Algorithms for the House Allocation Problem

Zhiyuan Lin

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8QQ

A dissertation presented in part fulfilment of the requirements of the Degree of Master of Science at The University of Glasgow

02/09/2013

Abstract

House Allocation problem is the problem of matching a set of applicants with a set of houses, where each applicant has a preference list, which is a non-empty subset of the set of all houses organised in order of preference and may have ties.

The objective of this project is to implement algorithms that calculate matchings according to different optimality criteria for House Allocation problem and its extensions, and perform empirical evaluations over the outcomes of the algorithms.

The outcomes of the project included the implementations of three algorithms: maximum cardinality Pareto optimal matching algorithm for CHA, maximum popular matching algorithm for CHAT, and rank-maximal matching algorithm for CHAT. Empirical evaluations were also conducted over the outputs of these algorithms.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic format. **Please note that you are under no obligation to sign this declaration, but doing so would help future students.**

Name: _____ Signature: _____

Acknowledgements

I would like to thank my parents for their support during the project.

I would like to thank my friends, especially Finlay McCourt, for all kinds of help they offered during the project.

I would also like to thank Jade Ye for her inspiration.

Most importantly, I would like to express my sincere gratitude to my supervisor Dr. David F. Manlove for his dedicated help and guidance.

Contents

Chapter 1

Introduction

In real life, it is often necessary to allocate a set of limited resources among a finite group of individuals. Allocating campus housing to students, for example, is a problem that every university has to face.

Resources differ from each other in many ways. In the case of campus housing, some flats maybe better located or have larger rooms than others. Individuals normally have different preferences over the resources. Some students might prefer a cheaper but smaller room, while others might be willing to pay more for a comfortable and well-furnished flat. The allocation of resources certain have a direct impact on the welfare of those individuals involved. Allocation results differ from one another. What is the best way to allocate the resources? It depends the criteria for optimality. For example, it is often desirable to maximise the number of students that get allocated to campus housing. Giving the largest number of students possible their favourite housing can also help improve student satisfaction. However, as different allocation results inevitably favour different students, it is nearly impossible to produce a result that gives every one their favourite housing in practice. That's why it is important to determine a set of reasonable and achievable criteria for optimality. Formally we call the individuals involved in such a problem *applicants*, and each allocation result a *matching*. Possible criteria for optimality include *Pareto optimality*, *rank-maximality*, and *popularity*. Pareto optimality is a criteria considered as essential by many economists. An allocation (matching) is Pareto optimal if there is no other matching in which some applicants are better off and no applicant is worse off. A matching is rank-maximal if the maximum number of applicants obtain their first choices, and subject to this, the maximum number obtain their second choices,

et cetera. A matching is popular if there is no other matching that is preferred by a majority of applicants. More criteria can be identified.

This type of problem is often called a *House Allocation problem* and it is actually a typical problem model in the discipline of matching problems, which involves economics, algorithms and graph theory. Economists use matching problems to explore the formation of relationships, including the relationships between individuals and resources, the human relationships such as marriage. Computer scientists and mathematicians develop efficient algorithms to obtain matchings in a given problem instance based on different optimal criteria. Graph theory provides certain theoretical foundation for matching in the context of a graph, which consists of vertices (nodes) and edges connecting some of the vertices. For example, a typical problem in graph theory is maximising the number of vertices matched.

Matching problems are widely studied. Professor Lloyd Shapley was awarded 2012 Nobel Memorial Prize in Economics Sciences for his contribution to *stable marriage problem*. An example of stable marriage problem problem is the assignment of graduating medical students to their first hospital appointment.

House Allocation problem, as a special type of matching problem, has a wide range of application. Apart from the campus housing example mentioned before, other applications of House Allocation problem includes the assignment of students to projects (with a finite set of projects for selection), the allocation of employee to job posts.

The purpose of this project is to implement algorithms that deal with House Allocation problem and some of its extensions according to different optimal criteria, and conduct a empirical study over the results produced by these algorithms.

In Chapter 2, the background information about the House Allocation problem and the relevant algorithms, including fundamental definitions and theories, and the relevant works are introduced. Chapter 3 discusses the design issues in the project including the detail of the algorithms and relevant data structures. Chapter 4 then discusses the implementation level details. Details of correctness testing and empirical studies over the algorithms are shown in Chapter 5.

Chapter 2

Background

2.1 The House Allocation problem (HA) and its extensions

The House Allocation problem (HA) was first introduced by Hylland and Zeckhauser in 1979 [5]. It is often referred to as a bipartite matching problem with one-sided preferences. Examples of such problems include assigning students to projects, matching students to campus housing and assigning individuals to jobs.

In a standard HA instance I there is a set of *applicants* (denoted by $A = \{a_1, a_2, \ldots, a_{n_1}\}$, where n_1 is the number of applicants) and a set of indivisible objects (known as *houses*, denoted by $H = \{h_1, h_2, \ldots, h_{n_2}\}$, where n_2 is the number of houses). Let $n = n_1 + n_2$. An applicant a_i may find several houses in H (say h_1, h_2 and h_3) acceptable.

Each applicant ranks his acceptable houses in strict order of preferences. Each pair (a_i, h_j) where a_i finds h_j acceptable are assigned a *rank* i, which is the number of houses that a_i prefers to h_j plus 1. The list of all the houses that a_i finds acceptable in rank order is called the *preference list* of applicant a_i, denoted by A_i. In a basic HA instance, a preference list is strictly ordered. This means for any h_j, h_k in a_i's preference list, if $j < k$, a_i prefers h_j to h_k and a_i does not find any two houses equally favourable. On the other hand, houses do not have preference lists. This means that all applicants that find a house h_i acceptable are the same for h_i.

Define E as the set of all pairs (a_i, h_j) where a_i finds h_j acceptable. Let $m = |E|$.

$a_1 : h_1, \boldsymbol{h_2}, h_3$
$a_2 : h_2, \boldsymbol{h_3}$
$a_3 : \boldsymbol{h_1}$

Figure 2.1: Example of House Allocation Problem Instance

Consider each individual (either a house or an applicant) in I as a vertex, and every pair in E an edge, then we have the *bipartite graph* $G = (A \cup H, E)$ of I. Also we have $E = E_1 \cup E_2 \cup \cdots \cup E_r$, where $E_i (i = 1, 2, \ldots, r)$ is the set of edges with rank i.

In terms of a bipartite graph, a matching M in an HA instance I contains a subset of edges in E and no two edges in M share an vertex. If two vertices (a_i, h_j) are connected by an edge, we say they are adjacent to each other or they are neighbours to each other. If an applicant a_i and a house h_j is *matched to* (or *assigned to*) to each other, then $(a_i, h_j) \in M$. Also if an applicant a_i is matched in M, then let $M(a_i)$ denote the house that a_i is matched to. Vice versa, $M(h_j)$ denotes the applicant that h_j is matched to. Define the *regret* of a matching M to be the maximum rank of the matched pairs in M.

The *size* of a matching $|M|$ is the number of applicants matched in the matching, while the *cost* of a matching is the sum of the ranks of every pair $(a_i, h_j) \in M$.

Let r denote the largest number that any applicant uses to rank a house. The *profile* of a matching M is a vector $(x_1, x_2, \ldots, x_r)$ where x_i is the number applicants that are engaged to their ith choices.

A House Allocation problem instance is given in Figure 2.1. After each applicant is the applicant's preference list, with assignment in bold. The underlying bipartite graph of this instance is shown in Figure 2.2, with the matched pairs connected with bold lines.

2.1.1 The extensions of HA

An extension of the House Allocation problem is House Allocation with Ties (HAT). The difference between an HA instance and an HAT instance is that in an HAT instance an

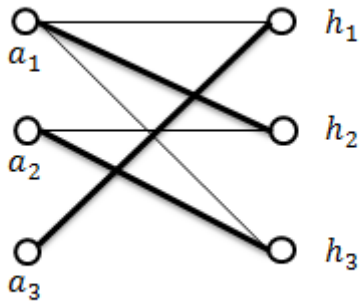

Figure 2.2: Bipartite Graph of the Instance shown in Figure 2.1

$$a_1 : h_1, (h_2, h_3)$$
$$a_2 : h_2, h_3$$
$$a_3 : h_1$$

Figure 2.3: Example of House Allocation Problem Instance with Ties

applicant can be indifferent with two or more houses $h'_1, h'_2, \ldots, h'_k$. The set of houses $h'_1, h'_2, \ldots, h'_k$ are said to be tied in the preference list. Equivalently we can say the preference lists contain ties. An example of preference list with ties is given in Figure 2.3. We can see that h_2 and h_3 on a_1's preference list were tied to each other. Another extension of HA is the *Capacitated House Allocation problem*(CHA) where a house h_j can be assigned to several applicants at the same time. The maximum number of applicants that a house h_j can be assigned to at the same time is called its *capacity*, denoted by c_i. Let C be the sum of the capacities of the houses. CHA and HAT could be combined into Capacitated House Allocation problem with Ties (CHAT). Alternatively, HA can be considered as a special case of CHAT where there is not ties and all houses have capacity 1.

A matching M in a CHAT instance I contains a subset of edges in E and no two edges in M share an applicant vertex (edges might, however, share a house vertex and the number of edges connected to a house h_j should be less than or equal to its capacity).

$$a_1 : h_1, (h_2, h_3), h_4$$
$$1 \quad 2 \quad 2 \quad 4$$

Figure 2.4: Example of Ties and Ranks

$a_1 : h_1, \boldsymbol{h_2}$
$a_2 : \boldsymbol{h_2}$
$a_3 : \boldsymbol{h_1}$

$c_1 = 1$
$c_2 = 2$

Figure 2.5: Example of a CHAT Problem Instance

The definition of $M(a_i)$ remains the same. However the $M(h_j)$ in I now denotes the set of all the applicants matched to h_j in M. For a CHAT instance, the definitions of rank and regret remain the same. The only difference is that for an applicant a_i now there can be more than one houses with rank i. Figure 2.4 shows an applicant's preference list with ties and the ranks of houses for the applicant.

In terms of graphs, a *capacitated bipartite graph* G could be derived from a CHAT instance I. Each applicant vertex in G has capacity 1 while each house vertex has capacity that could be greater than 1. A house vertex h_j is defined as *full* if the number of applicant vertices h_j is matched to is equal to its capacity c_j. Figure 2.5 shows a CHAT problem instance while Figure 2.6 shows the underlying capacitated bipartite graph. Note that both a_1 and a_2 in this instance are matched to h_2.

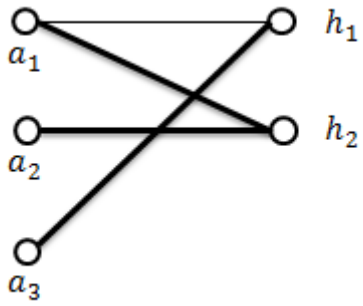

Figure 2.6: Capacitated Bipartite Graph of the Instance shown in Figure 2.5

2.2 Maximum Matching Algorithm

It is very often desirable to match as many applicants as possible. Here we define a *maximum* matching as a matching that contains the largest possible number of edges. Such matching is also called a maximum-cardinality matching. Note that a maximum matching is different from a *maximal* matching, in which there is no pair (a_i, h_j) such that a_i and h_j are both undersubscribed and a finds h acceptable. A maximum matching is always maximal, but a maximal matching might not be maximum. We start by discussing maximum matching algorithm in HA/HAT setting (with houses of capacity 1).

An algorithm that calculates a maximum matching M from a bipartite graph G is the Hopcroft-Karp algorithm [4], which yields a result in $O(\sqrt{n}m)$ time, where n is the number of vertices and m is the number of edges. The *augmenting path algorithm*, which is a simplified but less efficient version of Hopcroft-Karp algorithm, also produces a maximum matching by repeatedly finding and eliminating augmenting paths.

Given a bipartite graph G and a matching M, an *alternating path* is made up of edges in M and edges not in M alternately. An *augmenting path* then is defined to be an alternating path which starts and ends with *exposed* (namely unmatched) vertices. The path $\{a_1, h_2, a_2, h_3\}$ in Figure 2.7 is an augmenting path because it starts and ends with exposed vertices a_1 and h_3.

Theorem 2.2.1 *M is of maximum cardinality if and only if M admits no augmenting path.[11]*

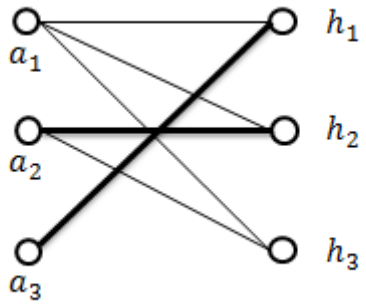

Figure 2.7: Bipartite Graph with an Augmenting Path

An augmenting path could be eliminated by matching all the vertices at even position on the path to their predecessor. Take the augmenting path $\{a_1, h_2, a_2, h_3\}$ in Figure 2.7 for example again. We only need to match h_3 with a_2 and h_2 with a_1. This process is called *augmentation*. To augment an augmenting path, one simply match every vertex at position n to the vertex at position $n+1$ where n is an integer. The result of augmentation in Figure 2.7 is shown in Figure 2.2.

As shown in Algorithm 1 an Augmenting Path algorithm in principle simply keeps searching for augmenting paths and once found augment along the path. When no augmenting paths could be found, the algorithm terminates and a maximum matching of the bipartite graph is found.

Algorithm 1 Augmenting Path Algorithm

```
1: for ( ; ; ) do
2:     Search for augmenting path P;
3:     if an augmenting path P is found then
4:         Augment M along P ;
5:     else
6:         break;
7:     end if
8: end for
```

Augmenting path algorithm could be adapted to calculate a maximum matching from a capacitated bipartite graph. The principle of the algorithm remains the same for a CHAT instance, only a house vertex might be matched to multiple applicant vertices. There are two ways to this. The first way is to convert the capacitated bipartite graph G to a non-

Algorithm 2 Search for Augmenting Path in the bipartite graph of an HA instance

```
1: Reset all vertices to unvisited;
2: Unlabel any start vertices;
3: Create a queue q
4: while there exists an exposed and unvisited applicant vertex u do
5:    Add u to the end of q
6:    Label u as the start vertex
7:    while  q is not empty do
8:       Remove head from q, store as vertex v
9:       Label v as visited
10:      for each vertex w on v's adjacency list do
11:         if w is not visited then
12:            Label w as visited
13:            Set v as w's predecessor
14:            if w is exposed then
15:               return  w as the end of an augmenting path
16:            else
17:               Add w's mate to q
18:            end if
19:         end if
20:      end for
21:   end while
22: end while
23: return  "no augmenting path could be found";
```

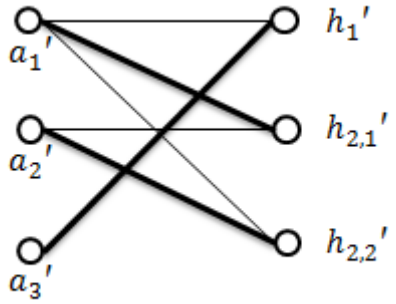

Figure 2.8: Cloned Graph of Figure 2.6

capacitated bipartite graph $C(G)$ through cloned vertices. Let h_j be a house vertex of capacity c_j, there should be c_j clone house vertices $\{h'_{j,1}, h'_{j,2}, \ldots, h'_{j,c_j}\}$ in $C(G)$, each of which has capacity 1 and is other-wisely the same to h_j (mainly in that they are connected to the same set of applicants). The Augmenting Path algorithm for HA could then be used to the clone graph $C(G)$ without any changes to calculate a maximum matching $C(M)$ for $C(G)$. The capacitated bipartite graph G can then inherit its matching from $C(G)$ by matching each pair (a_i, h_j) in G where $(a_i, h_{j,k})$ are matched in $C(G)$. Figure 2.8 shows the cloned graph of Figure 2.6 and the matching is obviously maximum.

Another way of calculating a maximum matching for a CHAT instance requires only one small change to the algorithm shown in Algorithm 2. In line 17 of Algorithm 2, if the house has capacity greater than 1, vertex w may have multiple mates, and we need to add all w's unvisited mate to q.

2.3 Maximum cadinality Pareto Optimal matching

The term *Pareto optimality*, named after Italian Economist Vilfredo Pareto, is originally used in the field of Economics to describe a state of allocation of resources where no individual can be made better off without making at least one other individual worse off. Consider a classic scenario in which a set of limited resources are allocated to a finite number of individuals. A change to the allocation that makes one or more individuals better off without making other individuals worse off is called a *Pareto improvement*. A Pareto optimal allocation is one to which no Pareto improvements could be made.

A lot of effort has been made to devise an algorithm that calculates a Pareto optimal matching from a given HA instance. We start by introducing Pareto Optimal Algorithm for House Allocation problem. The Algorithm will then be extended to Capacitated House Allocation problem.

2.3.1 Simple and Random Serial Dictatorship

For an HA instance I, one of the simplest mechanisms that calculate a Pareto optimal matching in I is the *simple serial dictatorship*. The simple serial dictatorship mechanism was described in [1].

Simple serial dictatorship [1]: Let all applicants be in a pre-decided order, assign the applicant that is ordered first to his/her first choice house; the applicant ordered second is then assigned to his/her top choice among the remaining unassigned houses; continue the process until all the applicants are assigned or there is no house left.

Although the simple serial dictatorship mechanism yields a Pareto optimal matching, it is not always favourable because it discriminates between applicants. This problem could be solved by choosing an ordering randomly before using simple serial dictatorship. Such a mechanism is called *random serial dictatorship*.

Random serial dictatorship [1]: Randomly produce an ordering of applicants with uniform distribution; then use the simple serial dictatorship according to the ordering.

2.3.2 Core from Random Endowments

An alternative mechanism called *core from random endowments* was introduced in [15]. To illustrate this mechanism, it is necessary to introduce the *core* of a market, the *Housing Market* problem, and Gale's *Top Trading Cycles* (TTC) mechanism.

The core of a market, in game theory, is the set of allocations that cannot be profitably improved by a subset of the market's participants. That means the allocations (matchings) in the core is Pareto optimal. The core can be empty, meaning that no such allocations exist in the market. The Housing Market problem (HM) is different from House Allocation problem only in that in an HM instance every applicant owns a house at the very beginning (so the number of houses and the number of applicants are always

$a_1 : h_1, \boldsymbol{h_2}$
$a_2 : h_2, \boldsymbol{h_3}$
$a_3 : h_3, \boldsymbol{h_1}$

Figure 2.9: A House Allocation Problem Instance

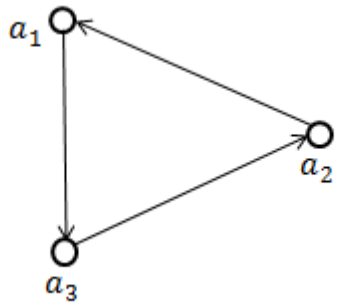

Figure 2.10: An example of Envy Graph

equal). This is called an *initial endowment*. Shapley and Scarf [15] proved that there always exists a Pareto optimal matching in a house market instance and could be found using Gale's Top Trading Cycles mechanism. The mechanism is named after David Gale. Consider the instance shown in Figure 2.9; applicant a_1 prefers a_3's house; a_3 prefers a_2's house; and a_2 prefers a_1's house. This forms the *cycle* shown in Figure 2.10. The graph is also called an envy graph.

