapping from `houseId` to `houseCapacity` similar to the project-capacity-lecturer information in SPA.

6 Evaluation

There were three types of evaluation performed – functional testing, empirical measurements, and a usability study.

6.1 Functional Testing

JUnit Testing JUnit tests were written for each SPA algorithm, extending the testing suites that previously existed for the system and following roughly the same logic. However, instead of storing the instances in string form in the testing suite, the instances used for testing were stored in text files to be read in, which tests more components of the system at once.

In total, there are 36 JUnit tests distributed across four test suites, testing, for example, that the API responds correctly to requests and that the matchings returned by the API for some sample instances and algorithms are of the correct form and contain the correct content.

Stability Testing The stability checker mentioned in the implementation was used to verify the outputs of the student- and lecturer-oriented algorithms. This was temporarily done in the SpringModel before returning the web request, and later using automated tests.

Using the automated tests, the student- and lecturer-oriented algorithms were used to solve over 5,000 randomly generated instances, each with 250 students, 350 projects, 50 lecturers, and total project and lecturer capacities of 500 and 350, respectively. The stability results were recorded in a file and exploratory data analysis confirmed that both algorithms produced only stable results for these instances.

Integration Testing Basic integration tests to verify the user interface changes and that the deployment was successful were performed manually by testing every change against a set of tasks and instances for each problem class. Some of these yielded issues with edge cases that were not detected previously by the compiler and JUnit tests.

Table A.1 shows an overview over the set of test cases with aims, tasks, expected outputs, and whether the tests pass.

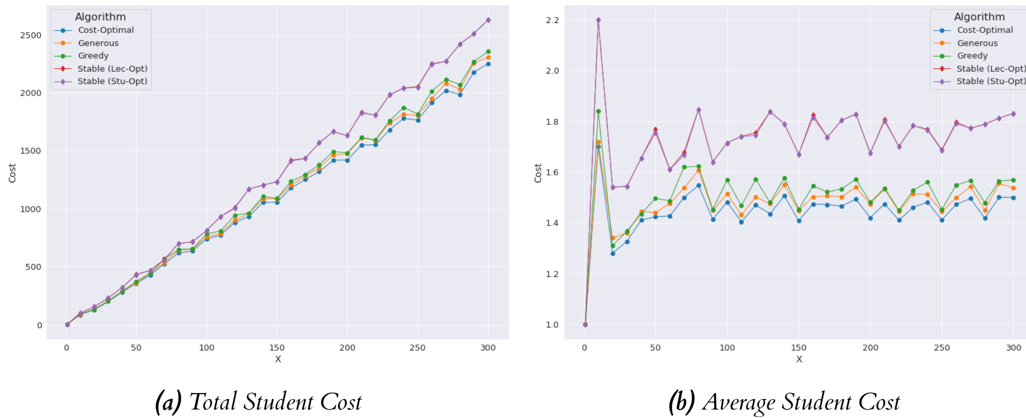
At the end of development, there were no outstanding issues detected by any of these methods.

6.2 Empirical Evaluation

The SPA algorithms were evaluated empirically against a set of generated instances with varying parameters. This was done on the TULAI Linux server in the school with an Intel i7 quad-core processor and 32GB of RAM through a remote SSH connection. Using Maven, the results were generated by directly accessing the SpringModel logic without going through the web endpoints, stored in comma-separated value files, and analysed using Python.

For all experiments, the instance sizes are parametrised by a factor $x \in \mathbb{N}$ so that in each instance, there are $5x$ students, $7x$ projects, and x lecturers, and the total project and lecturer capacities are $10x$ and $7x$, respectively. Also, none of the instances allow ties.

Varying the Instance Size The first experiment investigates how the matching sizes, costs, and computation times vary across the algorithms as the instance size grows. For this, 300 random instances of varying sizes with $1 \leq x \leq 300$ with one repetition per size were generated. The student preference list lengths vary randomly between 4 and 6, and student and project skewness factors are set to 5.



(a) Total Student Cost *(b) Average Student Cost*

Figure 6.1: *Student Costs as the Instance Size Varies*

Algorithm	**Cost**
Cost-Optimal	1.44
Generous	1.48
Greedy	1.50
Stable (Lecturer Optimal)	1.73
Stable (Student Optimal)	1.73

Table 6.1: *Average Student Cost by Algorithm*

As expected, Figure 6.1a shows that the total student cost increases as the instance size increases. However, Figure 6.1b shows that the average student cost remains roughly equal across the sizes, with some variance introduced by the fact that there is only one repetition per size. It is interesting to see, however, that there is a clear pattern in the hierarchy of algorithms, with the cost-optimal naturally yielding the lowest average student cost and the stable algorithms yielding the highest average student cost. Table 6.1 summarises the average student costs across all sizes for each algorithm and confirms this observation. The Generous and Greedy algorithms are placed between the Cost-Optimal and Stable, in this order. It is interesting to see that the average student costs are the same for both stable algorithms, which is probably due to the fact that 74.19% of the considered instances have a unique stable matching, as the student- and lecturer-oriented algorithms returned the same matching. Figure 6.2 shows that, in general, for the stable algorithms, the total lecturer cost is significantly larger than the total student cost and becomes bigger by orders of magnitude as the instance size increases.

Table 6.2 shows that for all of the selected sizes, the one-sided algorithms produce matchings where all students are matched, whereas the stable algorithms produce matchings where, on average, around 97% of students are matched.

Figure 6.3 shows that the computation times of the stable algorithms are negligible compared to the one-sided algorithms. This was expected given the theoretical linear and quadratic computational efficiency bounds, respectively.

Varying the Student Preference List Lengths The following experiment investigates how the matching sizes and costs vary across the algorithms as the student preference list lengths increase.

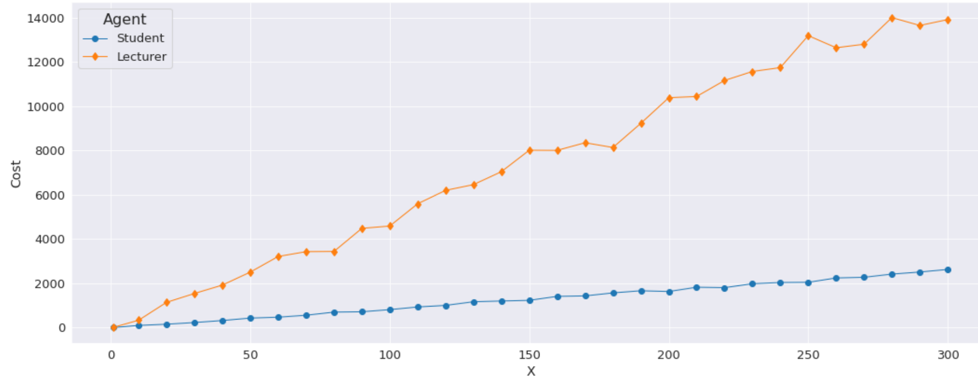


Figure 6.2: *Costs as the Instance Size Varies for the Student-Optimal Stable Algorithm*

Parameter X	**1**	**50**	**100**	**150**	**200**	**250**	**300**
# Students	5	250	500	750	1000	1250	1500
Matching Size (Stable)	5	245	474	738	973	1215	1435
Matching Size (One-Sided)	5	250	500	750	1000	1250	1500

Table 6.2: *Matching Sizes against Instance Sizes*

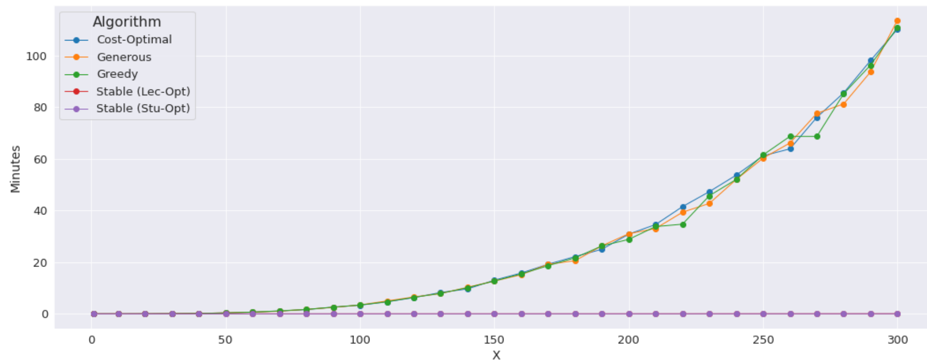
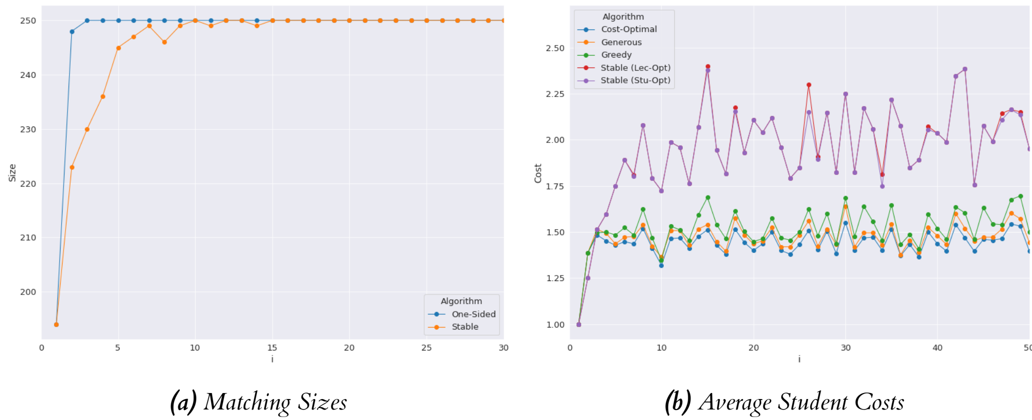