Roth and Postlewaite [14] later proved that the result of Gale's TTC mechanism is the unique matching in the core of an HM instance. Roth [13] proved that it is a dominant strategy in Gale's TTC mechanism for each player to show his/her true preferences, namely the mechanism is strategy-proof.

As an HM instance can be regarded as an HA instance with an initial matching (endowments), it is possible to convert a House Allocation problem instance to an HM instance and find the core of the induced HM instance as a solution to the original HA problem. This mechanism is named the *core from random endowments*.

Algorithm 3 Gale's Top Trading Cycles for HM

1: **while** (there are still agents in the market) **do**
2: Let each applicant has a pointer pointing to the owner of his/her favourite house. {A cycle is defined as an ordered list of applicants $a_1, a_2, \ldots, a_i$ where a_1 points to a_2, a_2 to a_3, ..., and a_i to a_1. Since there are finite number of agents, there is at least one cycle.}
3: Find a cycle C
4: Assign each applicant in C to the house of the applicant he points to.
5: Remove all the agents in C and his/her houses from the market
6: **end while**

Core from random endowments [15]: Given an HA instance I, randomly select a matching for I and calculate the core of the induced housing market using Gale's TTC mechanism.

2.3.3 Maximum cardinality Pareto optimal matching for HA

The core from random endowments mechanism does not necessarily produce a Pareto optimal matching with maximum size.

Formally speaking, a matching M is maximal if there is no pair (a_i, h_j) such that a_i and h_j are both unmatched in M but a_i finds h_j acceptable. A matching M is said to be *trade-in-free* if there is no pair (a_i, h_j) such that h_j is unmatched and a_i likes h_j better than his/her current house. A matching M is defined to be *coalition-free* if M admits no coalition, which is basically a cycle defined in Section 2.3.2.

[2] presented an algorithm for find a maximum cardinality Pareto optimal matching on the basis of Gale's TTC method. A major result of [2] is:

Theorem 2.3.1 *A matching M in an HA instance is Pareto optimal if and only if M is* maximal, trade-in-free *and* coalition-free.

The *maximum cardinality Pareto optimal matching algorithm for HA* is based on Theorem 2.3.1 and consists of three phases, mirroring the three conditions in the theorem. Phase 1 of the algorithm calculates a maximum matching M_1 of the given HA instance. As was discussed in Section 2.2 the Augmenting path Algorithm can be used to achieve this goal. Phase 2 involves repeatedly identifying applicants that prefer an unmatched house

to their existing assignment and promoting them to their preferred houses. The result of this phase is a trade-in-free matching M_2. In Phase 3, M_2 is converted to a coalition-free matching using Gale's TTC mechanism. Details of this algorithm is discussed in Section 3.

2.3.4 Maximum cardinality Pareto optimal matching for CHA

The maximum cardinality Pareto optimal matching algorithm described in Section 2.3.3 can be extended to the cases where a house might be matched to more than one applicant, namely the Capacitated House Allocation characterised in Section 2.1.1.

Sng [16] discussed the conditions for a matching to be Pareto optimal in an instance of CHA. The conditions for a matching to be Pareto optimal in CHA are similar to those in HA.

However, it is important to understand that in a CHA instance the interpretations of *maximal* and *trade-in-free* are different from those in an HA instance. In a CHA instance I, A matching M is defined to be *maximal* if there is no pair (a_i, h_j) such that a_i is unmatched and h_j is not full (undersubscribed), and a_i finds h_j acceptable. A matching M is defined to be *trade-in-free* if there is no pair (a_i, h_j) such that h_j is not full and a_i likes h_j better than his/her current house. The definition of *coalition-free* remains the same as that of HA.

Theorem 2.3.2 *[7, 16] A matching M in an CHA instance is Pareto optimal if and only if M is* maximal*,* trade-in-free *and* coalition-free*.*

Sng [16] also provided a three-phase algorithm for finding a maximum cardinality Pareto optimal matching in a CHA instance. The three phases in *maximum cardinality Pareto optimal matching algorithm for CHA* is in principle generalisation of the three phases in maximum Pareto optimal matching algorithm for HA. Phase 1 here again involves calculating a maximum matching in the given instance. As was discussed in Section 2.2, the augmenting path algorithm can be extended to find a maximum matching in a capacitated bipartite graph. Phase 2 again is to convert the result of Phase 1 to a trade-in-free maximum matching. The way to achieve this is to repeatedly looking for applicants that prefer an undersubscribed house to their existing assignment and promoting them to the houses that they preferred. Phase 3 then makes the matching coalition-free.

2.4 Popular matching

2.4.1 Popular matching for HA

The concept of popular matching was defined in [3]. In order to formally define popular matching, it is necessary to establish applicants' preferences over matchings. An applicant a_i is said to *prefer* matching M to another matching M' if:

1. a_i is matched in M, but unmatched in M'; or
2. a_i is matched in both matchings and prefers h_j to h'_j, where $(a_i, h_j) \in M$ and $(a_i, h'_j) \in M'$.

A matching M is defined to be *popular* if there is no matching M' such that the number of applicants preferring M' to M exceeds the number of applicants preferring M to M'. A popular matching might not necessarily exist for a given House Allocation instance.

The maximum cardinality popular matching algorithm for HA, which determines whether a given HA instance admits a popular matching and finds a largest such matching if one exists, was given in [3]. The complexity of the algorithm was $O(n + m)$, where n is the number of vertices and m is the number of edges. Abraham et al. [3] also discussed the popular matching in the context of HAT and devised an $O(\sqrt{n}m)$ algorithm for finding a maximum cardinality popular matching in an HAT instance if one exists.

To calculate a popular matching, it is also necessary to introduce the concept of a *last resort* houses. For each applicant a_i, let there be a unique last resort house $l(a_i)$, which is appended to the end of a_i's preference list. A matching is defined as *applicant-complete* if every applicant is matched. The size of a matching with last resort is the number of applicants not matched to their last resorts. For an applicant a_i in an HA instance, define $f(a_i)$ to be the first-ranked house on a's preference list. Any such house is called a *f-house*. Also let $s(a_i)$ be the first house that is not a f-house on a_i's preference list. Also $f(a)$ denotes the union of $f(a_i)$ for all applicants. The set $s(a)$ denotes the union of $s(a_i)$ for all applicants. Abraham et al. proved that:

Lemma 2.4.1 *[3] A matching M in an HA instance is a popular matching if and only if:*

1. *every f-house is matched in M*

2. *every applicant a_i is assigned to $f(a_i)$ or $s(a_i)$*

Given a bipartite graph G, its *reduced graph* G' is defined to be a subgraph of G that contains only edges connecting each applicant a_i to $f(a_i)$ and $s(a_i)$. $G' = (A \cup H, E')$, where $E' = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i) \cup s(a_i)\}$. Lemma 2.4.1 can then be converted to:

Theorem 2.4.1 *[3] A matching M in an HA instance is a popular matching if and only if:*

1. *every f-house is matched in M*

2. *M is applicant-complete in G'*

Based on 2.4.1, Abrahma et al. [3] devised a linear-time popular matching algorithm for HA. The algorithm was essentially a simplified version of Algorithm 7.

2.4.2 Popular matching for HAT

For instances with ties (HAT), the approach to finding popular matchings (or determining whether one exists) is in principle the same with the approach for HA. However, the definition of certain concepts need to be adjusted. For an applicant a_i, $f(a_i)$ now denotes the set of first ranked houses for a_i. It is apparent now there might be more than one houses in the set. Also the *first-choice graph* of G is defined to be the sub-graph of G that contains only first-ranked edges E_1, denoted by G_1. After defining $f(a_i)$, it is again necessary to define $s(a_i)$ which makes sure an applicant a_i is matched to either $f(a_i)$ or $s(a_i)$ in a popular matching.

As defined in Section 2.2, given a maximum matching M in G, an alternating path is made up of edges in M and edges not in M alternately. A vertex v is said to be *even* if there exists an even length alternating path (with respect to M) linking from an exposed vertex to v. Likewise, a vertex v is defined to be *odd* if there is an odd length alternating path from an exposed vertex to v. And a node is *unreachable* if there is no alternating path from an exposed vertex to it. let $\mathcal{E}, \mathcal{O}$ and $\mathcal{U}$ denote the set of even, odd

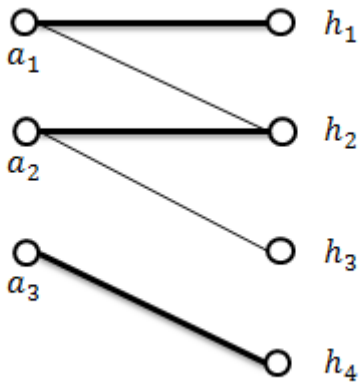

Figure 2.11: A bipartite graph and a maximum matching [6]

and unreachable nodes respectively. Also note that the given the same bipartite graph G, the composition of $\mathcal{E}, \mathcal{O}$ and $\mathcal{U}$ remains the same in spite of which maximum matching is used.

Consider Figure 2.11, the matching $M = \{(a_1, h_1), (a_2, h_2), (a_3, h_4)\}$ in the bipartite graph G was obviously maximum. With M and G it is easy to obtain the $\mathcal{EOU}$ labelling: $\mathcal{E} = \{h_1, h_2, h_3\}$, $\mathcal{O} = \{a_1, a_2\}$, and $\mathcal{U} = \{a_3, h_4\}$.

We have the following lemma:

Lemma 2.4.2 *[11] Given a graph G and a maximum matching M in G, let $\mathcal{E}, \mathcal{O}$ and $\mathcal{U}$ be the sets defined by G and M. we have:*

1. *$\mathcal{E}, \mathcal{O}$ and $\mathcal{U}$ are pairwise disjoint.*
2. *In any maximum matching in G, every vertex in $\mathcal{O}$ is matched with a vertex in $\mathcal{E}$, and every vertex in $\mathcal{U}$ is matched with another vertex in $\mathcal{U}$. The size of a maximum matching is $|\mathcal{O}| + |\mathcal{U}|/2$.*
3. *No maximum matching of G contains $\mathcal{OO}$ edges, $\mathcal{OU}$ edge or $\mathcal{EU}$ edges.*

In [3] it was proved that given a popular matching M, $M \cap E_1$ is a maximum matching in G_1.

Given a bipartite graph G, let M_1 be a maximum matching in the first-choice graph G_1. According to the second conclusion in Lemma 2.4.2, every odd or unreachable house must be matched in M_1. This means they must be matched to their first choice houses. These houses cannot be members of $s(a_i)$ for any a_i. So it is defined that $s(a_i)$ is the top-ranked even houses for a_i in G_1. It should also be noted that $s(a_i) \neq \emptyset$ because last resort houses are always even. Again we have $G' = (A \cup H, E')$, where $E' = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i) \cup s(a_i)\}$.

After adjusting the definitions, we can reach a conclusion similar to Theorem 2.4.1:

Theorem 2.4.2 *[3] A matching M in an HAT instance is a popular matching if and only if:*

1. *every f-house is matched in M*
2. *M is applicant-complete in sub-graph G'*

2.4.3 Popular matching for CHA and CHAT

Manlove & Sng [8] considered popular matchings in instances of CHA and CHAT. They described a $O(\sqrt{C}n_1 + m)$ algorithm in CHA instances and a $O((\sqrt{C} + n_1)m)$ algorithm

in CHAT instances for finding a maximum cardinality popular matching if one exists (C is the sum of the capacities of the houses; n_1 is the number of applicants; m is the number of edges).

Let f_j be the number of first-ranked edges that is linking to h_j. For an instance of CHA, $f(a_i)$ again denotes the first-ranked houses for applicant a_i. However $s(a_i)$ now denotes the most-preferred house h_j for a_i that either:

1. h_j is not a f-house; or
2. h_j is a f-house but $f(a_i) \neq h_j$ and $f_j < c_j$.

Note that $f(h_j)$ now denote the set of applicants that consider h_j as their f-house., while $M(h_j)$ denotes the set of applicants that are matched to h_j. Apparently there can be more than 1 of such applicants. Under these definitions, Manlove & Sng [8] proved that an applicant a_i must be matched to either $f(a_i)$ or $s(a_i)$. So we have:

Theorem 2.4.3 *[8] A matching M in a CHA instance is popular if and only if:*

1. *for every f-house h_j, if $f_j \leq c_j$, h_j must be matched to all applicants in $f(h_j)$, else h_j must be matched with applicants in $f(h_j)$ to its capacity.*
2. *M is an applicant-complete matching in the reduced subgraph G'.*

For a CHAT instance, the definitions of $f(a_i)$ and $s(a_i)$ are similar to those in an HAT instance. However the process of defining $s(a_i)$ was different, because the Gallai-Edmonds decomposition needed to be extended to capacitated bipartite graph. This can be done using *cloned graph*.

As introduced in Section 2.2, given a graph G, the clone graph $C(G)$ of G replaces each house h_j of capacity c_j in G with cloned houses of capacity 1, each of which is adjacent to the same set of applicants as h_j. Let M be a maximum matching in G, we can also extend M to $C(G)$ by allocating each applicant in $M(h_j)$ to one of h_j's unmatched clones. The new matching is denoted by $C(M)$ and is a maximum matching in $C(G)$. It was then proved that a $\mathcal{EOU}$ labelling in G can be inherited from $C(G)$, because any two clones of a house have the same $\mathcal{EOU}$ label. And the labelling in $C(G)$ can be done with the process described in Section 2.4.2.

Note that the properties of Gallai-Edmonds decomposition as presented in Section 2.4.2 also needs to be adjusted slightly in the context of capacitated bipartite graph. For a maximum matching M in G, every odd house now needs to be fully matched and only to even applicants; similarly every unreachable house needs to be fully matched and only to unreachable applicants; and the size of the matching $|M| = |\mathcal{O}_A| + |\mathcal{U}_A| + \sum_{h_j \in \mathcal{O}_H} c_j$, where $\mathcal{O}_A$ denotes the set of odd applicant vertices, $\mathcal{U}_A$ denotes the set of unreachable applicant vertices and $\mathcal{O}_H$ denotes the set of odd house vertices.

With $f(a_i)$ and $s(a_i)$ redefined, again we have $G' = (A \cup H, E')$, where $E' = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i) \cup s(a_i)\}$.

Theorem 2.4.4 *[8] A matching M in an CHAT instance is a popular matching if and only if:*

1. *every f-house is matched in M*
2. *M is applicant-complete in G'*

Observe the similarity between Theorem 2.4.1, Theorem 2.4.2 and Theorem 2.4.4. The general principle of finding a popular matching is the same. Only the definition of $f(a_i)$, $s(a_i)$ and thus the composition of the reduced graph G' were generalised as the problem was generalised. Algorithm 7 was devised on the basis of Theorem 2.4.4.

2.5 Rank-maximal matching

Another possible optimality criterion is the size the profile, which was already defined in Section 2.1. Matchings under optimal criteria that involve the profile are called profile-based optimal matchings. *Rank-maximal matching* is a type of profile-based optimal matching. We discuss the algorithms for finding a rank-maximal matching here.

2.5.1 Rank-maximal matching for HAT and CHAT

The rank-maximal matchings was first introduced and discussed in [6]. Irving et al. also gave a $O(min(z\sqrt{n}, n + z)m)$ time algorithm for finding a rank-maximal matching in a

given HAT instance, where z is the maximal rank of an edge in an optimal solution, n is the number of vertices, and m is the number of edges. This algorithm naturally works for HA instances as well.

Let $\succ$ denote the lexicographic order on profiles. Given two profiles $(x_1, x_2, \ldots, x_r)$ and $(y_1, y_2, \ldots, y_r)$ from two matchings in the same HA instance, $(x_1, x_2, \ldots, x_r) \succ (y_1, y_2, \ldots, y_r)$ if there exists some k such that $x_k = y_k$ for $1 \leq i < k$ and $x_k < y_k$.

If there does not exist a matching M' for M such that $M' > M$ in a given instance, M is considered to be a rank-maximal matching.

Let $G_i = (A \cup H\ ,\ E_1 \cup E_2 \cup \ldots E_i)$. That is to say G_i contains all edges whose ranks are less than or equal to i. Then we have $G_1 = (A \cup H\ ,\ E_1)$. Irving et al. [6] observed that the rank-maximal matching problem can be converted to a maximum matching problem by deleting all edges that do not belong to a rank-maximal matching from the graph. Let the first ranked reduced graph $G'_1 = G_1$ and M_1 be a maximum matching in G'_1, we can acquire an $\mathcal{EOU}$ labelling for vertices in G_i as described in Section 2.4.1, denoted by $\mathcal{E}_1$, $\mathcal{O}_1$ and $\mathcal{U}_1$. It was then proved that a rank-maximal matching of G_2 cannot use any edges that belong to the set of $\mathcal{O}_1\mathcal{O}_1$ or $\mathcal{U}_1\mathcal{U}_1$ edges. Besides, a rank-maximal matching in G_2 can not use any edge whose rank j is greater than 1 and connected to some vertex in $\mathcal{O}_1 \cup \mathcal{U}_2$. So if we delete the edges described above from $\mathcal{E}_2$ and let $G'_2 = G'_1 \cup \mathcal{E}_2$, a rank-maximal matching of G_2 can be found by augmenting M_1 in G'_2. This is because the augmentation would keep the number of vertices that are matched to first rank edges the same while maximising the number of vertices that are matched to the second rank edges, while it is clear that M_1 is a rank-maximal matching in G_1 (as there are only rank 1 edges and M_1 is maximum). Irving et al. extended this idea through mathematical induction and proved that if we have M_i as a maximum matching in G'_i and delete the edges with respect to $\mathcal{E}_i$, $\mathcal{O}_i$ and $\mathcal{U}_i$, we can have $G'_{i+1} = G'_i \cup \mathcal{E}_{i+1}$. More specifically,given G'_i and M_i, we delete all $\mathcal{O}_i\mathcal{O}_i$ or $\mathcal{U}_i\mathcal{U}_i$ edges; we also delete all edges incident to a vertex in $\mathcal{O}_i \cup \mathcal{U}$ all E_j where $j > i$. Then we add E_{i+1} to G'_i and the result is G'_{i+1}. M_i can then be augmented in G'_{i+1} to obtain M_{i+1} Irving et al[6] proved that:

Theorem 2.5.1 *[6] For any $1 \leq i \leq r$, a rank-maximal matching in G_i is a maximum matching in G'_i; and M_i that is acquired through the above mentioned process is a rank-maximal matching in G_i.*

Sng[16] discussed rank-maximal matching in the context of CHAT. The rank-maximal

algorithm for CHAT is essentially the same as the algorithm devised by Irving et al.[6] to CHAT. The only obvious difference is that the $\mathcal{EOU}$ labelling needs to be conducted with the help of cloned graphs $C(G)$ and $C(M)$. The process is similar to that discussed in Section 2.4.3. The algorithm is presented in Section 8.The time complexity of the algorithm for CHAT is $O(min(z\sqrt{C}, C + z)m)$, where z is the maximal rank of an edge in an optimal solution and C is the total capacity of all houses.