Figure 6.3: *Computation Times as the Instance Size Varies*

For this, for each $x \in \{50, 100\}$, 350 random instances of varying list lengths parametrised by a factor $i \in \mathbb{N}$ with $1 \leq i \leq 350$ with one repetition were generated. The student and project skewness factors are set to 5.

Figure 6.4a shows the results for $x = 50$ and that for both the one-sided and the stable matchings, the size generally increases as the list length increases. However, for the one-sided algorithms, all students are matched from $i = 3$, while the size produced by the stable algorithms converges to the number of students at around $i = 15$. This was expected as stability is more restrictive than cost-optimality, for example.

Regarding average student costs, Figure 6.4b shows that the stable algorithms lie closely together and the one-sided algorithms lie closely together, with the stable algorithms having significantly higher average student costs. This makes sense, as the one-sided algorithms optimise precisely for student cost and student profile while disregarding lecturer costs. Exploratory data analysis confirmed that the lecturer costs are significantly higher in matchings produced by the one-sided algorithms than by the stable algorithms. Finally, the plot also shows a trend of increasing average



(a) Matching Sizes

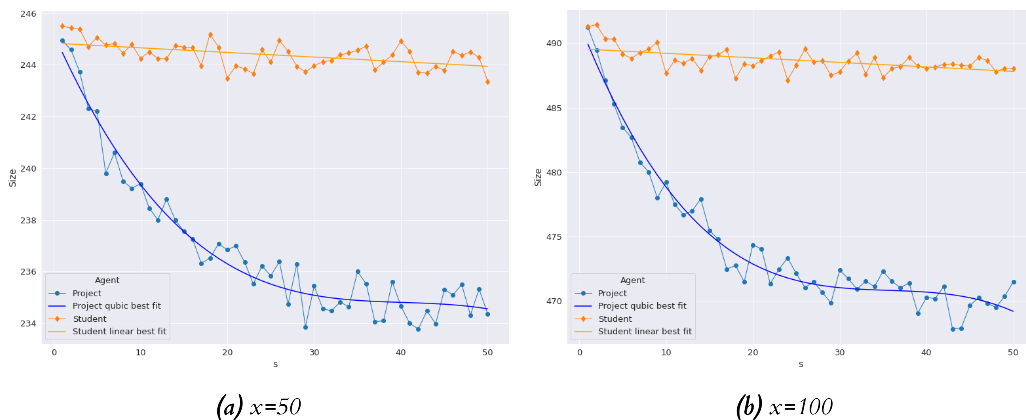
(b) Average Student Costs

Figure 6.4: *Sizes and Costs as the Student Preference List Length Varies for x=50*

student cost as i increases until around $i = 7$. This is due to the fact that the average cost is equivalent to the average profile position, and for $i = 2$, for example, the profile is restricted to $\{1, 2\}$ by definition, thereby skewing the average cost to smaller values. Similar results were found for $x = 100$.

Varying the Agent Popularity The last experiment investigates how the matching sizes vary across the algorithms as the agent popularity increase. For this, initially, for each $x \in \{50, 100\}$, there were 50 random instances generated with student skewness factor $s \in \mathbb{N}$ varying between 1 and 50, 50 random instances with project skewness factor s varying between 1 and 50, and 50 random instances with bot student and project skewness factor s varying between 1 and 50. The student preference list lengths vary randomly between 4 and 6.

Exploratory data analysis suggested that the matching sizes of the one-sided algorithms do not vary as the agent popularity varies, with all students matched. However, there was a high variance in the stable matching sizes with inconclusive trends. It was determined that varying both the student and project popularity is not as insightful as varying them individually. Therefore, the experiment was rerun with 50 repetitions for each of the 200 experiment settings, generating 10.000 random instances in total. The sizes were then averaged out over each of the 50 repetitions for each setting.



(a) x=50

(b) x=100

Figure 6.5: *Stable Matching Sizes as the Popularity Varies*

Figure 6.5 shows the results of this experiment. Similar trends can be seen for both values of x – as

the project skewness factor increases, the matching size decreases on average and starts to stabilise around $s = 35$, and as the student skewness factor increases, there is a slight linear-like decrease in the matching size, although not as strong as when varying the project skewness. This makes sense, as the student preference lists are restricted to only 4-6 projects, so that they become a strong restriction as the project popularity increases, while the lecturers find all students acceptable that find at least one of their projects acceptable, thereby having much longer preference lists, on average.

6.3 User Study

To evaluate the changes to the system and get feedback on the general usability and future improvements from potential users of the web application, a user study was designed.

Study Design The study was conducted remotely to increase the participation of researchers that might otherwise have been inaccessible. Overall, the study was designed for people at every academic stage and people were asked to rate their experience level with matching algorithms and their academic level. The ethics checklist was followed and users were briefed and debriefed appropriately.

There was an information sheet covering the basics of the Toolkit and the SPA problem class and an interactive questionnaire that gave the users simple structured tasks to complete such as generating and uploading instances, running different algorithms, and comparing and saving the results (see Appendix). Along the way, the users were asked to answer questions on the tasks, for example, whether they can see the results and what cost a specific algorithm achieves. Afterwards, the users were asked to rate multiple usability-related questions on a Likert scale, to provide feedback on the appropriateness of the application for their research (if applicable), and to give open-ended answers on what they liked most about the system or their interaction and what could be improved to enhance the usability or feature set of the system. Some of the usability-related questions are phrased the same as in the user study conducted on an earlier version of the system by Lazarov (2018) to compare the results.

Findings The study had 15 participants. Figure 6.6 shows that the users had varying levels of knowledge with regards to matching algorithms and different academic positions. Table 6.3 presents the average responses to Likert-scale questions, rated between strongly disagree (one) and strongly agree (five).

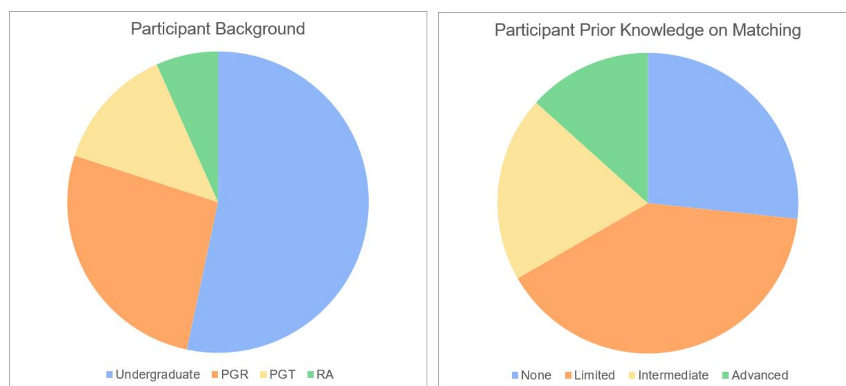


Figure 6.6: Participant Background

All users were able to upload an instance using the text box, generate stable matchings, and identify the size and cost of the matchings and that the student and lecturer optimal algorithms

Question	Median	Mean
The navigation to the previous selection tabs was intuitive.	Strongly Agree	4.40
The input fields were working well and the input types appropriate.	Strongly Agree	4.67
The instance saving worked smoothly.	Strongly Agree	4.93
The instance uploading worked smoothly.	Strongly Agree	5.00
The saving process worked smoothly.	Strongly Agree	4.86
The results in the downloaded file show what I expected them to show.	Strongly Agree	4.71
The application was easy to navigate.	Strongly Agree	4.93
Extracting information from the results was easy.	Strongly Agree	4.53
Overall, the tasks were easy to complete.	Strongly Agree	4.60

Table 6.3: *User Responses to Likert-scale Questions*

give the same matching. This required varying interactions with the results, switching tabs, and opening drop-downs.