2.6 The matching algorithm toolkit

A matching algorithm toolkit is now available thanks to the efforts of many students in the School of Computing Science. One of the aims of the project is to integrate the algorithms into the existing matching algorithm toolkit. It is essential to have an understanding of the design and architecture of the toolkit. In this section, a brief introduction of the toolkit is given.

The matching algorithm library were initially developed by Aleš Remta [12], an MSc student at University of Glasgow, during his Masters project in academic year 2009-2010. The idea was to create a matching algorithms library as a common reusable framework for various algorithms dealing with matching problems including House Allocation problem. Several algorithms, including a maximum cardinality Pareto optimal matching algorithm for HA that was known to contain bugs, were developed previously by others students at the school of computing science and were integrated into the library. The detail of these algorithms and their developer can be found in [12]. The library, however, did not come with a Graphical User Interface (GUI) and client code was needed in order to use the algorithms in the library.

In academic year 2010-2011, Philip Yuile, a Level 4 student, designed a GUI for the matching algorithm library as a part of his project. The GUI allows someone with no substantial programming knowledge to have access to the algorithms. Yuile also added a few algorithms for House Allocation to the library, including an algorithm implemented by Asim Mahmood in academic year 2009-2010 that finds a popular matching in an HA instance if one exists. The detail of these algorithms can be found in [17].

Augustine Kwanashie, a current Ph.D. student in the school then extended the application further by adding algorithms and modifying the GUI. This is the current state of the

matching algorithm toolkit and it is still under continuous development.

2.6.1 The library

There were four logical components originally in the framework of the matching algorithm library. Those are *Reader, Model, Solver,* and *Writer*.

The Model object is the representation of the problem. More specifically, the Model contains Agent objects which are representation of applicants and houses in the case of House Allocation problem. An Agent may or may not have a preference list, which is represented by a PreferenceList object. A PreferenceList object has a instance variable referring to a list of PreferenceListElement objects. These objects and classes also provided methods to facilitate basic manipulation of the Agents, an *engageTo* method for assigning an Agent object to another for example.

The Solver class provides, as its name suggests, facility that involved in solving a matching problem. It is reusable for all matchings. Each Solver object holds an Algorithm object (say *paretoCHA*). The Solver will evaluate whether the input Model object is of a correct type for the algorithm. Solver objects also store Matching objects which is the result of the algorithm. Besides, Solver keeps track of the statistics relevant to the matching.

The Reader reads specified input (from text files) while the Writer controls the output of the matching library. Two types of writers are provided originally: matching writer, and the problem instance writer. The former prints out the matching and its relevant statistics for evaluation. The latter outputs a randomly generated instance and is only available for limited types of problems.

A random instance generator is also provided in the library.

2.6.2 The GUI

The GUI of the toolkit was designed to consist of two components, the Model and the View (the Controller, with respect to M-V-C pattern, was integrated with the view).

The view component was implemented using Java AWT and Swing. it contains all the input panels and output tabs, but has no direct interaction with the matching algorithm

library. The model component handles the interaction between the GUI and the library. It also keeps record of users input from the panels of the GUI's view component as well as the problem instances and solvers.

An introduction to the GUI can be found in Appendix F.

2.7 Other existing software applications dealing with House Allocation problem

Matching algorithms are widely studied and implementations of some matching algorithms are available in on-line libraries such as LEDA (*Library of Efficient Data Type and Algorithms*) and LEMON (*Library for Efficient Modelling and Optimisation in Networks*).Both *LEDA* and *LEMON*, for example, provided an maximum matching algorithm in weighted graph.

With respect to House Allocation problem and its extensions, few relevant software applications or source code are available to the public. There is an implementation of rank-maximal matching algorithm implemented by Dimitrios Michail available on [9]. However, as House Allocation problem is a sort of problem that many organisations face regularly, there exist certain practical systems that deal with this sort of problem. The School of Computing Science at University of Glasgow, for example, have a system that assigns students to projects which is based on generous maximum matchings in CHA instances. MIT has also developed a system to assign students to housing [10]. The implementation and design details of these systems however are again not available to public.

Chapter 3

Design

The major design decisions and the pseudo code of the algorithms will be introduced in this chapter.

3.1 Algorithms

Three algorithms – *maximum cardinality Pareto optimal matching algorithm for CHA*, *popular matching algorithm for CHAT* and *rank-maximal matching algorithm for CHAT* – are implemented during the project, because these algorithms can deal with extended instances such as those with capacitated houses and ties.

3.1.1 Maximum cardinality Pareto optimal matching algorithm for CHA

The algorithm for finding a maximum Pareto optimal matching in I was presented by Sng [16, 7]. Phase 1 involves the augmenting path algorithm for finding a maximum matching in the capacitated bipartite graph G of CHA instance I. The augmenting path algorithm keeps searching for and eliminating augmenting paths until there does exist any augmenting path in G. Let M_1 denote the result of Phase 1.

Phase 2 convert M_1 to a trade-in-free matching. This phase involves repeatedly searching for applicants who prefer an undersubscribed house to the houses they're currently

matched to. If any such applicant is found, the applicant will be promoted to the house he/she prefers. Note that the re-assignment of an applicant will create a vacancy in the applicant's previous house, which may be the preferred house of some other applicant. The process will be repeated until no promotion is possible. The algorithm is presented in Algorithm 4.

As part of the initialisation for Algorithm 4, for each house h_j we create a linked list L_j which initially consists of the set of pairs (a_i, r) where a_i prefers h_j to $M(a_i)$ and r is the rank of h_j on a_i's preference list. Also a stack S which contains all undersubscribed house h_j where L_j is non-empty is constructed. And let $curr_i$ store the rank of a_i's current assignment.

In the main loop of Algorithm 4, an undersubscribed house h_j is popped from S. Then remove a pair (a_i, r) from L_j.

If a_i prefers h_j to $M(a_i)$, we store $M(a_i)$ in variable h_k and re-assign a_i to h_j. Also $curr_i$ is updated to r. Then we need to check if h_j is still undersubscribed because h_j might have capacity greater than 1. if h_j is undersubscribed and L_j is non-empty, it should be pushed back onto the stack S. Finally because house h_k is undersubscribed now, it should also be checked whether L_k is non-empty. If L_k is non-empty and h_k is not already on stack S, then h_k is pushed onto S.

If a_i prefers $M(a_i)$ to h_j then h_j is pushed directly back onto S.

Phase 3 of the Pareto optimal matching algorithm is basically a generalisation of Gale's TTC algorithm to CHA. Given the matching result of Phase 2, denoted by M_2, Phase 3 turns M_2 into a coalition-free matching. Phase 3 involves repeated searching for and eliminating cyclic coalition. The algorithm is provided in Algorithm 5.

Let there be a pointer for every applicant a_i, denoted by $f(a_i)$. The pointer always point to the first element in applicant a_i's preference list, which might change during the process of the algorithm. A linked list L_j is created for each house h_j. L_j contains the applicants who prefer h_j to their currently matched houses. Note these lists are differently defined from the lists used in Phase 2 of the algorithm.

At the beginning of the algorithm, a copy of the input matching M is made, denoted by M'.

The main loop of the algorithm utilises a stack of applicants, denoted by P, which is

Algorithm 4 Maximum Pareto optimal matching algorithm Phase 2 for CHA [7]

1: **while** $S \neq \emptyset$ **do**
2: Let $h_j = S$.pop();
3: Let $(a_i, r) = L_j$.remove(0);
4: Let $curr_i$ be the rank of a_i's current matching on a_i's preference list;
5: **if** $r < curr_i$ **then**
6: Let $h_k = M(a_i)$;
7: Remove (a_i, h_k) from M;
8: Add (a_i, h_j) to M;
9: Set $curr_i$ to r;
10: **if** h_j.isUndersubscribed() & $L_j \neq \emptyset$ **then**
11: S.push(h_j);
12: **end if**
13: Let $h_j = h_k$;
14: **end if**
15: **if** $L_j \neq \emptyset$ & !S.contains(h_j) **then**
16: S.push(h_j);
17: **end if**
18: **end while**

the representation of an envy graph. The stack P helps to detect and eliminate cyclic coalitions. A queue Q, which contains applicant a_i who are waiting to be assigned to $f(a_i)$ in M, is maintain throughout the algorithm. The queue Q is updated in the subroutine **process**(Q), which is shown in Algorithm 6.

The subroutine **process**(Q) assigns every applicant a_i in Q to $f(a_i)$, denoted by h_j, and then removes (a_i, h_k) from M' (h_k being a_i's original assignment. Applicant a_i is then labelled so that it will not be processed again (this means every applicant will be processed for only once). Then if Q is not empty, the first element in Q should be removed from L_k. Because the first(next) element on Q will be assigned to its first choice house soon and it is unnecessary to update its preference list. If a_i is on stack P, it is also necessary to remove a_i from P. If $M'(h_k)$ at this stage is empty, it means the house h_k can no longer be involved in a coalition, therefore we delete h_k from the preference list of every unlabelled applicant a_t that is on L_k. Note that this will cause $f(a_t)$ to be updated, so it is necessary to check after each deletion whether $f(a_t) = M(a_t)$ and if so add a_t to Q to be processed.

Algorithm 5 starts by copying M. After that Q is initialised, and then every applicant a_i where $f(a_i) = M(a_i)$ is added to Q to be processed. Then the **process**(Q) subroutine is executed for the first time. Then we enter into the main loop of the program. As was mentioned before, a stack P is used here. For each unlabelled and matched applicant a_i, we use P to keep record of paths that starts from a_i. The path P is initialised to contain only a_i and then built in the **while** loop. A counter z_i is kept for each applicant a_i to keep record of the number of times that a_i is on P. Note that if a_i is on P more than once, it means there is a coalition. In the inner while loop, we start by popping an applicant a_j from P. If $z_j \neq 2$, we push a_j back onto stack P and push an applicant a_k from $M'(f(a_j))$ (a_j envies a_k) onto the P. Counter z_k is thus incremented by 1. If in the inner while loop $z_j = 2$, a coalition is discovered and the coalition is eliminated/satisfied by adding popping the applicants involved from P and adding them to Q to be processed. The subroutine **process**(Q) was then called. The subroutine as described before match each applicant a_i to $f(a_i)$ in M.

The correctness of Algorithm 5 is supported by Theorem 2.3.2.

Algorithm 5 Maximum Pareto optimal matching algorithm Phase 3 for CHA [7]

Make M' a copy of M;
$Q = \emptyset$; {Q is an empty queue}
for applicant $a_i \in I$ **do**
  if $f(a_i) = M(a_i)$ **then**
    Q.add(a_i);
  end if
end for
Execute subroutine process(Q);
for each unlabelled and matched applicant a_i **do**
  $P = \{a_i\}$; {P is a stack of applicants}
  $z_i = 1$;
  while $P \neq \emptyset$ **do**
    $a_j = P$.pop();
    if $z_j = 2$ **then**
      $a_k = a_j$;
      repeat
        Q.add(a_k);
        $a_k = P$.pop();
      until $a_k = a_j$
      Execute subroutine process(Q);
    else
      P.push(a_j);
      Choose an applicant a_k from the set $M'(f(a_j))$; {Assume the first one in the set is selected}
      increment z_k by 1;
      P.push(a_k);
    end if
  end while
end for

Algorithm 6 Process(Q) [7]

1: **while** $Q \neq \emptyset$ **do**
2: $a_i = Q$.remove(0); {Q is a queue of applicants waiting to be processed}
3: $h_j = f(a_i)$;
4: $h_k = M(a_i)$;
5: remove (a_i, h_k) from M; {M is the current matching}
6: add (a_i, h_j) to M;
7: remove (a_i, h_j) from M';
8: label a_i;
9: **if** $Q \neq \emptyset$ **then**
10: remove Q.get(0) from L_k;
11: **end if**
12: **if** $a_i \in P$ **then**
13: remove a_i from P;
14: **end if**
15: **if** $M'(h_k) \neq \emptyset$ **then**
16: **for all** unlabelled $a_t \in L_k$ **do**
17: remove h_k from a_t's preference list;
18: **if** $f(a_t) = M(a_t)$ **then**
19: Q.add(a_t);
20: **end if**
21: **end for**
22: **end if**
23: **end while**

3.1.2 Popular matching algorithm for CHAT

The algorithm for calculating a maximum cardinality popular matching was introduced by Sng [16]. The algorithm is shown in Algorithm 7.

Given a CHAT instance I and the underlying bipartite graph G, the maximum popular matching algorithm starts by constructing G_1, which is a sub-graph of G that contains all the first-choice edges, and calculating a maximum matching M_1 in G_1. The next step is to obtain a Gallai-Edmonds decomposition ($\mathcal{EOU}$) labelling with respect to M_1 and G_1. This can be achieved by first creating a cloned graph $C(G)$ and the corresponding maximum matching $C(M_1)$, and then identifying alternating path starting from exposed vertices. The process is similar to searching for augmenting path (which is essentially a special kind of alternating path), the algorithm of which was presented in Algorithm 2.

With the $\mathcal{EOU}$ labelling, $s(a)$, which is the set of highest ranking even edges, can be calculated. Then G' can be constructed with $(a_i, f(a_i)$ and $(a_i, s(a_i))$ edges. Then we delete all edges in G' that connects an odd vertex with an odd or unreachable vertex. Then M_1 is augmented again to find a maximum matching M in G'. Now if M is not an applicant-complete matching, I does not admit a popular matching. If M is an applicant-complete matching, then M is a popular matching in G and the next step is to convert M to a maximum popular matching. First all edges in G' incident to a last resort house should be deleted and then a new maximum matching M' can be obtained by augmenting M in G'. If M' is not applicant-complete, we will assign the unmatched applicants to their last resort houses. The final M' is a maximum popular matching in I.

[3] provided an example of using the algorithm step by step with an HAT instance.

Algorithm 7 Maximum popular matching algorithm for CHAT

Build graph $G_1 = (A \cup H, E_1)$. $(E_1 = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i)\})$
Compute a maximum matching M_1 of G_1
Obtain a Gallai-Edmonds decomposition $\mathcal{EOU}$ of G_1 using $C(G_1)$ and $C(M_1)$
Build sub-graph $G' = (A \cup H, E')$. $(E' = \{(a_i, h_j) : a_i \in A \wedge h_j \in f(a_i) \cup s(a_i)\})$
Delete all $\mathcal{OO}$ edges and $\mathcal{OU}$ edges from G'
Compute a maximum matching M in G'
if M is not a applicant-complete matching in G' **then**
  return "The instance does not admit a popular matching";
else
  Remove all edges incident to a last resort house from G_1 $\{G_1 = \{A \cup H, E_1\}\}$
  Augment M on G', resulting in M';
  if M' is not applicant complete **then**
    match unassigned applicants to their last resort houses;
  end if
  return M as a popular match of the instance;
end if

3.1.3 Rank-maximal matching algorithm for CHAT

Sng [16] extended the algorithm devised by Irving et al. [6] (which was for HAT) to the CHAT case. The algorithm is given in Algorithm 8. The rank-maximal matching algorithm starts by making E'_k a copy of the set of all rank k edges. Then we create a sub-graph of G' containing only rank 1 edges, denoted by G'_1, and calculate a maximum matching in G'_1 with augmenting path algorithm, denoted by M_1. With G'_1 and M_1 we start the main loop of the algorithm. For i from 1 to r, the algorithm first obtains an $\mathcal{EOU}$ labelling of the vertices in G'_i. This is done with the same technique used in popular matching algorithm for CHAT, namely through clone graphs. After labelling, all edges in G'_i connecting connecting an odd vertex to an odd or unreachable vertex are removed. Also all edges incident to odd or unreachable vertices are removed from $E'_j (j > i)$. Then we add the edges in E'_{i+1} to the graph G'_i and name the result G'_{i+1}. We also augment M_i in G'_{i+1} to obtain a maximum matching in G'_{i+1}. This new matching is denoted by M_{i+1}. The new sub-graph G'_{i+1} and the matching M_{i+1} then enters the next iteration of the loop. The algorithm goes on until $i = r - 1$, in which case the resulting graph would be G'_r and M_r. The matching M_r is a rank-maximal matching in $G_r = G$.

Algorithm 8 Rank-maximal matching algorithm for CHAT

for $k = 1 to r$ **do**
  Let E'_k be a copy of E_k;
end for
Let $G'_1 = G_1$ $\{G_1 = (A \cup H, E_1)\}$
Compute a maximum matching M_1 in G'_1
for $i = 1$ to $r - 1$ **do**
  Create $C(G'_i)$ (the cloned graph of G'_i) and its maximum matching $C(M_i)$ (with respect to M_i);
  Obtain a Gallai-Edmonds decomposition from $C(G'_i)$ and $C(M_i)$;
  Let vertices in G'_i inherit the Gallai-Edmonds decomposition $\mathcal{E}_i$, $\mathcal{O}_i$, and $\mathcal{U}_i$ from $C(G'_i)$;
  Delete all $\mathcal{O}_i\mathcal{O}_i$ edges and $\mathcal{O}_i\mathcal{U}_i$ edges from G'_i
  Delete all edges incident to a vertex in $\mathcal{O}_i \cup \mathcal{U}_i$ from E'_j for all $j > i$
  Add E'_{i+1} to G'_i and obtain G'_{i+1}
  Compute a maximum matching M_{i+1} in G'_{i+1} by augmenting M_i
end for
return M_r as a rank-maximal matching

3.2 Data Structure

The toolkit originally provides certain data structures such as the Agent class. These are discussed in the Section 4.3. This section is focused only on the data structures not originally from the toolkit but used in the algorithms.

All three algorithms to some extent involves the use of graph theory (mainly those regarding bipartite graph), so the data structures representing the elements in a bipartite graph are essential to this project. A **Vertex** class was developed and used in all three algorithms. Each Vertex object, as its name suggests, represents a vertex(node) in the underlying bipartite graph of a matching problem. Each Vertex object holds a reference to the Agent object that it represents and the set of edges incident to a vertex are represented by an adjacency list (represent by an ArrayList). Each Vertex object should also hold a reference to a set Vertex objects at its assignments. This set is represented by *Java.Util.HashSet* because the class allows for constant time *add, remove* and *contains* operations. The Vertex class also has methods that support calculations such as setting and getting f-houses. A class diagram of the Vertex class is given in Figure D.3. A BipartiteGraph class was also implemented for each algorithm. A BipartiteGraph object is sometimes used to store a list (Java.Util.ArrayList) of applicant Vertex objects and a list of house Vertex objects. The Java ArrayList's *size, get, set* operations run in constant time and its *add* operation runs in amortized constant time. The use of ArrayList class ensures that the construction of lists of vertices and the retrieval of a specific element on a list (given its index) can be executed very efficiently. The constructor of the BipartiteGraph class bears the responsibility of building the bipartite graph according to the input from Model objects.

Note that several data structures from the *Java.Util* package were used. These are:

- ArrayList⟨E⟩
- LinkedList⟨E⟩
- HashSet⟨E⟩
- Stack⟨E⟩

The time complexity of some of the operations of these data structures are shown in Table 3.1. The *Java.Util.LinkedList* class uses a doubly linked list implementation. Re-

	add(E e)	remove(E e)	remove(int i)	contains(E e)	get(int i)
HashSet<E>	O(1)	O(1)	N/A	O(1)	N/A
ArrayList<E>	amortized O(1)	O(n)	O(n)	O(n)	O(1)
LinkedList<E>	O(1)	O(1)	O(n)	O(n)	O(n)

Table 3.1: Java.Util

moving a node in the list with a reference to node is $O(1)$ time, but retrieving an element at index i is $O(n)$ time unless i is at the head or tail of the list. The *Java.Util.ArrayList* has constant time *get(int i)* method. However, removal of an element takes constant time. The *Java.Util.HashSet*, as was mentioned before, provides very efficient basic operations but does not store elements in a certain order. Thus *Java.Util.HashSet* was used where orders of elements are not critical. *Java.Util.LinkedList* was used where order is needed and removal is of elements happens often. Besides, the LinkedList was also used to represent the queue used in Algorithm 5. The *Java.Util.LinkedList* was used in cases where *get* operation was needed.

3.3 Integration

To integrate algorithms into the toolkit, it is necessary to understand the design and architecture. As was mentioned in Section 2.6.1, interfaces were provided for every major component and factory pattern was used. The new algorithm must implement the **Algorithm** interface and the relevant factory class must be modified to facilitate the new Algorithm class. Figure 3.1 shows the part of toolkit that is most relevant to the algorithms, with maximum Pareto optimal matching algorithm for CHA as an example. The class diagrams for popular matching algorithm and rank-maximal matching algorithm are similar to this and given in Appendix D.

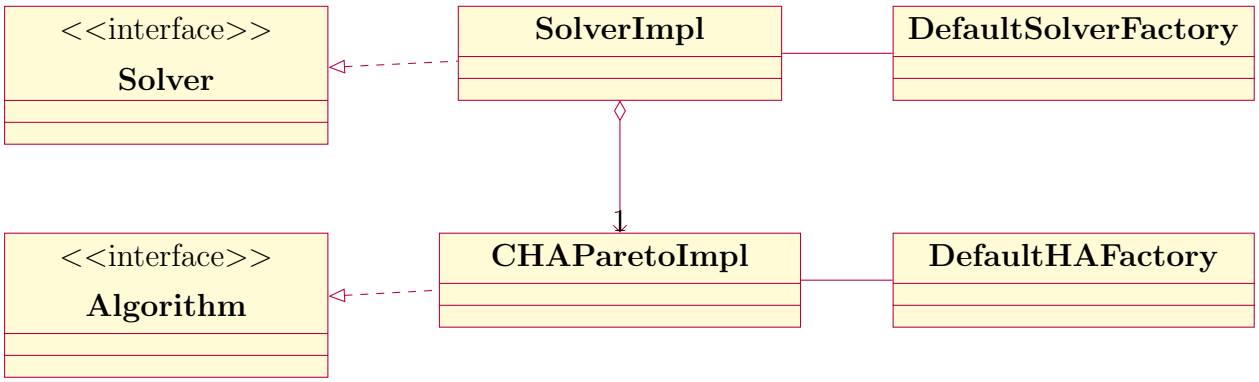

Figure 3.1: Class diagram for Pareto optimal matching algorithm

Chapter 4

Implementation

As the toolkit was developed using Java, the algorithms were also developed in Java. An iterative development methodology was adopted for each algorithm (maximum Pareto optimal matching algorithm for CHA, popular matching algorithm for CHAT, and rank-maximal matching algorithm for CHAT) in the project. This chapter illustrates the development stage of the project.

4.1 Algorithms

Three algorithms are implemented during the project, following the design and architecture of the matching algorithm library. The corresponding Java classes are:

- Maximum cardinality Pareto optimal matching algorithm for CHA
  — *impl.solver.algorithms.ha.CHAParetoImpl.java*
- Maximum cardinality popular matching algorithm for CHAT
  —*impl.solver.algorithms.ha.popular.CHATPopularImpl.java*
- Rank-maximal matching algorithm for CHAT
  —*impl.solver.algorithms.ha.profile.CHATProfileRankMaximalImpl.java*

Each of these Java classes implements the **Algorithm** interface provided in the toolkit, as was illustrated in Figure 3.1. As was required by the interface, each of the above

Figure 4.1: Input and Output of the Algorithm Classes

classes has a *run* method which takes in a **Model** object as a parameter and returns a **Matching** object as its result, as demonstrated in Figure 4.1. The constructor of an Algorithm class requires a MatchingFactory object as a parameter, which as its name suggests, is used (as in factory pattern) for creating Matching objects.

4.1.1 Maximum cardinality Pareto optimal matching algorithm for CHA

At the beginning of the *run* method the Model object is cast into a ModelHA object, as this is a model for House Allocation problem. Because the maximum Pareto optimal matching algorithm for CHA cannot deal with instances that have ties between houses on a preference list, it is necessary to check whether ties existing in the ModelHA object at the beginning of the *run* method. This facility, fortunately, is provided by the *hasTiedLists* operation the ModelHA class. Because the same ModelHA object might be used as input for several different algorithms, it is also necessary to make sure all applicants (and consequently all houses) are not assigned.

For the purpose of clarity, each phase of maximum Pareto optimal matching algorithm is represented by a method in the Java class. The structure of these algorithms are discussed in Section 3.1.1. It was discovered the AgentImpl provided a constant time *getRankOf* method for retrieving the rank of a house on an applicant's preference list. So in phase 2 of the algorithm there was no need to keep lists L_j of pairs (a_i, r) for each house, instead only lists of a_i was needed.

At the end of the algorithm after the matching is calculated and stored in the ModelHA object, a Matching object is instantiated using the *instanceOfMatching* method of the MatchingFactory object. The information of the matching is then added to the Matching object by pairs of *(applicant, house)*.

4.1.2 Maximum cardinality popular matching algorithm and rank-maximal matching algorithm

The algorithms for calculating popular matching and rank-maximal matching in CHAT instances were implemented following the same approach used to develop maximum cardinality Pareto optimal matching algorithm for CHA. Different steps in these algorithms are separated into different private methods in order to make the algorithm as clear as possible. These two algorithms can deal with instances with ties, so there is no need to check whether the input Model object has ties. Besides, there is a chance that a given instance does not admit a popular matching and when this happens, the *run* method simply returns **null**.

4.2 Modification to the toolkit

In order to fully integrate the algorithms, small modifications were made to the relevant classes in the toolkit. An example of these relevant classes was shown in Figure 3.1. The GUI of the toolkit was also modified to replace the old algorithms with the new ones and adjust the order of the algorithms displayed. These changes to GUI are mainly made to *front_end.model.GUIModel.java*.

4.3 Legacy Code

The matching algorithms were implemented following the interfaces and structures defined by the library, so they inevitably interact and reuse with the classes and methods provided by the toolkit.The Agent class and the Model class were extensively used in the implemented algorithms. See Figure 4.3 for the structure of these classes in the toolkit. The methods of the AgentImpl class is also given in Appendix E. The methods often used are *engageTo, getRankOf, breakUp, breakUpFirst, isEngaged,* and *getIndex*. The *breakUp* methods of the AgentImpl class runs in linear time and it is believed that improvements can be made over its efficiency. *getRankOf* method runs in constant time with a Map representation. Besides, the Java code of augmenting path algorithm is modified on the basis of the code presented on Dr David Manlove's lecture slides.

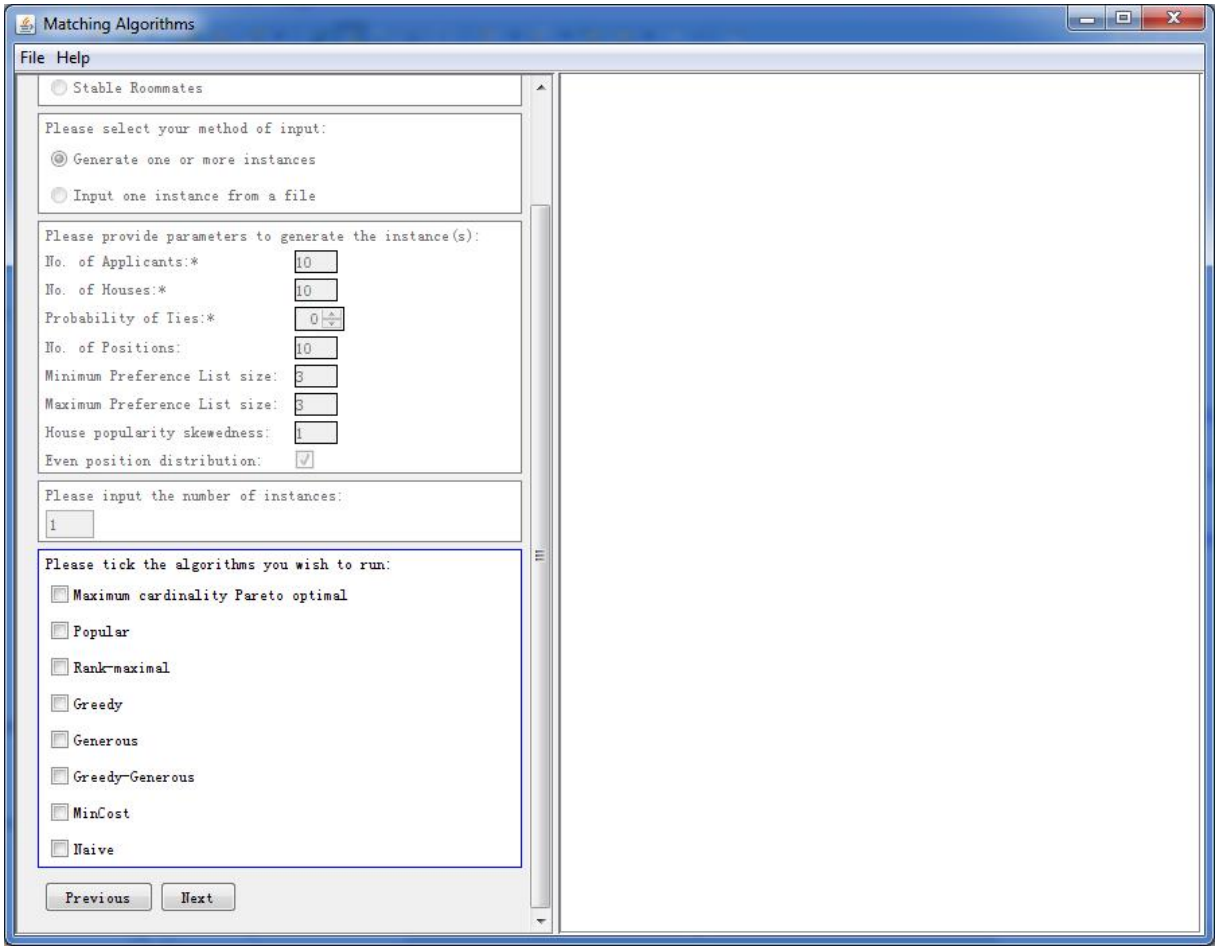

Figure 4.2: GUI after modification

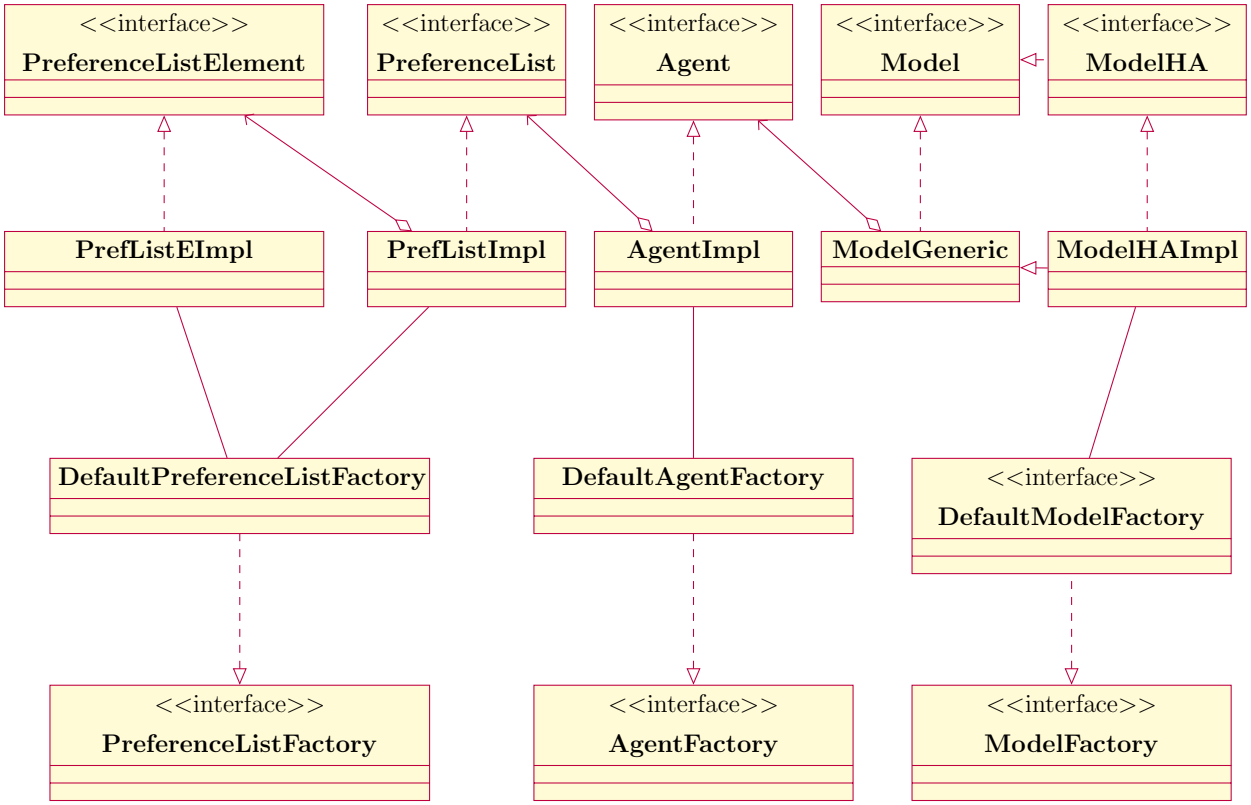

Figure 4.3: Model, Agent, PreferenceList, and PreferenceListElement [17]

4.4 Bugs Fixed for The Matching Algorithm Toolkit

As the matching algorithm toolkit is under continuous development with input from different developers, it is inevitable that the toolkit has certain bugs. The project in a way served as a test for the toolkit and some bugs were identified and fixed. The details of these bugs are discussed in this section.

Bug 1 (Status : fixed)

It was found that an Agent object representing a house (and is thus without a preference list) could not be matched (using engageTo method) to more than one Agent. The problem was caused by the *getAddPosition(Agent a)* method, which requires the Agent object to have a preference list in order to add a second engagement. The bug was fixed by adding a IF statement for the case where Agent object holds no PreferenceList object.

Bug 2 (Status : unfixed)

The GUI hides the algorithms unsuitable for the generated instances automatically. How-

ever, it was discovered that once an algorithm was disabled, even if an instance suitable for the algorithm was used (by going back to previous panels and changing instances), it would not appear again. The bug was caused by the *front_end.gui.GUI.disableUnsuitableAlgs()* method, but since it was not unclear how best to fix it, it was left untouched. In practice one could fix the problem by restarting the toolkit.

Bug 3 (Status : fixed)

It was found that the toolkit failed to display summarising statistics when multiple instances were used. Further examination revealed that the problem was caused by the *evaluation.Util.repairCost(Model m, MatchingStats stat, Matching matching)* method used in the *impl.solver.SolverImpl.initStats()*. The method throwed an exception whenever ModelHA was involved and the exception was neglected because it was incorrectly caught by a *catch* method. The bug was fixed through a small change to *impl.solver.SolverImpl.java*. The fix however, is temporary, and further re-factoring is believed to be necessary.

Bug 4 (Status : fixed)

The rank of a house on an applicant's preference list was incorrectly calculated. The error was caused by *impl.reader.DefaultReaderStrategy.makePrefList(int[] info, boolean[] tieNext, Agent[] agents)*.

Chapter 5

Evaluation

This chapter discusses the evaluation carried out on the algorithms implemented. As user evaluation is not plausible or useful for the algorithms (while the user evaluation of the GUI was conducted by Philip Yuile [17]), the evaluation in this project mainly consist of the correctness testing and the empirical evaluation.

5.1 Correctness Testing

The aim of correctness testing is to validate the correctness of the implementation of the algorithms. The critical methods of the algorithms were tested during development. After the implementation of each algorithm, a fair amount of effort was devoted to developing JUnit tests for the algorithms. A few sample instances were also used to manually verify the output of the algorithms.

JUnit supports the use of parameters in a test class. The test method in this parameterized test class is executed multiple times with the different parameters. In this case, a list of randomly generated Model instances are constructed as parameters for the tests. This approach makes it easy to test the algorithms over a large set of randomly generated instances. The testing strategy for the algorithms are discussed in the following sections.

5.1.1 Testing the maximum cardinality Pareto optimal matching algorithm for CHA

As demonstrated in Theorem 2.3.2, a matching is Pareto optimal only when it is maximal, trade-in-free, and coalition-free. In addition, the output of the maximum cardinality Pareto optimal matching algorithm should be a maximum matching. So we test the algorithm over randomly generated instances to see if the output matchings satisfy these standards.

Given a CHA instance I, to test whether a matching M in I is maximal and trade-in-free, one needs only to traverse the list of applicants once. If any applicant a_i is found such that a_i is unassigned while there exists a house h_j on a_i's preference list that is unmatched, or a_i is assigned yet a_i prefers an unmatched house h_j to his/her current assignment h_k, the matching M would be either not maximal or not trade-in-free and thus the algorithm would fail to pass this test. To test whether a matching M is coalition-free, a process similar to that described in Algorithm 5 was used. Only this algorithm was comparatively simpler. A depth first search utilising stack was used to detect the existence of cycles. If no cycles exists, then M is coalition-free and the algorithm passes the test.

To test whether a matching M is maximum, it is necessary to introduce the concept of a *vertex cover*[11]. In the discipline of graph theory, given a graph G, a vertex cover is a set C of vertices such that every edge in G is incident at least to one vertex in C.