For the random instance generator, the participants agreed or strongly agreed that the input fields and types are appropriate and working well. However, there were varying opinions on the intuitiveness of the navigation between program steps, although the majority strongly agrees with it being intuitive. Overall, the answers are very split when asked whether the greedy or the generous matching has a higher number of first and second choices fulfilled for the students. Purely theoretically, the greedy matching must have at least as many if the profile has at least two positions. Therefore, the results here indicate either that the number was equal which was not an answer option, or that the profile was interpreted incorrectly by many users.

When dealing with input from files, most or all participants strongly agreed that the instance saving and uploading and the results saving features worked smoothly. However, one user did not agree that the downloaded results show what they expected to see. Only four participants stated that they conduct or have conducted any research related to matchings. All of them would consider using the Toolkit for their research and none suggested improvements or features for the Toolkit to become a better tool for them.

Regarding usability, on average, the participants strongly agreed that the application was easy to navigate, agreed that extracting information from the results was easy, and strongly agreed that the tasks were easy to complete. Some recommendations and ideas derived from the open-ended answers on improving the system are to improve the preview of the sliders in the random generators to show the value while moving rather than after, to save the parameters rather than instances, to explain the terms used in the Toolkit better, to explain why some algorithms are hidden, and to show the preference lists in a more readable way.

These findings roughly match the study results from Lazarov (2018), with the participants here agreeing slightly stronger on average with the statement that extracting information from the results was easy. This could be either due to randomness or because of the adjustments made to order the output and the textual output for the SPA matchings.

Resulting Changes As multiple comments suggested changing the preview of the slider values, these were fixed to show the value on movement instead by adjusting the respective event handler. Furthermore, the explanations in the manual were adjusted again to improve readability, and finally, the changes to the preference list inputs mentioned in the implementation section were completed after the study.

7 Conclusion

7.1 Achievements

Overall, this project successfully improved and extended the Toolkit, resulting in a more efficient and user-friendly research and demonstration tool with greatly expanded capabilities. All requirements that were set out to be achieved are satisfied, including all nice to haves and amended nice to haves, with the exception of a fully continuous integration procedure as explained.

Mainly, the existing system was improved by navigating the complex codebase and bringing together different standalone versions of it in one unified and version-controlled system, the fixing of bugs and inconsistencies introduced by past developers, and by increasing its usability through additional user input features and more flexible input data handling, providing default values and range indicators for the random instance generator, and improving and extending the user manual. Furthermore, the system was extended to the SPA problem class which covers a highly integrated user interface, five different algorithms that have been made available with two solving for stability and three solving for profile-based optimality, the implementation of an instance reader, random instance generator, and instance, matched instance, matching, and matching statistics writers.

The functionality of the system was tested through automated and manual tests on random instances and edge cases, and the stability of the matchings was checked using a newly implemented stability checker. Extensive empirical analyses have been conducted with the five SPA algorithms, varying different parameters such as the instance sizes, student preference list lengths, and agent popularity, leading to insights into how these parameters and algorithm choices affect the matching sizes, agent costs, and computation times. Finally, by gathering feedback on the system through a user study involving 15 participants with varying levels of expertise on matching problems, the usability of the system was evaluated and concluded to be excellent. Furthermore, some suggestions for improvement were gathered, some of which were addressed during the remaining project duration.

7.2 Reflection

This project was a personal and technical challenge, but I greatly enjoyed the journey and am very happy with the achievements and final product. It was an opportunity to apply and improve many research techniques and software engineering skills learned throughout my studies and work experience to a complex system with a long history, lots of legacy code, and a real user base. Here, I got the chance to merge the work of many different researchers and contributors into a coherent solution that can be used by scientists and practitioners alike. While diving deep into the topic of matching under preferences, I learned more about algorithmics and complexity and developed and collected many research ideas that I would like to pursue in the future.

Given the large existing codebase and fragmented legacy projects, naturally, I was required to spend a long time reading up on the projects and theoretical foundations of the work before starting any new modifications. If I were to do this project again, I would spend more time planning the long-term goals for the system before starting the implementation, and therefore

prioritising the simplification of the system's implementation. It would have also been nice to have more time to include more variants of SPA such as lower-quotas, and stable algorithms for SPA-ST.

Overall, I am happy to know that the Toolkit will remain in use and that my changes are a valuable addition for researchers and other users, as evidenced by the user study.

7.3 Future work

Both on the practical and theoretical sides, there remain many extensions and topics worthy of future investigation and implementation related to the Toolkit and the SPA problem.

In terms of extending and ameliorating the Toolkit, it would be great to reduce the overall complexity of the codebase that has accrued over the many projects and contributors. By cutting legacy features and re-implementing major parts of the system, it would be easier to extend and more flexible for future features. Regarding usability, it would be good to let users input different file formats and to be more flexible overall with the input, as well as provide better error handling and error communication. By improving the implementation performance through more efficient code and architecture, larger instances could be solved without exceeding the timeout thresholds. On the algorithmic side, there are still many matching algorithms that could be implemented and integrated into the Toolkit. For example, the more efficient greedy and generous maximum matchings for SPA with one-sided preferences by Kwanashie et al. (2015), or the weakly stable matching approximation algorithm (Cooper and Manlove 2018) and the strong and super-stable polynomial time algorithms (Olaosebikan and Manlove 2018) for SPA-ST. Also, the SPA-P problem is not integrated at all yet and would need significant implementation work and possibly a slight redesign of the overall software architecture.

Theoretically, it would be interesting to extend the work on profile-based optimal matchings by Kwanashie et al. (2015) to combined profiles of students and lecturers to incorporate two-sided preferences. It would also be a nice achievement to speed up the flow computations through more efficient of parallel computations (Beraldi et al. 1997) as the empirical evaluation's computation times have shown. On a different note, one could develop a stable matching algorithm for lower quotas on projects and lecturers with or without closure as mentioned in the background section, and investigate this problem's complexity. Finally, it would be interesting to investigate how robust the stable matchings are by studying temporal preferences that change over time, similar to Gangam et al. (2022).

A Appendices

A.1 Application Screenshots

Figures A.1 – A.7 are screenshots taken when navigating the SPA problem class in the Toolkit.

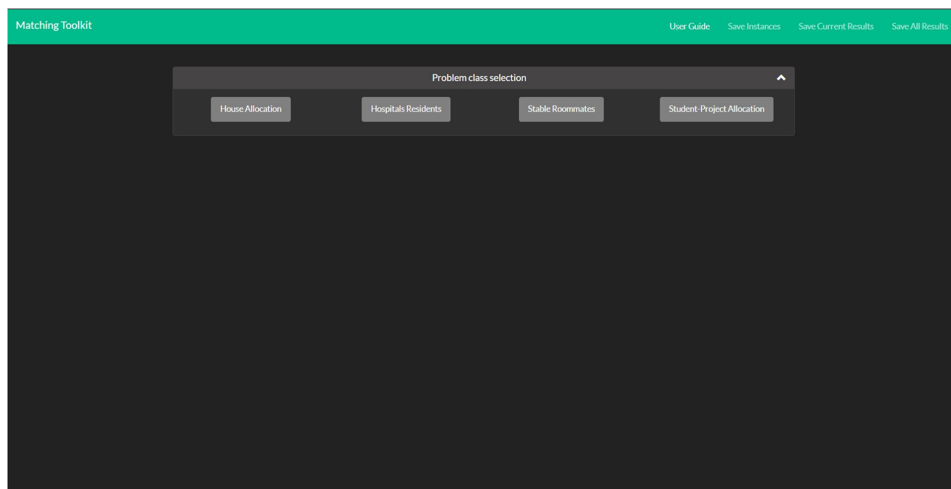


Figure A.1: Problem Class Selection

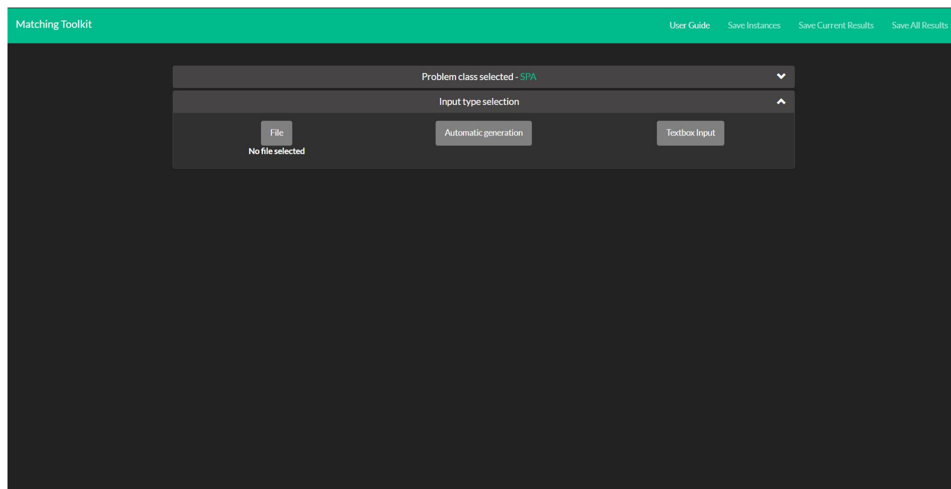


Figure A.2: Input Type Selection

Figure A.3: Parameter Selection

Figure A.4: Text-Box Input

Figure A.5: Algorithm Selection

Figure A.6: Results

Figure A.7: User Manual

A.2 Setup and Quickstart Instructions

A.2.1 Folder Structure

In the source code, the `matwa/` folder contains everything related to the Django front-end application and the `matws/` folder contains everything related to the Spring Boot back-end application.

A.2.2 Requirements and Installation

To set up the MATWA front-end,

1. Install Python 2.7 and a suitable version of pip,
2. Install Django 1.11.9,
3. To install required Python packages, run `pip install -r matwa/requirements.txt`.

To set up the MATWS back-end,

1. Install Java JDK version 1.8,
2. Install a suitable Maven version, e.g. 3.6.0,
3. Additional project dependencies will be handled automatically by Maven.

A.2.3 Setup

First, adjust the `matwsAddress` variable in `matwa/static/matchingApp.js` to the IP of your local machine with port 9115, i.e. `IP:9115`. Then set `DEBUG` to `True` in `matwa/matwa/settings.py` to activate developer mode.

A.2.4 Usage

To start the front-end developer server, in the `matwa/` folder, run `python manage.py runserver IP:8008` with `IP` replaced by the IP of your local machine. This starts the server on locally on port 8008.

To compile the back-end, in the `matws/` folder, run `mvn clean package`. To start the back-end developer server locally, in the `matws/target` folder, run `java -jar FILE` with `FILE` replaced by the name of the newly created .jar file in the same folder.

The front- and back-end should now be able to communicate and you should be able to make full use of the web application functionality in your local browser using the `IP:8008` URL, with `IP` replaced by the IP of your local machine.

A.2.5 Features and Manual

Features and instructions on how to use them can now be found on `IP:8008/matching/manual` URL, with `IP` replaced by the IP of your local machine.

A.2.6 Testing

To run the JUnit test suites, in the `matws/` folder, run `mvn test`.

A.3 Adding New Algorithms

To add a new algorithm implementation to an existing problem class, there are generally no front-end changes required.

In the back-end, place any legacy code in the folder `legacy/`, create a solver class in `impl/solver/algorithms/PROBLEMCLASS/` and add the solver class to the respective factory in the same folder and its model in `api/solver/PROBLEMCLASSFactory`. Then in the `SpringModel`, add the new algorithm to the respective `AlgTooltips` HashMap and `PROBLEMCLASSalg` array, add respective logic to the `executeAlgs` method to generate a solver object, and finally add respective logic to the `disableUnsuitableAlgs` method to disable the new algorithm for unsuitable instances.

A.4 General Contribution Guidance

Getting Access to the Project The current way to get access is to contact Professor David Manlove directly, as he manages the GitLab repository. An alternative way is to download the codebase directly from the live server if access is granted.

Contributing on Git Even if there is just one developer, software engineering best practices should be followed so that future contributors can track and verify previous changes. Therefore, when changing the codebase, a new branch should be checked out with a short name describing the changes. Commits should be created after every implemented feature or significant fix, and merge requests raised with descriptions of the changes after a coherent new feature set is implemented.

Updating the Live Server The easiest way is to use an integrated development environment (IDE) that has an SSH extension such as VS Code. Then:

1. Access the University's internal network, either through physical presence or a VPN connection.
2. Access the School of Computing Science network, either through physical presence or a VPN connection (username@sibu.dcs.gla.ac.uk).
3. Set up an SSH connection to the mithril server's virtual machine (username@mithril) on Port 8023, ideally directly in the IDE.
4. Direct to directory `/opt/boris` (e.g. using `cd /opt/boris`).
5. Check whether the services `boris-matwa` and `boris-matws` are running (e.g. using `systemctl list-units –type=service`).
6. Stop the services from running (e.g. using `sudo systemctl stop SERVICENAME`).
7. Update the relevant files. If there are permission issues, give yourself rights to the relevant files (e.g. using `sudo chown -R $USER /opt/boris/matws/` and similarly for matwa).
8. Start the services again (e.g. using `systemctl start SERVICENAME`) and check that they are actually running.
9. Ask an admin of the mithril machine to run the following commands outside of the virtual machine:
   (a) `rsync -avr matwa:/opt/boris/matwa/static ~/`
   (b) `sudo rsync -avr ~/static /opt/www/matwa/public`

A.5 Test Cases

#	Aim	Description	Expected Output	Passes
1	Check live server	Open matwa.optimalmatching.com	Problem class selection panel	Yes
2	Check HA runs successfully	Select HA class, select automatic instance generation, leave defaults or enter 1 into number fields, select all algorithms	Results panel for multiple algorithms	Yes
3	Check HR runs successfully	Select HR class, select automatic instance generation, leave defaults or enter 1 into number fields, select all algorithms	Results panel for multiple algorithms	Yes
4	Check SR runs successfully	Select SR class, select automatic instance generation, leave defaults or enter 1 into number fields, select all algorithms	Results panel for multiple algorithms	Yes
5	Check SPA runs successfully	Select SPA class, select automatic instance generation, leave defaults or enter 1 into number fields, select all algorithms	Results panel for multiple algorithms	Yes
6	Check text box input	Generate a random SPA instance, save the instance, paste the instance into the text box	Results panel for multiple algorithms	Yes
7	Check file input	Generate a random SPA instance, save the instance, upload the instance	Algorithm selection panel	Yes
8	Check instance saving	Generate a random SPA instance, save the instance, inspect whether the file	Plausible instance file	Yes
9	Check result saving	Generate a random SPA instance, select one algorithm, save the results, inspect whether the file	Results from web app	Yes
10	Verify User Manual	Open the web app and click on User Guide	User guide	Yes

Table A.1: *Manual Integration Test Cases*

A.6 Code Contributions

This project improved and extended many front- and back-end features. Therefore, changes were made all over the codebase. The GitLab stats state more than

1. 35 commits and
2. 256 (not-necessarily unique) file changes in these commits.

It is difficult to accurately state all contributions to the codebase, as some changes were introducing new functionality, some were improving or correcting old ones, and some were deleting unnecessary code altogether. A full history of changes can be found in the respective GitLab repository, `https://git.dcs.gla.ac.uk/Matchings-group/wa-toolkit`.

A.6.1 Core Front-End Changes

Here, I will mention some files to which I have done major changes to implement the new functionality mentioned above and in the dissertation:

1. `matwa/matching_app/urls.py`: added SPA form URL pattern
2. `matwa/matching_app/views.py`: added SPA form request handler
3. `matwa/static/css/...`: put correct scripts that were removed in a previous version and minor CSS adjustments
4. `matwa/static/js/AppClass.js`: SPA parameter validation and instance stats generation
5. `matwa/static/js/matchingApp.js`: major changes to integrate SPA, slider values, text-box input, input regex...
6. `matwa/static/favicon.ico`: designed and added favicon for website
7. `matwa/templates/matching_app/...Params.html`: changed default values, input types and guidance for all problem classes, newly designed and implemented parameter form for SPA
8. `matwa/templates/matching_app/index.html`: improved usability, added SPA problem class, text-box input, and input features
9. `matwa/templates/matching_app/manual.html`: improved usability, rephrased many sections, updated information

A.6.2 Core Back-End Changes

Here, I will also mention some files to which I have done changes to implement the new functionality mentioned above and in the dissertation:

WebService:

1. `matws/src/main/java/WebService/AlgorithmRunner.java`: added SPA problem class handling
2. `matws/src/main/java/WebService/FileCheck.java`: added SPA problem class handling
3. `matws/src/main/java/WebService/ParameterCheck.java`: added SPA problem class handling

Logic:

1. `matws/src/main/java/Logic/SpringModel.java`: here, the biggest changes were needed as the core functionality and program flow handling happens here. SPA handling for all API endpoints, orchestration of solvers, writers, readers, models, etc...