There exists a method to convert a maximum matching to a minimum vertex cover in a given graph. Given a non-capacitated bipartite graph $G = (A \cup H, E)$ and a maximum matching M in G, let Z denote the set of exposed vertices in A. Then we have $|Z| = |A| - |M|$. Let Y denote the set of vertices in H that can be reached by an alternating path starting from a vertex in Z. Similarly, let X denote the set vertices in A that can be reached by an alternating path starting from a vertex in Z. Then $Z \subseteq X$. Let $C = (A \setminus X) \cup Y$. Also we have $|M| = |C|$ We can prove that if C is vertex cover, then M is a maximum matching. Let $\alpha(G)$ denote the size of a minimum vertex cover in G and $\beta(G)$ denote the size of a maximum vertex cover in G. König's theorem[11] states that $\alpha(G) = \beta(G)$ for a bipartite graph G. Suppose that the algorithm outputs a matching M. To test whether M is maximum, convert M to a vertex cover C using the process described above, ensure C is a vertex cover and check whether $|C| = |M|$. If so, we claim that M is maximum. For, suppose not, then $\beta(G) > |M|$. Thus by König's theorem there

exists a minimum vertex cover C' for G such that $|C'| = \alpha(G) = \beta(G) > |M| = |C|$. Thus C is a smaller vertex cover than C'. That's a contradiction. Hence M is a maximum matching after all.

Using this relationship between vertex cover and maximum matching in G, to prove that a matching M is a maximum matching, we only need to convert M to a vertex cover C and verify that C is a vertex cover in G. This can be achieved without much difficulty because for C to be a vertex cover, it is required for every applicant a_i that either a_i is in C or every neighbour of a_i must be in C in a bipartite graph.

Note that the above discussion is based on non-capacitated graphs, in case of capacitated graph, the cloned graph technique is used again and the theory above still stands.

Many different instances were generated to test the algorithms. The parameters used to generate the instances and the test results could be found in Appendix B.

5.1.2 Testing the maximum popular matching algorithm and rank-maximal matching algorithm for CHAT

For popular matching and rank-maximal matching, there exists no theory that could be easily found to help verify these criteria efficiently. So a brute-force (exhaustive) approach was used to test the algorithms. Given an instance I, all the possible matchings in I were generated and compared to the results of the algorithms. The recursive algorithm for generating all possible matchings given an CHAT instance was provided by Dr David Manlove and demonstrated in Algorithm 9. The method can be started by calling *choose(0)*. The *compare* method differs with the criteria.

Note that a given instance I might not admit a popular matching. When this happens, it is also necessary to verify whether it is true that no popular matching exists in I. Given the set of all possible matchings in I, denoted by $M = \{M_1, M_2, \ldots, M_k\}$, this can be done by comparing every matching M_i to every other matching in M. If there does not exist a certain matching M_j such that M_j is at least as popular as any other matching in M, then the algorithm passes the test for this instance I. The drawback of this testing method is that, obviously, it is very inefficient and does not scale well, especially in the case where popular matching does not exist. As a result the tests for popular matching algorithm and rank-maximal matching algorithm could only be conducted with smaller instances.

Algorithm 9 *choose(i)*

```
1: {Let n be the number of applicants and i be the index of applicants starting from 0.}
2: if i > n − 1 then
3:    Let M′ be the current matching;
4:    Execute subroutine compare (M′, M);
5: else
6:    Execute subroutine choose(i+1);
7:    for each h_j on a_i's preference list do
8:       if h_j is undersubscribed then
9:          Assign a_i to h_j;
10:         Execute subroutine choose(i+1);
11:         Unassign a_i from h_j;
12:      end if
13:   end for
14: end if
```

Again different instances were generated to test the algorithms. The parameters used to generate the instances could be found in Appendix B.

5.2 Empirical Evaluation

The toolkit provided the facility to conduct empirical evaluations over the output of the algorithms. Besides helping to verify the correctness of the algorithms, the evaluations can also contribute to better understanding in the properties of different matching criteria.

The first evaluation investigates how the likelihood of finding popular matchings in CHAT instances are affected by different parameters.

5.2.1 Maximum cardinality Pareto optimal matching

The maximum cardinality Pareto optimal matching algorithm can not be used on House Allocation problem instances with ties. Let the number of applicants equal to the number of posts, which is total capacity of all houses. $N = N_a = N_p$. The number of houses $N_h = 0.1 \times N$. For this kind of matching, we were interested in how the size, cost, regret

L	Avg. Size	Avg. Cost	Avg. No. of First	Avg. Regret
1	414.43	414.43	414.43	1
2	497.02	631.19	362.851	2
3	499.98	666.22	376.006	3
4	500	692.82	379.623	4
5	500	717.43	380.626	5
6	500	738.12	380.838	5.997
7	500	755.41	381.251	6.994
8	500	772.16	381.081	7.982
9	500	786.6	381.284	8.965
10	500	797.85	381.678	9.893
14	500	838.22	381.437	13.554
18	500	865.55	381.168	17.015
22	500	889.89	381.024	20.416
26	500	909.98	381.157	23.427
30	500	925.19	380.362	26.695
34	500	933.86	381.094	29.129
38	500	945	380.852	31.938
42	500	942.11	382.102	34.15
46	500	955.7	380.994	36.541
50	500	959.98	380.839	38.553

Table 5.1: Statistics of maximum Pareto optimal matchings when $N = 500$

and number of applicants that get their first choice change when the length of preference lists L change, given a fixed $N = N_a = N_h$. Again 1000 randomly generated instances were used for every case with different N. Table 5.1 shows the statistics when $N = 500$. More statistics can be found in Appendix C.

The observations we can make from Table 5.1 are:

1. Initially, average size of the matchings grows with the length of preference lists L. However, the average size soon reaches N and remains the same afterwards.

2. Naturally, average cost of the matchings increase with the length of preference lists.

3. The average number of applicants that get their first choice does not seem to be affected by the length of preference lists except when $L = 1, 2$. Besides, for a random instance, a large portion (over 70% in the case where $N = 500$) of applicants get their first choice.

4. Naturally, average regret of the matchings increase with the length of preference lists. However, when the preference lists are long enough (close to N), the average

regret is obviously smaller than the length of preference lists. This suggests that not many applicants are assigned to their least preferred houses when the preference lists are long.

5.2.2 Popular matching

Irving et al. [6] conducted an empirical evaluation on the probability that a popular matching exists over HAT instances. We extended the evaluation to CHAT instances. This probability are influenced by the number of applicants N_a, the number of houses N_h, the number of actual posts N_p (which is the total capacity of all houses), the lengths of the preference lists L, and the probability that an entry in a preference list is tied with its predecessor, denoted by T.

To make the experimental results comparable, it is necessary to establish restriction over certain variables. We start by letting the number of applicants equal to the number of posts, namely $N_a = N_p = N$. Also Let the number of houses $N_h = 0.1 \times N_a$. Note that the posts are not evenly allocated over houses. This means although the houses have average capacity of $\bar{c} = 5$, some houses might have capacity less than 5 while others have capacity greater than or equal to 5.

Evaluations were conducted over instances with $N = 100, 200, 300, 500, 1000$. For each case we vary T with in the set of values $\{0.0, 0.2, 0.4, 0.6, 0.8\}$. For the case where $N = 100$, we tested all the possible L values ($L \in \{1, 2, 3, \ldots, 10\}$). See the corresponding tables for the L values used in other cases. Also for each case 1000 randomly generated instances are used to conduct the research.

The results of the case where $N = 500$ were summarised in Table 5.2. More data can be found in Appendix C. Figure 5.1 provides a more intuitive demonstration of the relationship between L, T and number of instances that admits a popular matching.

The other cases show similar patterns and the details those cases can be found in the Appendix. There are a few observations we can make from these statistics:

1. When K and N are fixed, the greater T is, the more likely there is a popular matching. It is very likely that a popular matching exists when there is a high probability for ties.

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	1000	1000	998	992	999
	3	1000	992	922	843	996
	4	998	994	845	869	1000
	5	976	978	785	921	1000
	6	908	939	741	933	1000
	7	802	885	725	945	1000
	8	700	852	713	953	1000
	9	649	815	747	955	1000
	10	585	802	717	977	1000
	14	501	775	708	967	1000
	18	533	790	711	959	1000
	22	505	773	715	969	1000
	26	528	814	717	955	1000
	30	527	804	741	961	1000
	34	504	780	717	967	1000
	38	522	767	707	961	1000
	42	530	773	715	965	1000
	46	513	767	692	956	1000
	50	525	778	707	950	1000

Table 5.2: Numer of instances with a popular matching for $N = 500$

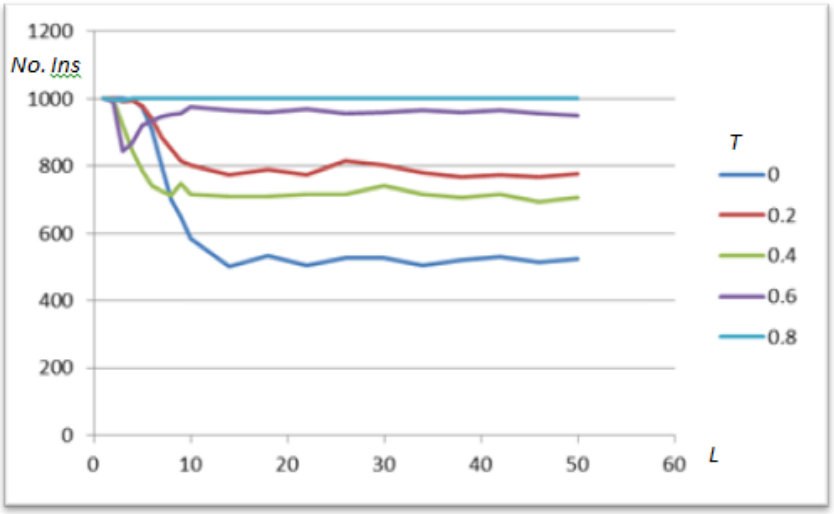

Figure 5.1: Numer of instances with a popular matching for $N = 500$

		N									
		100	200	300	400	500	600	700	800	900	1000
T	0.2	934	858	804	792	807	802	779	798	785	786
	0.0	833	750	665	613	569	564	539	549	509	460

Table 5.3: Number of solvable instances for popular matching algorithm when $L = 10$

2. When T and N are fixed and L is relatively small, increasing L has a negative effective over the probability of a popular matching. However, this effect will fade when L reaches a certain level.

3. The smaller T is, the faster the probability of a popular matching declines as L increases.

These results are understandable because longer the preference lists are, the more likely the applicants will have conflicting interests over different matchings, and the more difficult for there to be a popular matching. On the other hand, a high probability of ties generally makes it easier for applicants to be assigned to a higher ranked house and thus reduces the conflicts between applicants.

We could also explore the relationship between N and the probability of a popular matching existing by fixing L and T. The number of houses remained $N_h = 0.1 \times N$. Let $L = 10$ and $T = \{0.0, 0.2\}$, N was varied from 100, 200 to 1000. The results were summarised in Table 5.3. It can be observed that the more applicants and posts there are, the less likely there is a popular matching for a given instance. Again 1000 instances were randomly generated for each case.

It is also possible to investigate the relationship between N_h and the probability of there being a popular matching with N, L, and T fixed (again with 1000 instances for each case). Table 5.4 gives the experimental results when $N = 500$, $L = 5$, $T = 0$ while N_h varies (no longer fixed to $0.1 \times N$). The result suggests that with the number of applicants and posts (as well as other parameters) fixed, the more houses there are, the less likely the given instance admits a popular matching.

N_h	50	100	150	200	250	300	350	400	450	500
Number of solvable instances	969	832	501	209	62	20	21	4	6	0

Table 5.4: Relationship between number of solvable instances and number of houses

L	Avg. Size	Avg. Cost	Avg. No. of First	Avg. Regret
1	414.46	414.46	414.468	1
2	480.21	545.46	414.97	2
3	492.66	583.4	414.394	3
4	495.98	596.38	414.759	3.933
5	497.24	602.69	414.841	4.624
6	498.06	607.94	414.295	5.265
7	498.53	610.85	414.393	5.811
8	498.73	613.78	414.186	6.264
9	498.97	615.67	414.411	6.868
10	499.18	617.25	414.259	7.16
14	499.58	621.81	414.804	8.722
18	499.77	625.5	414.325	9.947
22	499.88	627.24	414.293	10.485
26	499.94	627.95	414.881	10.967
30	499.97	630.38	414.518	11.709
34	499.98	629.88	414.581	11.87
38	499.99	630.7	414.026	11.964
42	499.99	632.01	413.724	12.081
46	500	630.97	413.861	11.77
50	500	630.36	414.916	12.072

Table 5.5: Statistics of rank-maximal matching when $N = 500$ and $T = 0.0$ (no ties)

5.2.3 Rank-maximal matching

A table similar to Table 5.1 could be obtained for rank-maximal matchings. See Table C.12. The parameters used to generate instances were similar to those for Table 5.1. The probability of ties T was set to 0 (no ties).

The results give rise to the following observations:

1. The average size of matchings grows slowly with the length of preference lists and remains approximately the same after reaching a certain level (usually the maximum size).

2. The average cost of matchings, not surprisingly, grows with the length of preference lists when L is small and remains the approximately at the same level after reaching a certain level.

3. The average number of applicants that get their first choice remains unaffected by the length of preference lists

4. The average regret of the matchings also increase with the length of preference lists.

Although the results in Table 5.1 and Table C.12 were not derived from the same set of instances, the parameters used to generate those instances were of no difference. So these two tables are generally speaking comparable. By comparing these two tables, we can find that:

1. Maximum Pareto optimal matchings are comparatively more likely to be of a larger size, given a fixed length of preference lists and no ties. This is understand since the size of the matching is maximised.

2. Rank-maximal matchings tend to have lower average cost than maximum Pareto optimal matchings given a fix length of preference lists and no ties.

3. On average, more applicants get their first choice in rank-maximal matchings.

4. Rank-maximal matchings have lower average regret than maximum Pareto optimal matchings under the same conditions.

These observations are reasonable because the rank-maximal matching algorithm focus on the profile of matchings and tend to maximise the number of applicants that are assigned to higher rank houses. On the other hand, maximum Pareto optimal matching algorithm maximises the sizes of matchings and keeps the matching Pareto optimal without actually considering the ranks of edges.

The influence of ties over the profile of a rank maximal matching was also investigated. The statistics can found in Table 5.6. Again 1000 randomly generated instances were used for each case. The results suggest that a higher T leads to a rank-maximal matching of a larger size. Also because ties tend to make it easier for applicants to be assigned to higher rank houses, the average cost declines as T rises, while the average number of applicants assigned to their first choices increases as T rises. Note that when $T = 0.8$, almost

N = 500, L = 10					
	0	0.2	0.4	0.6	0.8
Avg. Size	499.17	499.62	499.84	499.94	500
Avg. Cost	616.49	570.58	528.97	503.76	500.21
Avg. No. of First	414.474	442.76	475.844	496.408	499.79
Avg. Regret	7.19	5.53	3.983	2.079	1.13

Table 5.6: Statistics of rank-maximal matching when $N = 500$ and $T = \{0.0, 0.2, 0.4, 0.6, 0.8\}$

all applicants are assigned to their first choice, probably because for most applicant a_i most houses are tied to each other and they are all first choices for a_i. Besides, a higher probability of ties also result in a lower average regret.

Chapter 6

Conclusion and Future Work

The achievements of this project is discussed in the chapter. Possible improvements and suggestions regarding both the algorithms and the matching algorithm toolkit are also summarised here.

6.1 Product status

The purpose of the project was very clear: to implement the maximum cardinality Pareto optimal algorithm, the popular matching algorithm, and the rank-maximal algorithm. All three algorithms were implemented as planned and they all passed the correctness tests successfully. Besides, these algorithms were also fully integrated into the matching algorithm toolkit. A fair amount of effort was devoted to conducting empirical evaluation over the output of the algorithms and a large set of data was obtained.

It is safe to say the final product of the project has met the core functional requirements listed in Appendix A. The non-functional requirements were also generally satisfied. The project itself also served as a test to the matching algorithm toolkit, especially the part relevant to house allocation problem. A few bugs from the toolkit were identified during the project and most of them have been corrected. In conclusion, the project was successfully conducted.

6.2 Possible Improvements and Future Work

Although an effort was made to improve the efficiency of the algorithms, it is believed that further improvements can be made. For instance, implementation of Line 12 of Algorithm 6 uses the *contains* operation of stack P, which is a linear time operation. It is possible to finish the step in constant time with the help of an array of booleans indicating which applicants are on the stack and which are not. Besides, the implementation of the maximum Pareto optimal matching algorithm used the *breakUp* operation of the *impl.Model.AgentImpl* class frequently. It was later realised that the operation was a linear time operation for capacitated houses because the *engagements* were implemented as a LinkedList. It should, however, be noted that the improvements in running time are usually made at the expense of memory space.

More algorithms relevant to House Allocation problem and the optimality criteria can be identified. For example, there are other known methods to find a rank-maximal matching in an HAT instance.

It is also believed that further empirical evaluation over the algorithms can be conducted. Comparison between the results of different algorithms over a same set of instances, for example, might lead to interesting observations.

With respect to the matching algorithm toolkit, there are also a few suggestions. First of all, it would be very helpful if it was possible to save the matching results from the algorithms. Currently, the toolkit displays the results after computation, but there is no way to save the problem instance generated and the matchings (say, in a text file). Secondly, it might be helpful to display to the matching results of a house allocation problem instance in a bipartite graph. Thirdly, for educational purposes, it might be useful if the toolkit can display the calculation process of the algorithms step by step. This would undoubtedly require a lot of changes to the current toolkit and algorithms, but it could help verify the correctness of the algorithms while making it easier for students to understand.

Appendix A

Requirements

A.1 Functional Requirements

The functional requirements are organised in accordance with their priority following the MoSCoW method.

Must have

- The maximum cardinality Pareto optimal matching algorithm for Capacitated House Allocation problem (CHA) instances.
- The popular matching algorithm for Capacitated House Allocation problem instances with ties (CHAT)
- The rank-maximal matching algorithm for CHAT
- Data structures with efficient operations to support the algorithms

Should have

- The full integration of the above mentioned algorithms into the matching algorithm toolkit
- Support for input and output data structure required by the matching algorithm toolkit

- Empirical evaluation over the probability of a CHAT instance admitting a popular matching
- Proper documentation for design and implementation details of the algorithms
- Optimal efficiency for algorithms

Could have

- Extensive empirical evaluation over maximum Pareto optimal matching and rank-maximal matching
- Reusable data structures
- Identification and removal of bugs from the toolkit where appropriate
- Support for Javadoc from the data structures implemented.

A.2 Non-functional Requirements

- The algorithms should be implemented in Java.
- The algorithms should be self-documented and supports future modifications.

Appendix B

Testing Parameters

The following parameters were varied when generating instances to test the algorithms:

- N_a — number of applicants
- N_h — number of houses (for non-capacitated cases $N_h = N_p$)
- N_p — number of posts, which is the total capacity of all houses. For all experiments $N_p = N_a = N$.
- L_{min} — the minimum length of preference lists
- L_{max} — the maximum length of preference lists
- T — the probability that a house on a preference list is tied to the next one.
- N_I — $N_I = 1000$. The number of instances generated.