Models API:

1. `matws/src/main/java/api/model/Agent.java`: adjustments to make Suyi Yang's changes compatible
2. `matws/src/main/java/api/model/ModelSPA.java`: implementation of one-sided interface features
3. `matws/src/main/java/api/model/PreferenceList.java`: adjustments to make Suyi Yang's changes compatible

Solvers API:

1. `matws/src/main/java/api/solver/HRFactory.java`: adjustments to make Suyi Yang's changes compatible
2. `matws/src/main/java/api/solver/SPAAlgorithmType.java`: new enum to differentiate between one-sided algorithms
3. `matws/src/main/java/api/solver/SPAFactory.java`: made SPA solvers available in factory class

Writers API:

1. `matws/src/main/java/api/writer/statsaccumulator/StatsAccumulatorFactory.java`: added SPA problem class handling

Random Instance Generators:

1. `matws/src/main/java/generator/impl/DefaultGeneratorFactory.java`: made SPA random instance generator available in factory
2. `matws/src/main/java/generator/impl/GeneratorHRnew.java`: made file more readable to adapt for SPA
3. `matws/src/main/java/generator/impl/GeneratorSPAnew.java`: implemented SPA random instance generator for one- and two-sided instances
4. `matws/src/main/java/generator/impl/InstanceParameterSPA.java`: instance parameter specification and
builder for SPA problem class

Models:

1. `matws/src/main/java/impl/model/network/mcmf/...`: added legacy code for flow algorithms
2. `matws/src/main/java/impl/model/AgentImpl.java`: adjustments to make Suyi Yang's changes compatible
3. `matws/src/main/java/impl/model/ModelSPAImpl.java`: implementation of SPA model, mainly extended to one-sided model, adjusted checkModel and makeConsistent methods
4. `matws/src/main/java/impl/model/PrefListImpl.java`: adjustments to make Suyi Yang's changes compatible

Readers:

1. `matws/src/main/java/impl/reader/AbstractStandardReaderLogic.java`: fixed bug to allow instances with or without IDs for all problem classes - 'matws/src/main/java/impl/reader/StandardReaderHA.java': adjusted to allow instances with or without IDs
2. `matws/src/main/java/impl/reader/StandardReaderHR.java`: adjusted to allow instances with or without IDs
3. `matws/src/main/java/impl/reader/StandardReaderSPA.java`: reader logic implementation for SPA

Solvers:

1. `matws/src/main/java/impl/solver/algorithms/ha/...`: adjustments for Suyi Yang's project
2. `matws/src/main/java/impl/solver/algorithms/hr/...`: adjustments for Suyi Yang's project
3. `matws/src/main/java/impl/solver/algorithms/sm/...`: adjustments for Suyi Yang's project
4. `matws/src/main/java/impl/solver/algorithms/spa/DefaultSPAFactory.java`: made SPA algorithms available in factory
5. `matws/src/main/java/impl/solver/algorithms/spa/MinCostMaxFlowSPA.java`: made one-sided algorithms available
6. `matws/src/main/java/impl/solver/algorithms/spa/SPAImpl.java`: made student-oriented stable algorithm available
7. `matws/src/main/java/impl/solver/algorithms/spa/SPALecImpl.java`: made lecturer-oriented stable algorithm available
8. `matws/src/main/java/impl/solver/SolverImpl.java`: added SPA problem class handling

Writers:

1. `matws/src/main/java/impl/writer/instance/ ZhangInstanceWriterFactory.java`: implemented instance wrapper for lecturer-oriented stable algorithm

2. `matws/src/main/java/impl/writer/instance/ ZhangInstanceWriterSPA.java`: implemented instance wrapper for lecturer-oriented stable algorithm
3. `matws/src/main/java/impl/writer/matchedinstance/ DefaultMatchedInstanceWriterFactory.java`: added SPA problem class handling
4. `matws/src/main/java/impl/writer/matchedinstance/ MatchedInstanceHTMLWriterHA.java`: adjusted to allow instances with or without IDs
5. `matws/src/main/java/impl/writer/matchedinstance/ MatchedInstanceHTMLWriterSPA.java`: implemented matched instance writer for SPA HTML output
6. `matws/src/main/java/impl/writer/matchedinstance/ MatchedInstanceWriterSPA.java`: implemented matched instance writer for SPA output
7. `matws/src/main/java/impl/writer/matching/GUIMatchingWriterFactory.java`: added SPA problem class handling
8. `matws/src/main/java/impl/writer/matching/SPAHTMLMatchingStatsWriter.java`: implemented matching stats writer for SPA HTML output
9. `matws/src/main/java/impl/writer/matching/SPAMatchingStatsWriter.java`: implemented matching stats writer for SPA output
10. `matws/src/main/java/impl/writer/matching/SRHTMLMatchingStatsWriter.java`: adjusted to allow instances with or without IDs
11. `matws/src/main/java/impl/writer/statsaccumulator/ StatsAccumulatorFactoryImpl.java`: added SPA problem class handling

Legacy:

1. `matws/src/main/java/legacy/zhang/...` added lecturer-oriented stable algorithm legacy code for SPA

Utilities:

1. `matws/src/main/java/utilities/StabilityChecker.java` implemented SPA stability checker class
2. `matws/src/main/java/utilities/Util.java` implemented cost and profile repairs for SPA

Resources:

1. `matws/src/main/resources/instances/SPA/...` added small one- and two-sided SPA instances in text form for testing

Testing:

1. `matws/src/test/java/SPAAlgorithmsTest.java` implemented unit tests for all five algorithms
2. `matws/src/test/java/SPAEmpiricalEvalTest.java` implemented empirical test suite with random instance generation and result saving - NOTE: keep commented out by default as runtime > 1 week

A.7 SPA Stable Algorithm Pseudo-Code

The pseudocodes in Figures A.8, A.9 are of the student- and lecturer-oriented SPA stable algorithms, respectively, by Abraham et al. (2007).

```
SPA-student(I) {
    assign each student to be free;
    assign each project and lecturer to be totally unsubscribed;
    while (some student s_i is free and s_i has a non-empty list) {
        p_j = first project on s_i's list;
        l_k = lecturer who offers p_j;
        /* s_i applies to p_j */
        provisionally assign s_i to p_j;            /* and to l_k */
        if (p_j is over-subscribed) {
            s_r = worst student assigned to p_j;    /* according to L_k^j */
            break provisional assignment between s_r and p_j;
        }
        else if (l_k is over-subscribed) {
            s_r = worst student assigned to l_k;
            p_t = project assigned s_r;
            break provisional assignment between s_r and p_t;
        }
        if (p_j is full) {
            s_r = worst student assigned to p_j;    /* according to L_k^j */
            for (each successor s_t of s_r on L_k^j)
                delete (s_t, p_j);
        }
        if (l_k is full) {
            s_r = worst student assigned to l_k;
            for (each successor s_t of s_r on L_k)
                for (each project p_u ∈ P_k ∩ A_t)
                    delete (s_t, p_u);
        }
    }
    return {(s_i, p_j) ∈ S × P : s_i is provisionally assigned to p_j};
}
```

Figure A.8: *Pseudocode of Student-Oriented Stable Algorithm*

```
SPA-lecturer(I) {
    assign each student, project and lecturer to be free;
    while (some lecturer l_k is under-subscribed and
        there is some (student, project) pair (s_i, p_j) where
        s_i is not provisionally assigned to p_j and
        p_j ∈ P_k is under-subscribed and s_i ∈ L_k^j)
    {
        s_i = first such student on l_k's list;
        p_j = first such project on s_i's list;
        if (s_i is provisionally assigned to some project p)
            break the provisional assignment between s_i and p;
        /* l_k offers p_j to s_i */
        provisionally assign s_i to p_j; /* and to l_k */
        for each successor p of p_j on s_i's list
            delete (s_i, p);
    }
    return {(s_i, p_j) ∈ S × P : s_i is provisionally assigned to p_j};
}
```

Figure A.9: *Pseudocode of Lecturer-Oriented Stable Algorithm*

A.8 User Study Information Sheet

Student-Project Allocation Information Sheet to User Study

Frederik Glitzner (2478972g@student.gla.ac.uk)

Introduction

Student-Project Allocation (SPA) is an essential problem faced by schools and universities all over the world, with hundreds of participants in the Honours project allocation at the University of Glasgow alone. The model is also applicable to other contexts and can be used, for example, in wireless network engineering. Therefore, it is important to design fair, stable, and efficient algorithms that provide modular features, reasonable computation times and transparency over the allocation.

Assigning students to projects can be as simple as a random allocation. When introducing capacities on the project and supervising lecturers, the problem is referred to as SPA. As an extension, students can have ordinal preferences over the projects (one-sided preferences), and lecturers can have preferences over students (two-sided preferences).

The Matching Algorithm Toolkit (Toolkit) provides a web application allowing users to generate instances (problem settings) randomly or input them manually, and solve them using a range of state-of-the-art algorithms. Previously, the Toolkit only provided three matching problem classes. In House Allocation, people with preferences over houses are assigned to properties. In Hospitals Residents, junior doctors have preferences over hospital jobs and hospitals have preferences over their applicants. Finally, in Stable Roommates, every participant has preferences over the people they could live with. My contribution is the integration of the SPA problem class, the success and usability of which I would like to evaluate in this study.

Definitions and Optimality Criteria

SPA involves three sets of distinct agents: students, projects, and lecturers. Each project is offered by a single lecturer, and the goal is to find an assignment of students to projects (i.e. student-project pairs) so that every student is assigned to at most one project. Furthermore, projects and lecturers have capacities indicating the maximum number of students that can be assigned to it/them. Finally, students can provide a ranking list of projects, with the first entry being their most favourite project and the last project being their least favourite project. They can choose to rank as many projects as they would like to, with all non-rated projects considered unacceptable. In the two-sided setting, each lecturer provides a similar preference list and all students need to be considered acceptable. We may now consider this to be an optimisation problem for maximising the size of the matching (the number of students paired with a project) and for the participants to receive their most favourable choices.

We will consider four optimality criteria:

- **Stability**: there are no unmatched agents that could be accommodated in the matching, and no matched agents prefer each other over their current assignments.
- **Cost-optimality**: the overall sum of the ranks of assigned projects in the students' preference lists is minimised.
- **Greedy**: the number of happy students is maximized.
- **Generous**: the number of unhappy students is minimised.

Instance Format

Now consider the two-sided instance in Figure 1. There are three students, Alice, Bob, and Max, and two lecturers, each offering two projects. Every student finds two projects acceptable. The first lecturer, Dr Prince, can supervise two students, and the second lecturer, Dr Smith, can supervise one. Similarly, each project has an upper limit of students that can be assigned to it. The instance can then be converted by assigning ordered IDs (Figure 2).

Student Preferences	**Lecturer Preferences**	**Project Capacities**	
Alice: Prj 1 > Prj 2	Dr Prince: Alice>Bob>Max	Prj 1: 1,	Prj 2: 2
Bob: Prj 2 > Prj 3	Dr Smith: Bob>Alice>Max	Prj 3: 2,	Prj 4: 1
Max: Prj 1 > Prj 3	**Lecturer Capacities**	Dr Prince: 2, Dr Smith: 1	

Figure 1: Two-sided Instance with Names

Student Preferences	**Lecturer Preferences**	**Project Capacities**	
1 : 1 2	1 : 1 2 3	1 : 1	2 : 2
2 : 2 3	2 : 2 1 3	3 : 2	4 : 1
3 : 1 3	**Lecturer Capacities**	1 : 2 2 : 1	

Figure 2: Same two-sided Instance with IDs

A stable matching is then, for example, {(Alice, Project 1), (Bob, Project 2), (Max, Project 3)}, which can be represented using IDs by {(1, 1), (2, 2), (3, 3)}.

In general, the Toolkit uses IDs and the input format of a SPA-instance is shown in Figure 3 and the above instance given as an example in Figure 4.

```
NumStudents NumProjects NumLecturers
StudentId: Pref1 Pref2 (...)
StudentId: Pref1 Pref2 (...)
StudentId: Pref1 Pref2 (...)
LecturerId: LecturerCapacity: Pref1 Pref2 (...)
LecturerId: LecturerCapacity: Pref1 Pref2 (...)
ProjectId: ProjectCapacity: SuperviserId
ProjectId: ProjectCapacity: SuperviserId
ProjectId: ProjectCapacity: SuperviserId
ProjectId: ProjectCapacity: SuperviserId
```

Figure 3: SPA format in the Toolkit

```
3 4 2
1: 1 2
2: 2 3
3: 1 3
1: 2: 1 2 3
2: 1: 2 1 3
1: 1: 1
2: 2: 1
3: 2: 2
4: 1: 2
```

Figure 4: Above instance example

Now lets change the instance to be one-sided (no lecturer preferences) and having ties (students 1 and 3 find their ranked projects equally good). The instance can be seen in Figure 5.

```
3 4 2
1: (1 2)
2: 2 3
3: (1 3)
1: 2:
2: 1:
1: 1: 1
2: 2: 1
3: 2: 2
4: 1: 2
```

Figure 5: Modified instance with one-sided preferences and ties

A.9 User Study Task Sheet and Questionnaire

Matching Algorithm Toolkit - Usability Study

The aim of this experiment is to investigate the suitability of the developed extensions of the **Matching Algorithm Toolkit (Toolkit)**. We cannot tell how good web systems are unless we ask those people who are likely to be using them, which is why we need to run experiments like these. You can explore the page yourself first, then my tasksheet will guide you through some features to use. After each activity, there will be some questions for you to answer. Your answers will be stored in the Google Drive storage associated with the form. Answers will be **analysed anonymously** and the results will be used in my Level 4 Honours project dissertation, and potentially be published publicly. **Please ask any questions via email (2478972g@student.gla.ac.uk)**.

The whole study should take around **15 minutes** to complete. Please remember that it is the system, not you, that is being evaluated. *You are welcome to withdraw from the experiment at any time. If you do so, then it will not be possible for you to be debriefed about the purposes of the experiment.*

In Google anmelden, um den Fortschritt zu speichern. Weitere Informationen

* Erforderlich

Do you agree to taking part in this evaluation? *

- Yes
- No

Introduction

The **Toolkit** lets you generate, upload, and solve instances of matching problems. It can be accessed publicly here:

matwa.optimalmatching.com

Note that the **website is not mobile friendly** and works best on a regular laptop or desktop in full-screen mode.

If you do not know about matching, the SPA problem class, and instance formats, please read the **introduction** here:

https://drive.google.com/file/d/1Q6BYHyOBcSe1dFJkEOEGE8GMwjkQnyE0/view

Note that even if you decide not to read the introduction sheet now, **it may be useful to keep open for future reference** during the tasks.

What is your current knowledge with regards to matching algorithms? *

- None
- Limited
- Intermediate
- Advanced

What is your academic position? *

- Undergraduate Student
- PGT Student
- PGR Student
- Research Assistant/Associate
- Academic with Teaching Responsibilities
- Other

Textbox Input
Open the Toolkit at matwa.optimalmatching.com and take a look around.

Then select the **Student-Project Allocation** Problem class.

Choose the **Textbox Input** and paste this instance inside:

3 4 2
1: 1 2
2: 2 3
3: 1 3
1: 2: 1 2 3
2: 1: 2 1 3
1: 1: 1
2: 2: 1
3: 2: 2
4: 1: 2

Get the executable algorithms and select all **Stable** algorithms.

Run the algorithms and inspect the results.

Are you able to see the stable results? *

- Yes
- No

In the student optimal matching, what is the lecturer cost?

- 4
- 5
- 6
- 9

In the lecturer optimal matching, what is the size of the matching?

- 2
- 3
- 4
- 5

Do the student and lecturer optimal algorithms give the same matching?

- Yes
- No

Random Input

Go back to the **Input Type** selection for SPA and select **Automatic generation** this time.

Generate **ONE one-sided** instance with **30 students**, **40 projects**, and **10 lecturers**, with a total **lecturer capacity of 40** a **total project capacity of 50**, and an **even distribution** of positions.. The student preferences should be of **lengths between 4 and 6,** probability of **student ties 0.2**, and the **project skewness factor 5**.

Get the executable algorithms and select all **One-sided** algorithms.

Run the algorithms and inspect the results.

The navigation back to the previous selection tabs was intuitive. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The input fields were working well and the input types appropriate. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Are you able to see the one-sided results? *

- Yes
- No

Which agent group has a smaller cost in the cost-optimal matching?

- Students
- Lecturers

Compare the profiles of the greedy and the generous matchings. Which one has a higher number of first and second choices fulfilled for the students?

- Greedy
- Generous

Save the generated instance as a text file on your computer, using the button on the top right.

File Input

Go back to the **Input Type** selection for SPA and select **File Input** this time.

Upload the text file you just saved in the previous step.

Get the executable algorithms and select the **cost-optimal** algorithm.

Run the algorithms and inspect the result.

Save the current results, using the button on the top right, and **open** the file in browser.

The instance saving worked smoothly. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The instance uploading worked smoothly.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The saving process worked smoothly.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

The results in the downloaded file show what I expected them to show.