The skewness of distribution was always set to 1 and the total capacity was unevenly distributed over the houses.

The parameters used to test maximum Pareto matching algorithm for CHA are shown in Table B.1. More scattered tests were conducted over small sets of instances.

The parameters used to test popular matching algorithm for CHAT and rank-maximal algorithm for CHAT are shown in Table B.2. Due to the limitation of the test method, only small-size instances were used. Other parameters were used to test the algorithms over small sets of instances.

Case	N_a	N_h	N_p	L_{min}	L_{max}	T	N_I	Pass?
1	10	10	10	3	3	0	500	Y
2	10	5	10	3	3	0	500	Y
3	50	50	50	3	5	0	500	Y
4	50	30	50	10	10	0	1000	Y
5	50	30	50	30	30	0	1000	Y
6	500	500	500	20	20	0	500	Y
7	500	100	500	10	15	0	500	Y
8	1000	1000	1000	50	50	0	100	Y
9	300	100	200	5	10	0	1000	Y
10	500	300	600	5	10	0	1000	Y
11	500	600	600	5	10	0	1000	Y

Table B.1: Test parameters for maximum Pareto optimal matching algorithm for CHA

Case	N_a	N_h	N_p	L_{min}	L_{max}	T	N_I	Pass?
1	8	8	8	3	3	0	800	Y
2	8	4	8	2	3	0	800	Y
3	10	10	10	3	5	0	800	Y
4	10	10	10	3	5	0.2	800	Y
5	10	10	10	3	5	0.4	800	Y
6	10	5	10	3	5	0	800	Y
7	10	5	10	3	5	0.2	800	Y
8	15	8	15	4	4	0.2	200	Y
9	20	10	20	3	5	0.2	10	Y

Table B.2: Test parameters for popular matching algorithm for CHAT and rank-maximal algorithm for CHAT

Appendix C

Empirical Evaluation Data

Parameters involved in the evaluations:

- N_a — number of applicants
- N_h — number of houses (for non-capacitated cases $N_h = N_p$, while for capacitated instances N_h is mostly set to $0.1 \times N_a$.
- N_p — number of posts, which is the total capacity of all houses. For all experiments $N_p = N_a = N$.
- L — the length of preference lists in a problem instance, which is just the number of applicants. All preference lists in the same instance are set to be of the same length.
- T — the probability that a house on a preference list is tied to the next one.
- N_I — $N_I = 1000$. That means in each case with different parameters 1000 randomly generated instances were used.

The skewness of distribution was always set to 1 and the total capacity was unevenly distributed over the houses.

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	983	987	993	996	999
	3	910	937	958	982	998
	4	775	865	932	971	997
	5	650	787	914	980	997
	6	612	803	894	970	999
	7	593	749	885	978	1000
	8	571	740	896	969	1000
	9	553	742	893	983	1000
	10	555	737	898	987	1000

Figure C.1: Number of instances that admit popular matching. Non capacitated ($N_a = N_h = 10$).

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	999	991	985	989	1000
	3	958	959	966	999	1000
	4	898	939	971	998	1000
	5	850	917	976	997	1000
	6	808	919	983	1000	1000
	7	817	912	977	1000	1000
	8	847	911	980	999	1000
	9	821	932	981	1000	1000
	10	834	911	985	999	1000

Figure C.2: Number of instances that admit popular matching. Capacitated ($N_a = 100, N_h = 10$).

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	997	994	989	993	998
	3	954	952	935	941	989
	4	843	867	882	919	995
	5	667	823	840	917	999
	6	582	719	822	934	1000
	7	473	683	822	949	1000
	8	421	673	797	946	1000
	9	410	650	793	947	1000
	10	393	665	809	954	1000
	11	398	640	796	958	1000
	12	402	654	789	950	1000
	13	398	633	803	953	1000
	14	384	649	802	965	1000
	15	410	641	770	952	1000
	16	403	632	792	950	1000
	17	388	633	799	957	1000
	18	393	617	790	957	1000
	19	396	621	801	968	1000
	20	418	640	804	952	1000

Figure C.3: Number of instances that admit popular matching. Capacitated ($N_a = 100, N_h = 20$).

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	1000	995	994	989	997
	3	995	985	944	960	1000
	4	954	953	889	981	1000
	5	895	911	890	994	1000
	6	813	889	896	993	1000
	7	780	891	899	996	1000
	8	746	885	879	996	1000
	9	698	850	895	994	1000
	10	714	859	882	997	1000
	11	718	865	900	996	1000
	12	715	856	874	994	1000
	13	740	872	899	995	1000
	14	716	855	891	998	1000
	15	742	874	908	997	1000
	16	746	831	895	997	1000
	17	730	860	905	997	1000
	18	748	850	891	997	1000
	19	733	864	900	997	1000
	20	708	867	904	996	1000

Figure C.4: Number of instances that admit popular matching. Capacitated ($N_a = 200, N_h = 20$).

		T				
		0	0.2	0.4	0.6	0.8
L	1	1000	1000	1000	1000	1000
	2	999	999	996	990	999
	3	996	988	935	915	998
	4	983	962	869	952	1000
	5	933	929	844	974	1000
	6	850	900	808	984	1000
	7	774	866	813	985	1000
	8	737	884	832	978	1000
	9	674	837	835	987	1000
	10	672	829	819	992	1000
	12	656	824	810	989	1000
	14	655	826	795	993	1000
	16	643	817	826	985	1000
	18	643	838	824	987	1000
	20	649	837	802	987	1000
	22	632	808	826	986	1000
	24	653	850	818	995	1000
	26	633	822	815	983	1000
	28	654	828	822	989	1000
	30	654	838	800	985	1000

Figure C.5: Number of instances that admit popular matching. Capacitated ($N_a = 300, N_h = 30$).

		N								
		100	200	300	400	500	600	700	800	
L	10	934	858	804	792	807	802	779	798	
	N	919	873	816	817	786	752	736	721	

Figure C.6: Relationship between N_a and number of instances that admit popular matching, with $K = \{10, N_a\}, T = 0.2$

		N								
		100	200	300	400	500	600	700	800	
K	10	833	750	665	613	569	564	539	549	
	N	832	741	670	577	526	475	388	382	

Figure C.7: Relationship between N_a and number of instances that admit popular matching, with $K = \{10, N_a\}, T = 0.0$

L	Avg. Size	Avg. Cost	No. of First	Avg. Regret
1	83.47	83.47	83.476	1
2	99.57	122.56	76.582	2
3	100	129.72	77.764	3
4	100	134.86	77.849	3.965
5	100	139.07	78.016	4.847
6	100	141.07	78.504	5.605
7	100	144.98	78.165	6.401
8	100	146.46	78.448	7.069
9	100	148.59	78.238	7.626
10	100	149.92	78.161	8.14

Figure C.8: Statistics of maximum Pareto optimal matching in CHA instances. $H_a = 100, H_h = 10$

L	Avg. Size	Avg. Cost	No. of First	Avg. Regret
1	166.62	166.62	166.624	1
2	198.89	249.31	148.47	2
3	199.99	262.79	152.98	3
4	200	274.13	153.729	3.999
5	200	283.04	154.006	4.991
6	200	290.85	154.088	5.952
7	200	298.52	153.826	6.869
8	200	304.33	153.89	7.757
9	200	307.77	154.084	8.527
10	200	313.23	153.894	9.357
11	200	317.26	153.615	10.106
12	200	319.59	153.643	10.825
13	200	320.8	154.216	11.493
14	200	325.49	153.893	12.355
15	200	328.95	153.661	12.995
16	200	331.01	153.877	13.681
17	200	331.01	153.557	14.08
18	200	334.25	154.061	14.879
19	200	334.78	153.922	15.267
20	200	336.84	153.594	15.71

Figure C.9: Statistics of maximum Pareto optimal matching in CHA instances. $H_a = 200, H_h = 20$

L	Avg. Size	Avg. Cost	No. of First	Avg. Regret
1	249.34	249.34	249.341	1
2	298.35	376.36	220.347	2
3	299.98	396.81	227.728	3
4	300	413.03	229.301	4
5	300	427.33	229.462	4.998
6	300	438.97	229.735	5.987
7	300	450.1	229.695	6.955
8	300	459.82	229.625	7.891
9	300	465.61	230.078	8.819
10	300	474.5	229.755	9.692
12	300	487.97	229.5	11.41
14	300	496.69	229.058	12.957
16	300	504.59	229.587	14.578
18	300	511.11	229.774	16.1
20	300	517.57	229.746	17.555
22	300	521.07	229.686	18.651
24	300	525.88	229.694	19.998
26	300	531.18	229.74	21.564
28	300	535.3	229.365	22.281
30	300	535.94	229.562	23.411

Figure C.10: Statistics of maximum Pareto optimal matching in CHA instances. $H_a = 300, H_h = 30$

L	T=0.0					T=0.6			
1	83.74	83.74	83.744	1		83.61	83.61	83.616	1
2	96.86	110.16	83.555	2		99.41	101.76	97.072	1.742
3	99.3	117.31	83.682	2.828		99.98	101	98.976	1.414
4	99.78	119.51	83.545	3.161		99.99	100.72	99.279	1.32
5	99.89	120.33	83.547	3.379		99.99	100.65	99.353	1.287
6	99.97	120.6	83.658	3.433		100	100.63	99.386	1.284
7	99.99	120.76	83.408	3.435		100	100.62	99.381	1.286
8	99.99	120.85	83.317	3.428		100	100.62	99.379	1.271
9	100	120.65	83.525	3.422		100	100.61	99.384	1.28
10	100	120.43	83.726	3.386		100	100.59	99.405	1.269
	T=0.2					T=0.8			
1	83.46	83.46	83.467	1		83.75	83.75	83.757	1
2	97.98	107.27	88.703	2		99.6	100.29	98.916	1.331
3	99.69	111.07	89.257	2.479		99.99	100.05	99.936	1.034
4	99.9	111.53	89.518	2.585		100	100.02	99.973	1.021
5	99.96	112.28	89.277	2.705		100	100.02	99.974	1.02
6	99.99	112.19	89.377	2.685		100	100.01	99.981	1.015
7	99.99	112.2	89.427	2.748		100	100.04	99.956	1.028
8	99.99	112.07	89.493	2.725		100	100.04	99.953	1.03
9	100	111.78	89.656	2.665		100	100.03	99.966	1.024
10	100	112.17	89.421	2.704		100	100.02	99.971	1.022
	T=0.4								
1	83.72	83.72	83.727	1					
2	98.83	104.09	93.565	1.965					
3	99.84	104.75	95.193	2.089					
4	99.95	104.42	95.746	2.066					
5	99.99	104.52	95.672	2.019					
6	99.99	104.47	95.796	2.046					
7	99.99	104.62	95.699	2.058					
8	99.99	104.42	95.82	2.006					
9	100	104.35	95.826	1.977					
10	100	104.55	95.685	2.036					

Figure C.11: Statistics of rank-maximal matching in CHA instances. $H_a = 100, H_h = 10$

L	T=0				T=0.2			
1	414.71	414.71	414.713	1	414.71	414.71	414.716	1
2	480.03	545.58	414.491	2	485.97	532.34	439.612	2
3	492.74	583.32	414.679	2.999	495.94	555.11	442.412	2.974
4	495.98	597.51	414.025	3.954	498.04	561.76	442.434	3.558
5	497.31	603.84	414.347	4.653	498.67	564.63	442.572	4.004
6	497.97	607.24	414.726	5.234	498.97	566.94	442.15	4.428
7	498.46	611.45	413.926	5.8	499.28	568.87	441.798	4.763
8	498.67	612.48	415.01	6.35	499.4	569.91	442.224	5.144
9	499.02	614.72	414.687	6.635	499.48	570.65	442.355	5.417
10	499.17	616.49	414.474	7.19	499.62	570.58	442.76	5.573
14	499.57	622.86	414.255	8.953	499.75	573.91	442.387	6.572
18	499.78	624.64	414.391	9.63	499.88	574.94	442.64	7.212
22	499.87	628.55	414.502	11.017	499.95	576.09	442.57	7.674
26	499.93	629.46	414.463	11.436	499.96	577.37	442.098	8.011
30	499.96	630.41	414.077	11.662	499.98	577.16	442.429	8.052
34	499.99	629.7	414.69	11.741	499.99	577.24	442.416	8.354
38	499.99	629.53	414.271	11.537	499.99	577.69	442.62	8.391
42	499.99	631.11	414.189	12.079	500	577.81	442.745	8.367
46	500	629.85	414.379	11.537	500	578.49	441.883	8.61
50	500	631.07	414.798	12.054	500	577.9	442.665	8.647
	T=0.4				T=0.6			
1	414.88	414.88	414.882	1	414.51	414.51	414.518	1
2	491.18	519.16	463.211	2	495.25	508.09	482.414	1.998
3	497.62	523.35	473.749	2.775	498.86	503.15	494.749	2.059
4	498.84	525.03	475.429	3.092	499.58	503.34	496	2.033
5	499.4	526.17	476.014	3.306	499.77	503.43	496.261	2.006
6	499.57	526.4	476.332	3.409	499.85	503.52	496.363	2.019
7	499.71	527.71	475.792	3.521	499.92	503.72	496.345	2.015
8	499.8	527.36	476.584	3.685	499.93	503.75	496.329	2.047
9	499.85	528.02	476.25	3.763	499.93	503.98	496.201	2.133
10	499.84	528.97	475.844	3.983	499.94	503.76	496.408	2.079
14	499.94	528.82	476.827	4.264	499.98	504.01	496.327	2.098
18	499.97	529.37	476.281	4.232	499.99	504.23	496.489	2.291
22	499.99	530.24	475.965	4.433	499.99	504.34	496.259	2.304
26	499.99	529.53	476.214	4.389	499.99	504.33	496.341	2.315
30	499.99	529.82	476.553	4.518	500	504.31	496.266	2.303
34	499.99	531.66	475.948	4.984	500	504.22	496.503	2.331
38	499.99	529.72	476.317	4.611	499.99	504.32	496.392	2.315
42	500	531.02	475.662	4.683	500	504.49	496.298	2.34
46	500	530.25	476.119	4.642	500	504.37	496.387	2.442
50	500	530.52	476.604	4.789	500	504.37	496.369	2.3

Figure C.12: Statistics of rank-maximal matching in CHA instances. $H_a = 500, H_h = 50$

Appendix D

Class Diagrams

Figure D.1 and Figure D.2 show the class diagrams for classes related to popular matching algorithm and rank-maximal matching algorithm. A class diagram of the Vertex class implemented is shown in Figure D.3.

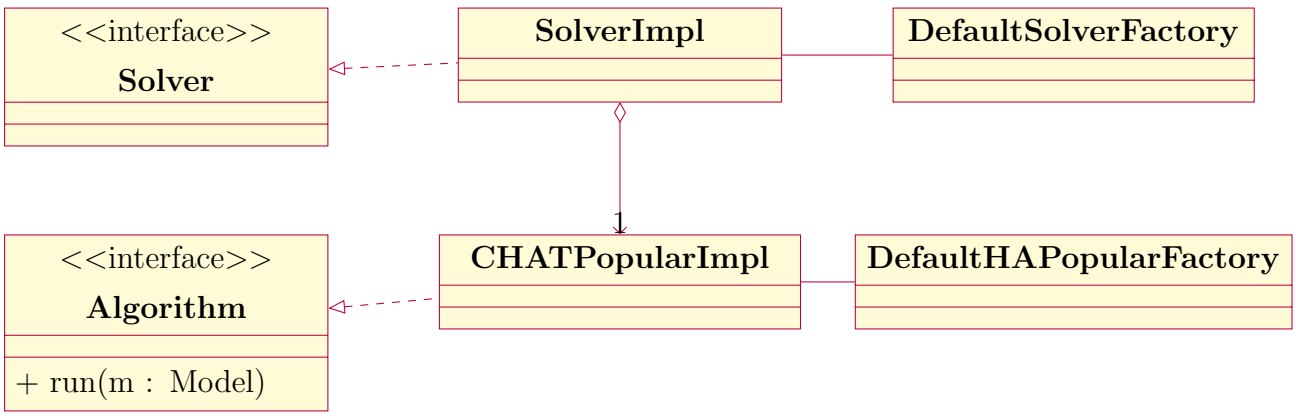

Figure D.1: Class diagram for Popular matching algorithm

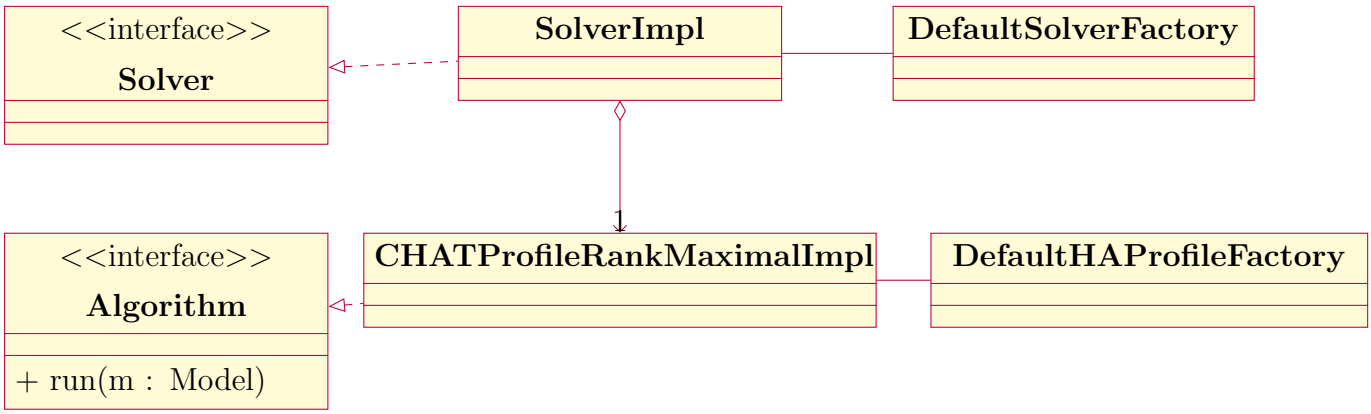

Figure D.2: Class diagram for rank-maximal matching algorithm

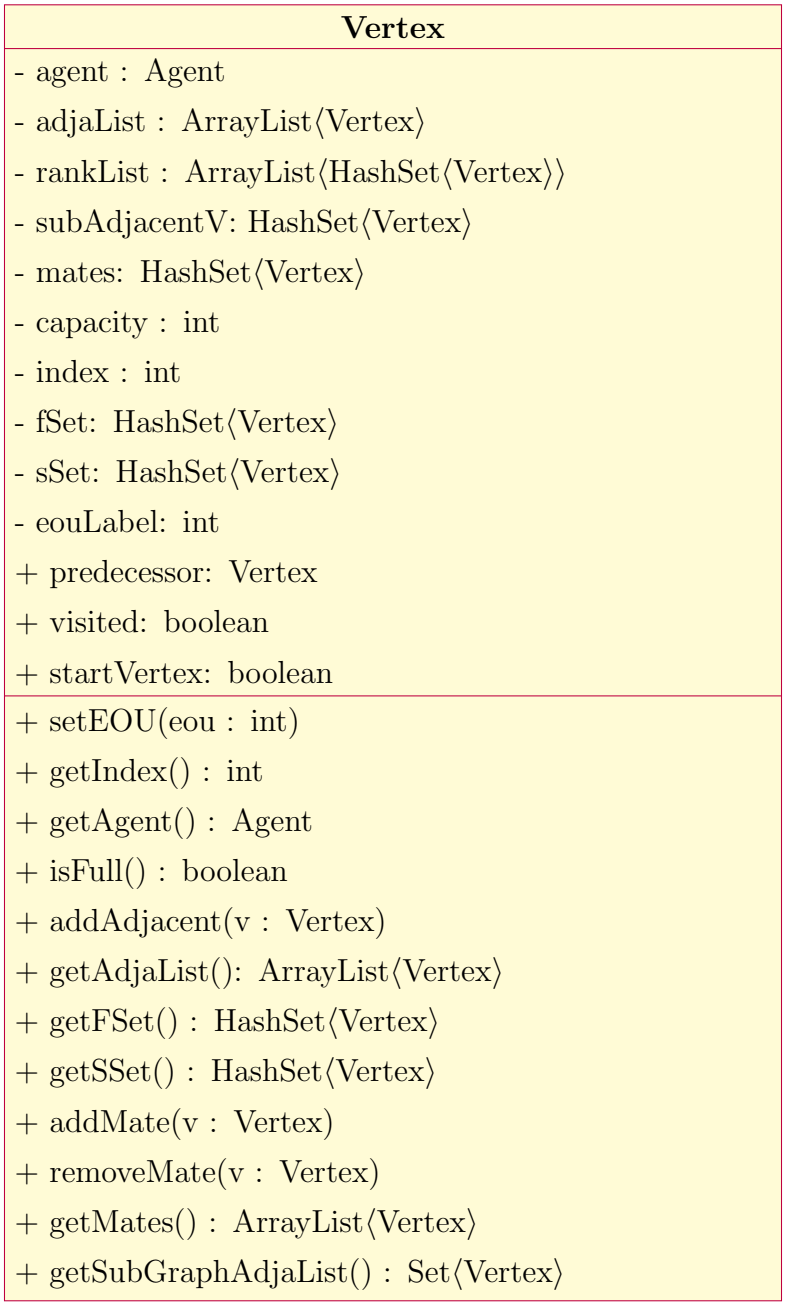

Figure D.3: Class diagram of Vertex class

Appendix E

Javadoc for *Impl.Model.AgentImpl*

The following table shows the methods provided by *Impl.Model.AgentImpl* class.