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Do you conduct (or have you conducted) any research related to matchings? *

- ○ Yes
- ○ No

Research Questions

Would you consider using the Toolkit for your research? *

- ○ Yes
- ○ No

What improvements or features is the Toolkit missing to become a better tool for your research?

Meine Antwort

Usability

The application was easy to navigate. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Extracting information from the results was easy. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

Overall, the tasks were easy to complete. *

	1	2	3	4	5	
Strongly Disagree	○	○	○	○	○	Strongly Agree

What did you like most about the system or your interaction with the system?

Meine Antwort

What could be improved to enhance the usability or feature set of the system?

Meine Antwort

Thank you

The main aim of the experiment was to investigate the usability of the Toolkit. I wanted to know how easy the pages are to navigate and how easy it is to extract relevant information from the results.

Please take a note of my email address (2478972g@student.gla.ac.uk), and let me know if you have any further questions about this study. Thank you for your help!

Senden Alle Eingaben löschen

Geben Sie niemals Passwörter über Google Formulare weiter.

Dieser Inhalt wurde nicht von Google erstellt und wird von Google auch nicht unterstützt. Missbrauch melden - Nutzungsbedingungen - Datenschutzerklärung

Google Formulare

A.10 User Study Ethics Checklist

School of Computing Science
University of Glasgow

Ethics checklist form for 3rd/4th/5th year, and taught MSc projects

This form is only applicable for projects that use other people ('participants') for the collection of information, typically in getting comments about a system or a system design, getting information about how a system could be used, or evaluating a working system.

If no other people have been involved in the collection of information, then you do not need to complete this form.

If your evaluation does not comply with any one or more of the points below, please contact the Chair of the School of Computing Science Ethics Committee (matthew.chalmers@glasgow.ac.uk) for advice.

If your evaluation does comply with all the points below, please sign this form and submit it with your project.

---

1. Participants were not exposed to any risks greater than those encountered in their normal working life.

   Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life. Areas of potential risk that require ethical approval include, but are not limited to, investigations that occur outside usual laboratory areas, or that require participant mobility (e.g. walking, running, use of public transport), unusual or repetitive activity or movement, that use sensory deprivation (e.g. ear plugs or blindfolds), bright or flashing lights, loud or disorienting noises, smell, taste, vibration, or force feedback

2. The experimental materials were paper-based, or comprised software running on standard hardware.

   Participants should not be exposed to any risks associated with the use of non-standard equipment: anything other than pen-and-paper, standard PCs, laptops, iPads, mobile phones and common hand-held devices is considered non-standard.

3. All participants explicitly stated that they agreed to take part, and that their data could be used in the project.

   If the results of the evaluation are likely to be used beyond the term of the project (for example, the software is to be deployed, or the data is to be published), then signed consent is necessary. A separate consent form should be signed by each participant.

   Otherwise, verbal consent is sufficient, and should be explicitly requested in the introductory script.

4. No incentives were offered to the participants.

   The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

5. No information about the evaluation or materials was intentionally withheld from the participants.
   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

6. No participant was under the age of 16.
   Parental consent is required for participants under the age of 16.

7. No participant has an impairment that may limit their understanding or communication.
   Additional consent is required for participants with impairments.

8. Neither I nor my supervisor is in a position of authority or influence over any of the participants.
   A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. All participants were informed that they could withdraw at any time.
   All participants have the right to withdraw at any time during the investigation. They should be told this in the introductory script.

10. All participants have been informed of my contact details.
    All participants must be able to contact the investigator after the investigation. They should be given the details of both student and module co-ordinator or supervisor as part of the debriefing.

11. The evaluation was discussed with all the participants at the end of the session, and all participants had the opportunity to ask questions.
    The student must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. In cases where remote participants may withdraw from the experiment early and it is not possible to debrief them, the fact that doing so will result in their not being debriefed should be mentioned in the introductory text.

12. All the data collected from the participants is stored in an anonymous form.
    All participant data (hard-copy and soft-copy) should be stored securely, and in anonymous form.

Project title ______STUDENT-PROJECT ALLOCATION IN THE MATCHING ALGORITHM TOOLKIT

Student's Name ____Frederik Glitzner_________________

Student Number ____2478972_________________________

Student's Signature ____Frederik Glitzner_________________

Supervisor's Signature _

Date ____Feb 1, 2023_____________________

A.11 User Study Responses

Timestamp	Do you agree to taking part in this evaluation?	What is your current knowledge with regards to matching algorithms?	What is your academic position?	Are you able to see the stable results?	In the student optimal matching, what is the lecturer cost?
2/11/2023 12:49:04	Yes	Limited	Undergraduate Student	Yes	5
2/16/2023 15:31:50	Yes	Intermediate	Undergraduate Student	Yes	5
2/17/2023 9:23:07	Yes	Limited	PGR Student	Yes	5
2/17/2023 11:49:30	Yes	Limited	Research Assistant/Associate	Yes	5
2/17/2023 13:05:27	Yes	Advanced	PGR Student	Yes	5
2/20/2023 11:34:49	Yes	Advanced	PGR Student	Yes	5
2/26/2023 11:04:09	Yes	None	Undergraduate Student	Yes	5
2/28/2023 13:12:33	Yes	Intermediate	Undergraduate Student	Yes	5
3/1/2023 17:04:11	Yes	Limited	Undergraduate Student	Yes	5
3/3/2023 9:40:29	Yes	None	PGT Student	Yes	5
3/3/2023 17:35:55	Yes	Limited	Undergraduate Student	Yes	5
3/6/2023 13:46:04	Yes	Intermediate	PGR Student	Yes	5
3/7/2023 21:31:55	Yes	Limited	Undergraduate Student	Yes	5
3/13/2023 17:55:02	Yes	None	Undergraduate Student	Yes	5
3/13/2023 18:31:00	Yes	None	PGT Student	Yes	

In the lecturer optimal matching, what is the size of the matching?	Do the student and lecturer optimal algorithms give the same matching?	The navigation back to the previous selection tabs was intuitive.	The input fields were working well and the input types appropriate.	Are you able to see the one-sided results?
3	Yes	3	4	Yes
3	Yes	5	5	Yes
3	Yes	2	4	Yes
3	Yes	3	4	Yes
3	Yes	4	5	Yes
3	Yes	5	4	Yes
3	Yes	5	4	Yes
3	Yes	5	5	Yes
3	Yes	5	5	Yes
3	Yes	5	5	Yes
3	Yes	5	5	Yes
3	Yes	5	5	Yes
	Yes	4	5	Yes
3	Yes	5	5	Yes
		5	5	Yes

Which agent group has a smaller cost in the cost-optimal matching?	Compare the profiles of the greedy and the generous matchings. Which one has a higher number of first and second choices fulfilled for the students?	The instance saving worked smoothly.	The instance uploading worked smoothly.	The saving process worked smoothly.	The results in the downloaded file show what I expected them to show.	Do you conduct (or have you conducted) any research related to matchings?
		5	5	5	5	No
Students	Greedy	5	5	5	5	No
Students	Generous	4	5	4	5	No
Students		5	5	4	2	Yes
	Generous	5	5	5	5	Yes
Students	Greedy	5	5	5	5	Yes
Lecturers	Generous	5	5	5	5	No
Students	Greedy	5	5	5	5	No
Students	Greedy	5	5	5	5	No
	Generous	5	5	5	5	No
Students	Greedy	5	5	5	4	No
Students	Greedy	5	5	5	5	Yes
Students	Greedy	5	5	5	5	No
Students	Greedy	5	5	5	5	No
		5				No

Would you consider using the Toolkit for your research?	What improvements or features is the Toolkit missing to become a better tool for your research?	The application was easy to navigate.	Extracting information from the results was easy.	Overall, the tasks were easy to complete.	What did you like most about the system or your interaction with the system?
		5	5	5	
		5	5	5	Multiple algorithms available
		4	5	5	The results are clearly shown, and I really like the summary information provided about the matching results
Yes	None.	5	4	4	I quite liked the fact that it was easy to navigate. And I quite liked that I could generate instances easily. This feature will be helpful for my future research.
Yes		5	5	5	
Yes		5	4	4	Ease of navigation, smoothness
		5	5	4	I like the nice UI and it was quite easy to understand what was asked.
		5	3	4	smooth animations
		5	5	4	Easy to navigate and understand
		5	4	5	The numbers highlighted in green; don't need to reload page etc. to change settings
		5	4	5	
Yes		5	5	4	The part of autogenerating values based on specific characteristics
		5	4	5	easy to return to previous steps if there is a need something to be changed. If the results need to be saved this is achieved in a pretty straightforward way as there is a button responsible for that.
		5	5	5	
		5	5	5	

What could be improved to enhance the usability or feature set of the system?
Easier comparison of matchings in results
One thing I didn't like was the previews from the sliders - it doesn't show the value I entered until I let go of the slider, which was particularly frustrating for probability of ties, since I had to estimate where 0.2 would be on the slider! Showing previews as I move the slider would be much better. Also, I found the different saving options to be a bit unclear: Until I clicked "Save Instances", I didn't know you could save the matching result as a .txt file rather than an .html file reporting the results. Another minor thing: When algorithms are hidden from me, I'd like to know why I can't use certain algorithms. For instance, why couldn't I use the stable algorithms with automatic generation? Finally, when using automatic generation, I'd like the option to save the parameters to a file then load them in again later.
I don't have any suggestions.
Please include definitions for the terms used in the toolkit.
The slider input in automatic generation could show the chosen number before letting go of the mouse, i.e., dynamically as you move the cursor right and left. Also, the number of instances is too high of a range for a slider: it was hard to land on a number I wanted.
way
Was not obvious to me what the cost (lecturer) was. I assume it is cost (total) - cost (student), but I was not sure so I left it blank. (I got cost (student) = 45 = cost (total).)