Methods provided by *impl.Model.AgentImpl.java*

This information was extracted from the Javadoc written by Aleš Remta.

Return Type	**Method**
void	**addLink**(api.model.Agent a) Add a link to the list (for instance one lecturer can offer several projects).
void	**attachPreferenceList**(api.model.PreferenceList pref) Add an initialised preference list to the agent.
api.model.Agent	**boundTo**() Returns the agent this agent is permanently linked to (for instance one project is linked to one lecturer).
api.model.Agent	**breakUp**(api.model.Agent a) Break the engagement of this agent with the agent in the argument.
api.model.Agent	**breakUpFirst**() Break the engagement with the most prefered agent in the engagements list.
api.model.Agent	**breakUpLast**() Break the engagement with the least prefered agent in the engagements list.
int	**compare**(api.model.Agent a, api.model.Agent b) Compares the ranks of the two specified agents to see which is more preferred by this agent.
int	**compareTo**(api.model.Agent a) Method defined by the Comparable interface.
void	**deleteAllHigherPositioned**(api.model.Agent a) Removes specified agent from this agents preference list.
void	**deleteAllHigherRanked**(api.model.Agent a) Removes all agents from this agent's prefrence list that have a higher rank than the specified agent (that are strictly prefered).
void	**deleteAllLowerPositioned**(api.model.Agent a) Removes all agents from this agent's prefrence list that are positioned below the specified agent (disregarding ties).
void	**deleteAllLowerRanked**(api.model.Agent a) Returns the agent this agent is permanently linked to (for instance one project is linked to one lecturer).
void	**deleteFromPrefList**(api.model.Agent a, boolean removeBack) Removes specified agent from this agents preference list.
void	**engageTo**(api.model.Agent a) Set up an engagement between this agent and the agent in the argument.
api.model.Agent	**engageToIfPreffered**(api.model.Agent a) Set up an engagement between this agent and the agent in the argument if the agent in the agrument is a prefferable choice for this agent compared to the worse engagement currently held.
api.model.Agent	**engageToIfPrefferedOrIndifferent**(api.model.Agent a) Operation not supported

api.model.Agent[]	**getAllLinks**() Returns an array of agents that are linked logically to this agent (for instance one lecturer can offer several projects).
api.model.Agent[]	**getEngagements**() Returns an array of agents that are engaged to this agent.
api.model.Agent	**getFirst**() Returns a reference to the agent in the first position in this agents preference list (the most preffered agent) or null, if the preference list is empty.
java.lang.String	**getID**() Returns the unique identifier of this agent.
int	**getIndex**() Returns the index of agent's position in the set.
api.model.Agent	**getLast**() Returns a reference to the agent in the last position in this agents preference list (the least preffered agent) or null, if the preference list is empty.
api.model.Agent	**getNext**() Returns the next agent in this agent's preference list or null if the end of the preference list has been reached.
api.model.Agent	**getNextRank**() Returns the agent in the first position of the next indifferent group of agents in this agents preference list or null if no lower preference group in the preference list.
int	**getPositionOf**(api.model.Agent a) Returns the position of specified agent in this agents preference list or -1 if specified agent not in the preference list.
AgentImpl[]	**getPreferenceListsAsArray**()
api.model.Agent	**getPrevious**() Returns the agent in the first position of the next indifferent group of agents in this agents preference list or null if no lower preference group in the preference list.
api.model.Agent	**getPreviousRank**() Returns the agent in the first position of the previous indifferent group of agents in this agents preference list or null if no higher preference group in the preference list.
int	**getRankOf**(api.model.Agent a) Returns the rank of specified agent in this agents preference list or -1 if specified agent not in the preference list.
int	**getRemainingCapacity**() Returns the number of possible engagements left.
int	**getTotalCapacity**() Returns maximum number of engagements for the agent.
boolean	**isEngaged**() s true if the agent is engaged to at least one partner, False otherwise.
boolean	**isEngagedTo**(api.model.Agent a) Determine whether this agent is engaged to the agent specified in the argument.
boolean	**isFull**() Returns true if the capacity of engegements for this agent has been reached,

	false otherwise.
boolean	**isTiedNext**() Indicates whether the current member in the prefernce list is tied (indifferent) with the next one.
boolean	**isTiedPrev**() Indicates whether the current member in the prefernce list is tied (indifferent) with the previous one.
int	**numEngagements**() Returns an array of agents that are engaged to this agent.
int	**prefListLenght**() Returns the length of this agents preference list.
void	**reset**() Reset all engagements and preference lists.
void	**resetCursor**() Added by @augustine to reset the currentElement back to null
void	**resetCursor**(api.model.Agent a)
void	**setBond**(api.model.Agent a) Creates a permanent link between this object and the object specified in the argument (for instance one project is linked to one lecturer).
void	**setRankOf**(api.model.Agent a, int rank)

Appendix F

User Manual for the matching algorithm toolkit

This appendix gives a brief introduction to the matching algorithm toolkit, including the GUI and the input/output format. We will focus on the part that concerns House Allocation problems. The standard steps to use the toolkit to solve House Allocation problem instances are as follow:

(Note: the toolkit should be started from *front_end.gui.GUIMain.java*.)

1 Choose the problem type (Figure F.1).

2 Choose input type (Figure F.2).

2.1 If Generate random instances, provide parameters (Figure F.3); And enter number of instances (Figure F.4).

2.2 If use instance from file, choose file (Figure F.5).

3 Select algorithm(s) (Figure F.6).

4 Run algorithm(s) (Figure F.7).

5 View results (Figure F.8 & F.9).

When a single problem instance is used, the detail information of the matching is shown,see Figure F.8. When multiple instances are generated and used, only the summary statistics