Bibliography

Abraham, D., Irving, R. and Manlove, D. (2007), 'Two algorithms for the student-project allocation problem', *Journal of Discrete Algorithms* **5**(1), 73–90.
URL: *https://www.sciencedirect.com/science/article/pii/S1570866706000207*

Anwar, A. A. and Bahaj, A. S. (2003), 'Student project allocation using integer programming', *IEEE Transactions on Education* **46**, 359–367.

Arulselvan, A., Ágnes Cseh, Groß, M., Manlove, D. and Matuschke, J. (2018), 'Matchings with lower quotas: Algorithms and complexity', *Algorithmica* **80**, 185–208.

Baidas, M. W., Bahbahani, Z. and Alsusa, E. (2019), 'User association and channel assignment in downlink multi-cell noma networks: A matching-theoretic approach', *EURASIP J. Wirel. Commun. Netw.* **2019**(1), 1–21.
URL: *https://doi.org/10.1186/s13638-019-1528-8*

Beraldi, P., Guerriero, F. and Musmanno, R. (1997), 'Efficient parallel algorithms for the minimum cost flow problem', *Journal of Optimization Theory and Applications* **95**(3), 501–530.
URL: *https://doi.org/10.1023/A:1022613603828*

Biró, P., Fleiner, T., Irving, R. and Manlove, D. (2010), 'The college admissions problem with lower and common quotas', *Theoretical Computer Science* **411**(34), 3136–3153.
URL: *https://www.sciencedirect.com/science/article/pii/S0304397510002860*

Cooper, F. (2020), Fair and large stable matchings in the stable marriage and student-project allocation problems, PhD thesis, University of Glasgow.

Cooper, F. and Manlove, D. (2018), A 3/2-approximation algorithm for the student-project allocation problem, *in* G. D'Angelo, ed., '17th International Symposium on Experimental Algorithms, SEA 2018, June 27-29, 2018, L'Aquila, Italy', Vol. 103 of *LIPIcs*, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 8:1–8:13.
URL: *https://doi.org/10.4230/LIPIcs.SEA.2018.8*

Elviwani, E., Putera Utama Siahaan, A. and Fitriana, L. (2018), Performance-based stable matching using gale-shapley algorithm, *in* 'Proceedings of the Joint Workshop KO2PI and the 1st International Conference on Advance & Scientific Innovation', ICASI'18, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, p. 59–68.
URL: *https://doi.org/10.4108/eai.23-4-2018.2277597*

Ferris, J. and Hosseini, H. (2020), 'Matchu: An interactive matching platform', *Proceedings of the AAAI Conference on Artificial Intelligence* **34**(09), 13606–13607.
URL: *https://ojs.aaai.org/index.php/AAAI/article/view/7090*

Gale, D. and Shapley, L. S. (1962), 'College admissions and the stability of marriage', *The American Mathematical Monthly* **69**(1), 9–15.
URL: *http://www.jstor.org/stable/2312726*

Gangam, R. R., Mai, T., Raju, N. and Vazirani, V. V. (2022), A Structural and Algorithmic Study of Stable Matching Lattices of "Nearby" Instances, with Applications, *in* A. Dawar and V. Guruswami, eds, '42nd IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2022)', Vol. 250 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 19:1–19:20.
URL: *https://drops.dagstuhl.de/opus/volltexte/2022/17411*

Halim, S. (2011), 'Visualgo', `https://visualgo.net/en/matching?slide=1`. Accessed: 2023-03-18.

Hopcroft, J. E. and Karp, R. M. (1973), 'An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs', *SIAM J. Comput.* **2**, 225–231.

Irving, R. W., Kavitha, T., Mehlhorn, K., Michail, D. and Paluch, K. E. (2006), 'Rank-maximal matchings', *ACM Trans. Algorithms* **2**(4), 602–610.
URL: *https://doi.org/10.1145/1198513.1198520*

Kamiyama, N. (2013), 'A note on the serial dictatorship with project closures', *Operations Research Letters* **41**, 559–561.

Király, Z. (2013), 'Linear time local approximation algorithm for maximum stable marriage', *Algorithms* **6**(3), 471–484.
URL: *https://www.mdpi.com/1999-4893/6/3/471*

Kuhn, H. W. (1955), 'The hungarian method for the assignment problem', *Naval Research Logistics (NRL)* **52**.

Kwanashie, A. (2015), Efficient algorithms for optimal matching problems under preferences, PhD thesis, University of Glasgow.

Kwanashie, A., Irving, R., Manlove, D. and Sng, C. (2015), Profile-based optimal matchings in the student/project allocation problem, *in* K. Jan, M. Miller and D. Froncek, eds, 'Combinatorial Algorithms', Springer International Publishing, Cham, pp. 213–225.

Lau, L. (2021), 'Algmatch', `https://liamlau.github.io/individual-project/`. Accessed: 2023-03-18.

Lavery, S. (2003), Algorithms for student/project allocation, Bachelor's thesis, University of Glasgow.

Lazarov, B. (2018), A web app for visualising matching algorithms, Bachelor's thesis, University of Glasgow.

Manlove, D. (2013), *Algorithmics of Matching Under Preferences*, Vol. 2 of *Series on Theoretical Computer Science*, WorldScientific.
URL: *https://doi.org/10.1142/8591*

Micali, S. and Vazirani, V. V. (1980), An o(vlvl c lel) algoithm for finding maximum matching in general graphs, *in* '21st Annual Symposium on Foundations of Computer Science (sfcs 1980)', pp. 17–27.

Monte, D. and Tumennasan, N. (2013), 'Matching with quorums', *Economics Letters* **120**(1), 14–17.
URL: *https://ideas.repec.org/a/eee/ecolet/v120y2013i1p14-17.html*

Morey, R. D. (2021), 'Student project allocation', `https://richarddmorey.github.io/studentProjectAllocation/`. Accessed: 2023-03-18.

Olaosebikan, S. (2020), The Student-Project Allocation Problem: structure and algorithms, PhD thesis, University of Glasgow.

Olaosebikan, S. and Manlove, D. (2018), 'Super-stability in the student-project allocation problem with ties', *Journal of Combinatorial Optimization* **43**, 1203–1239.

Olaosebikan, S. and Manlove, D. (2019), 'An algorithm for strong stability in the student-project allocation problem with ties'.
URL: *https://arxiv.org/abs/1911.10262*

Oozeer, Y. (2019), 'Matching algorithm visualiser', `https://matchingalgorithmvisualiser.github.io/`. Accessed: 2023-03-18.

Orlin, J. B. (2013), Max flows in o(nm) time, or better, *in* 'Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing', STOC '13, Association for Computing Machinery, New York, NY, USA, p. 765–774.
URL: *https://doi.org/10.1145/2488608.2488705*

Orlin, J., B, J. and Center, M. (1993), 'A faster strongly polynomial minimum cost flow algorithm', *Operations Research* **41**.

ProjectsGeek (2017), 'Student project allocation and management project', `https://projectsgeek.com/2017/03/student-project-allocation-project.html`. Accessed: 2023-03-18.

Remta, A. (2010), A java api for matching problems, Master's thesis, University of Glasgow.

Technical University of Munich (2016), 'Tum matching', `https://algorithms.discrete.ma.tum.de/matching/`. Accessed: 2023-03-18.

University of Glasgow (2023), 'Matching algorithm toolkit', `https://matwa.optimalmatching.com/`. Accessed: 2023-03-18.

Yang, S. (2022), Maximum cardinality popular matching vs stable matching and maximum matching, Master's thesis, University of Glasgow.

Zhang, Y. (2019), Implementation of two allocation algorithms on students and projects, Master's thesis, University of Glasgow.