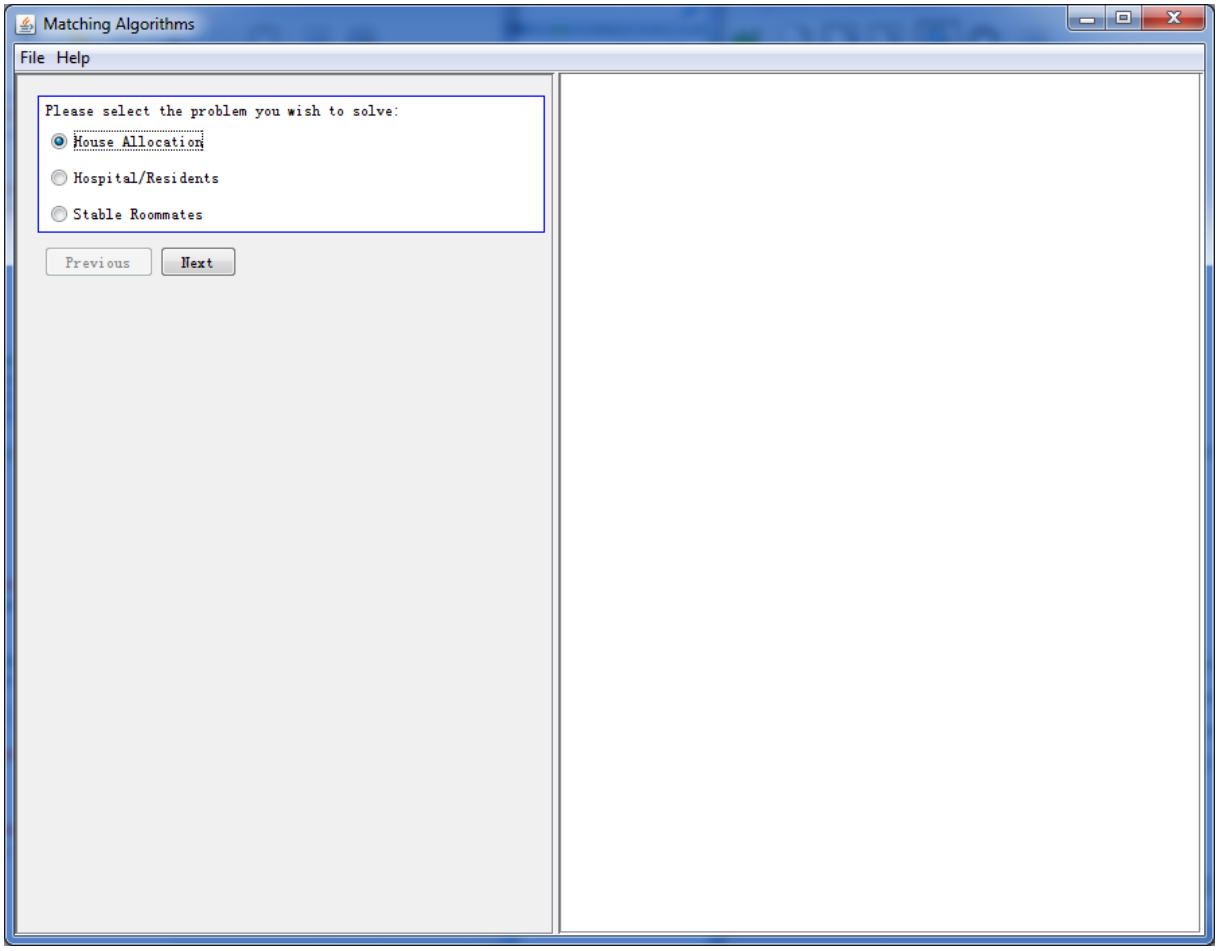

Figure F.1: Select problem type

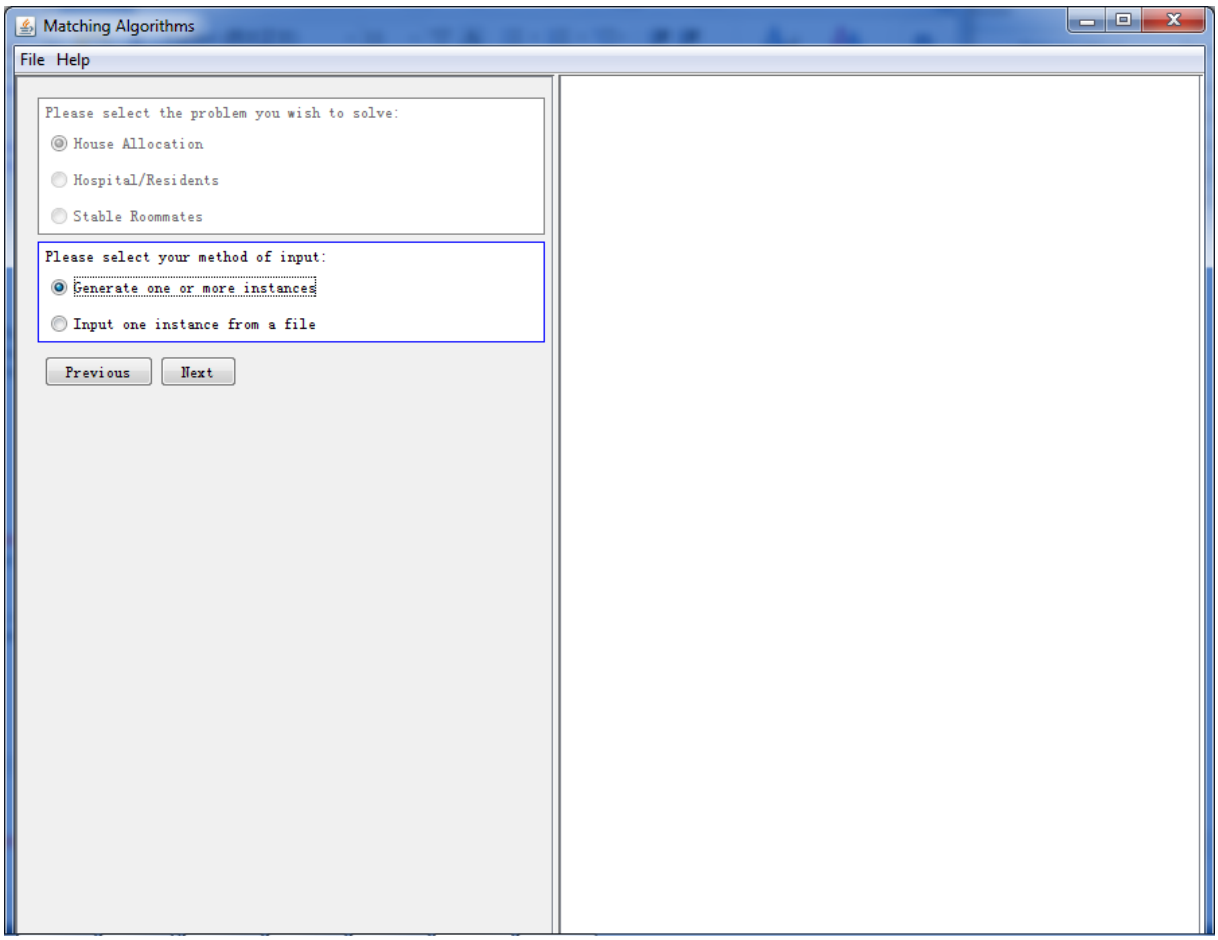

Figure F.2: Select input type

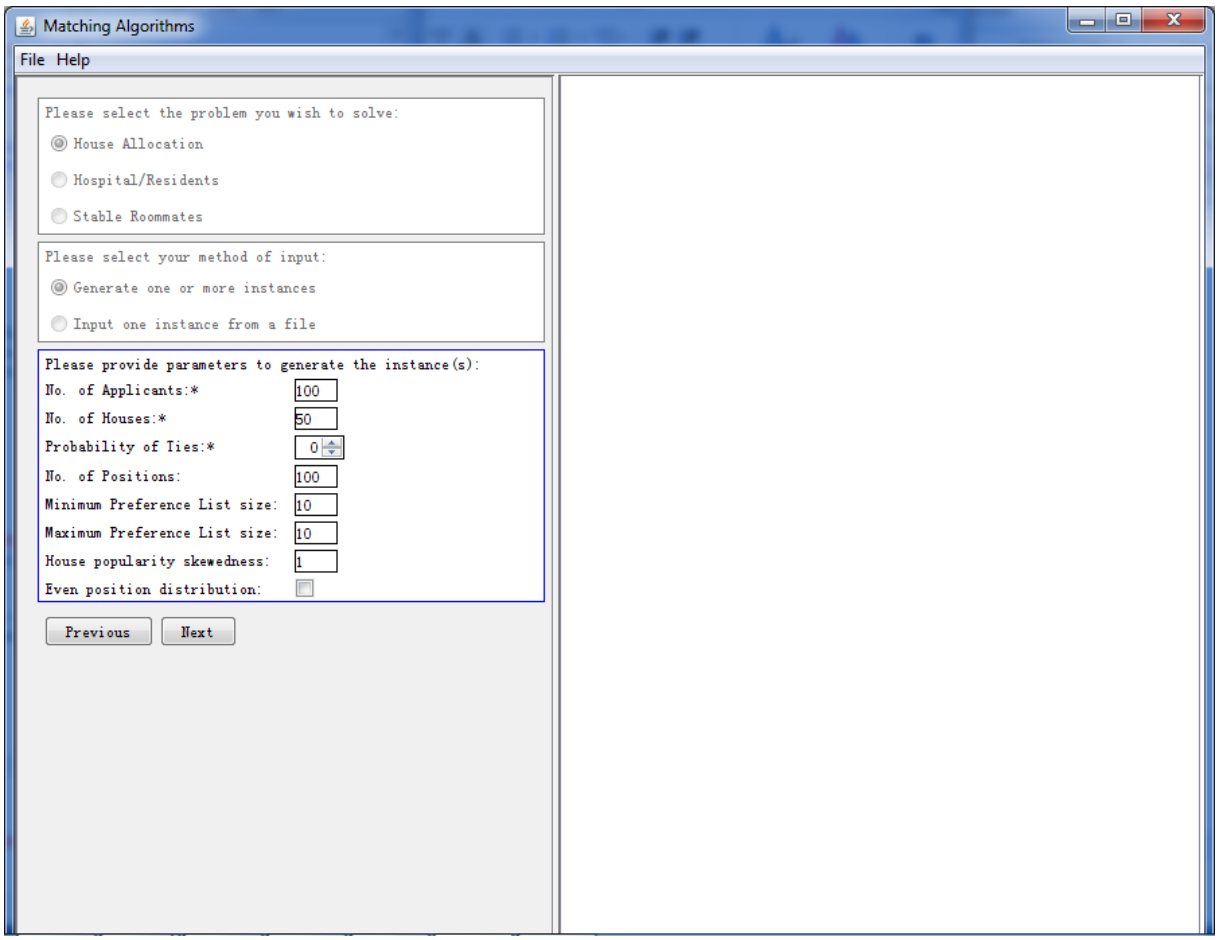

Figure F.3: Enter parameters

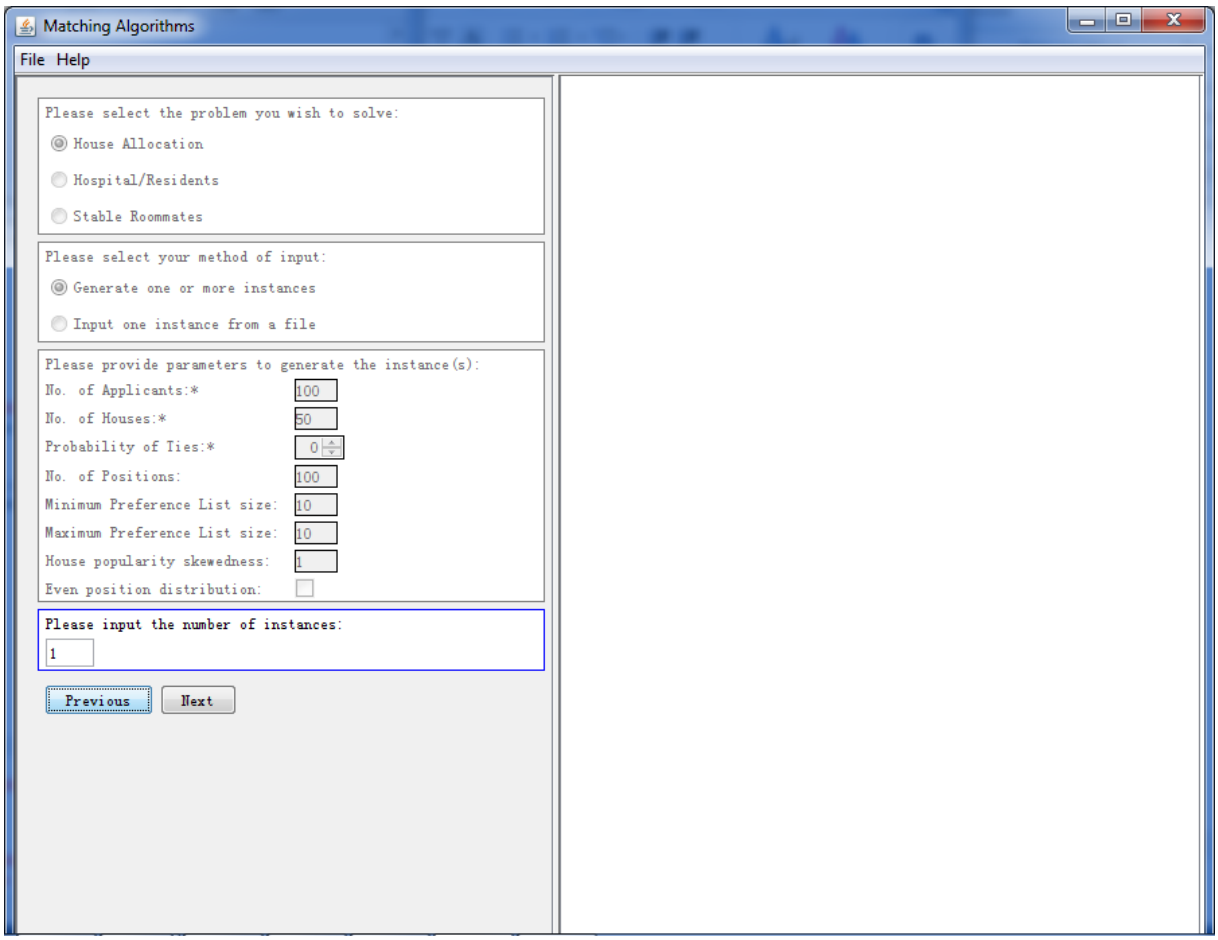

Figure F.4: Number of instances

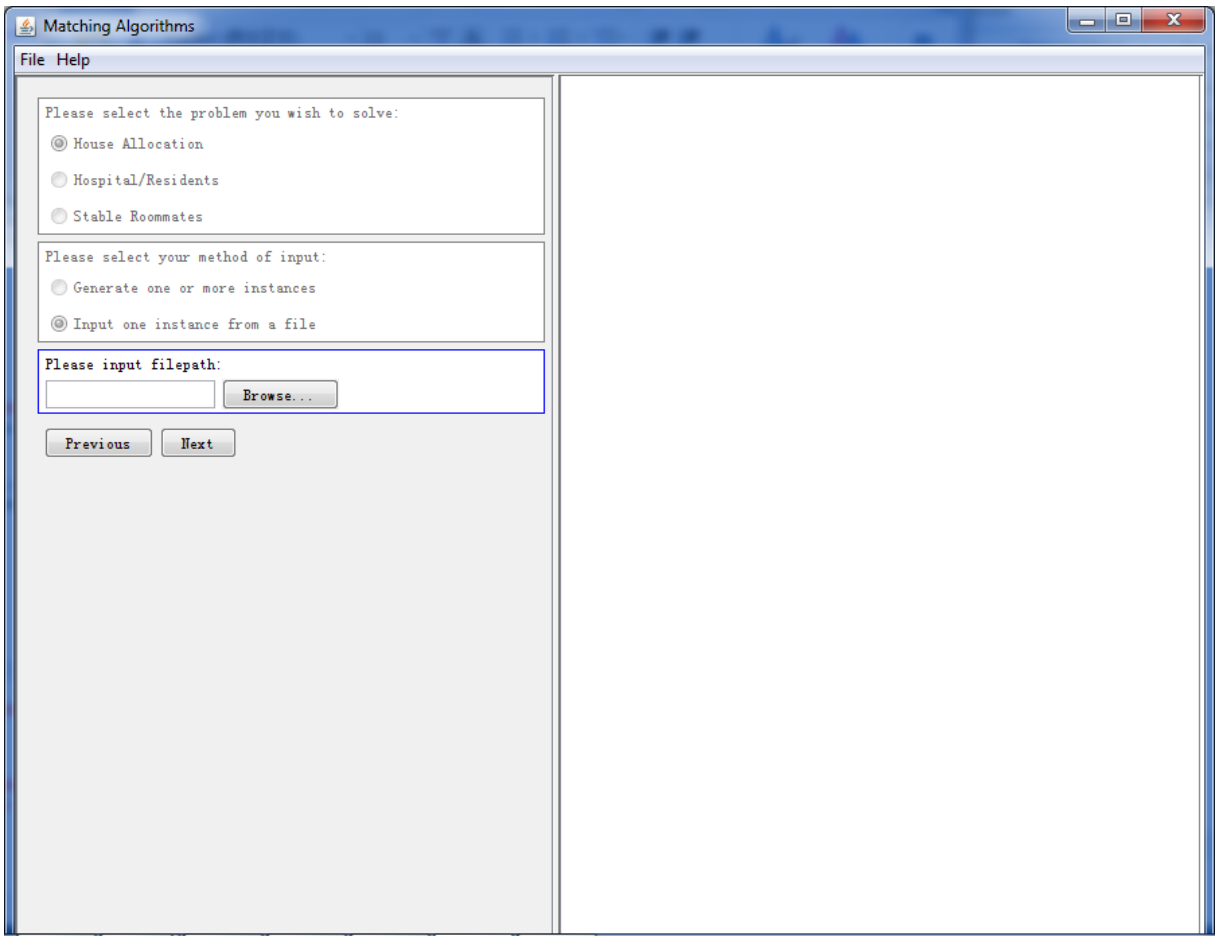

Figure F.5: Choose input file

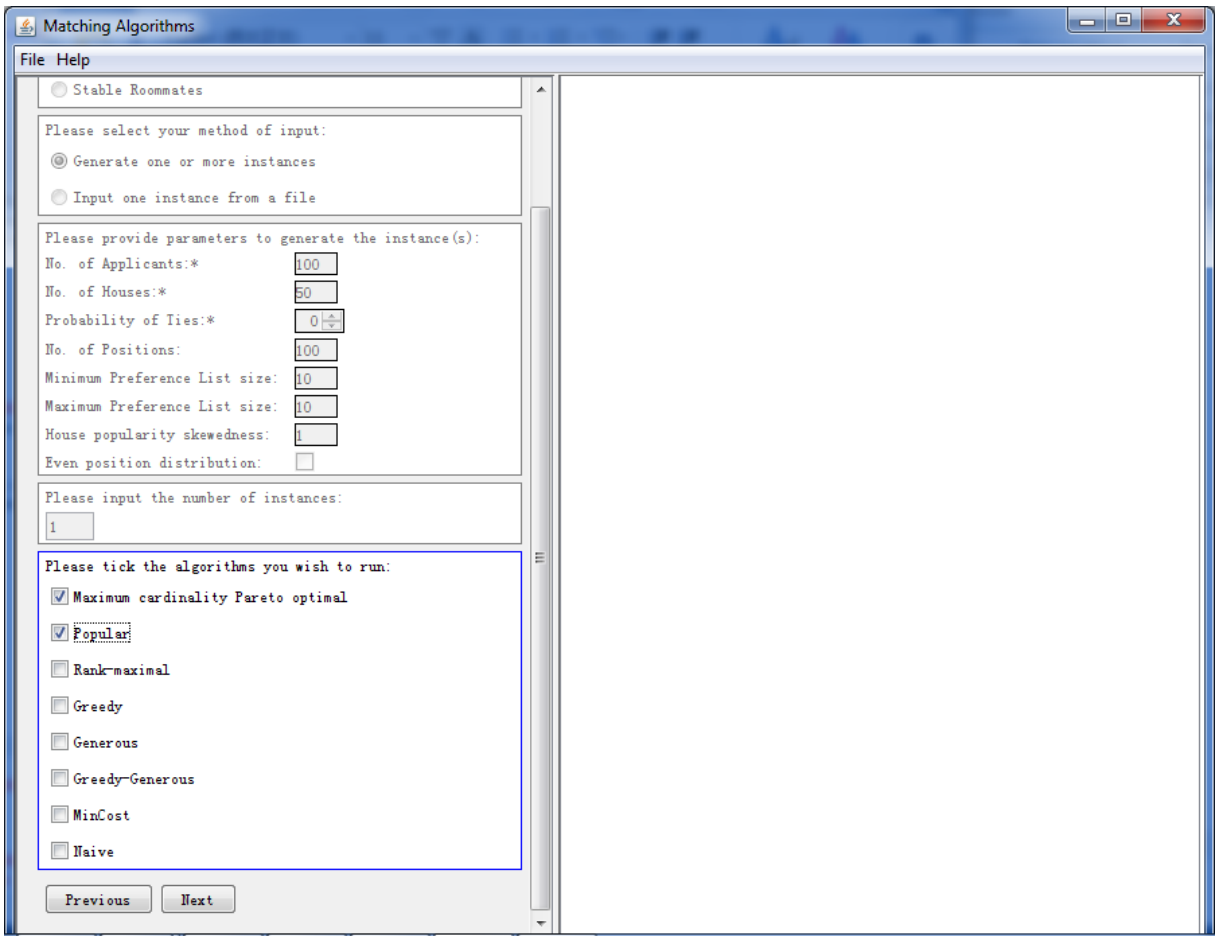

Figure F.6: Select algorithm(s)

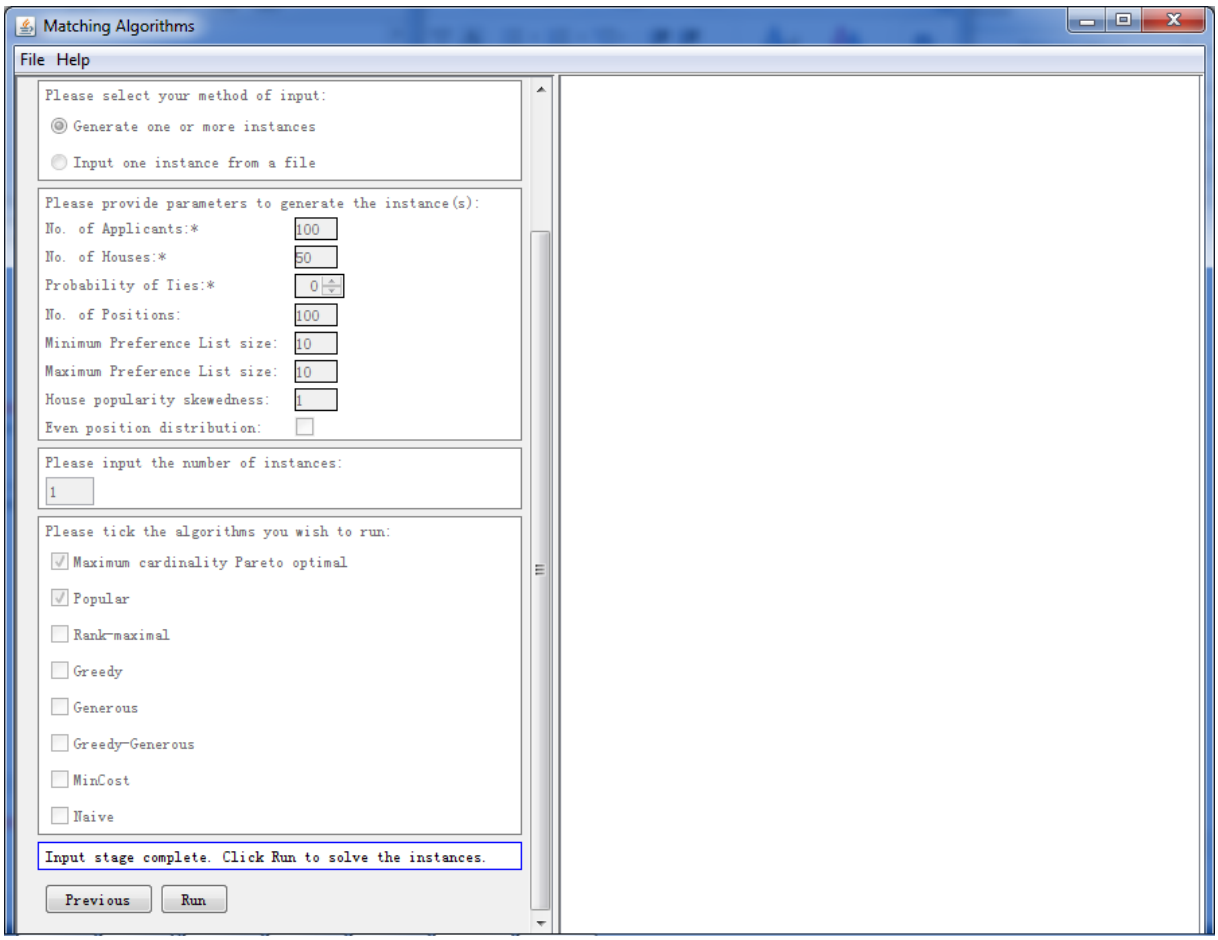

Figure F.7: Run algorithm(s)

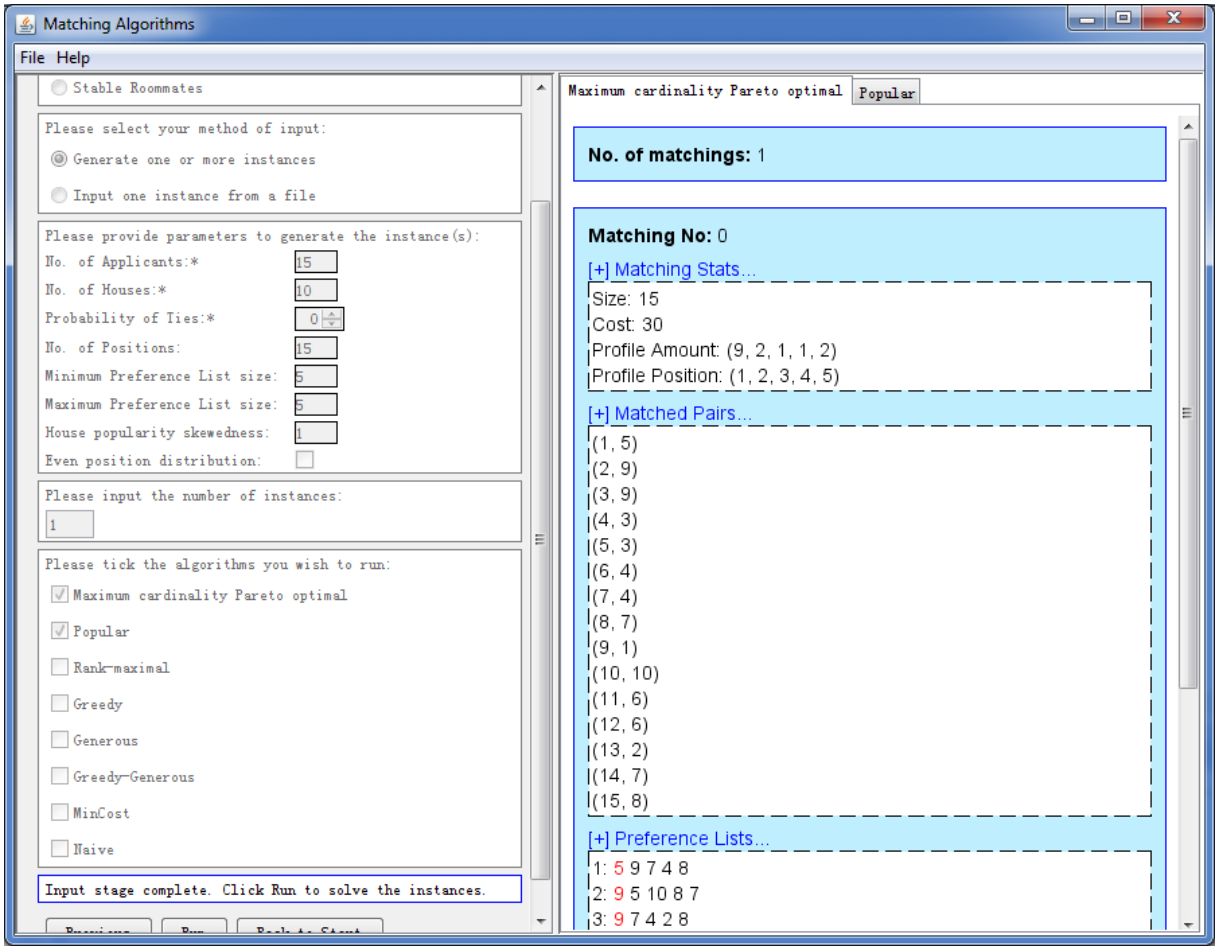

Figure F.8: Results (single instance)

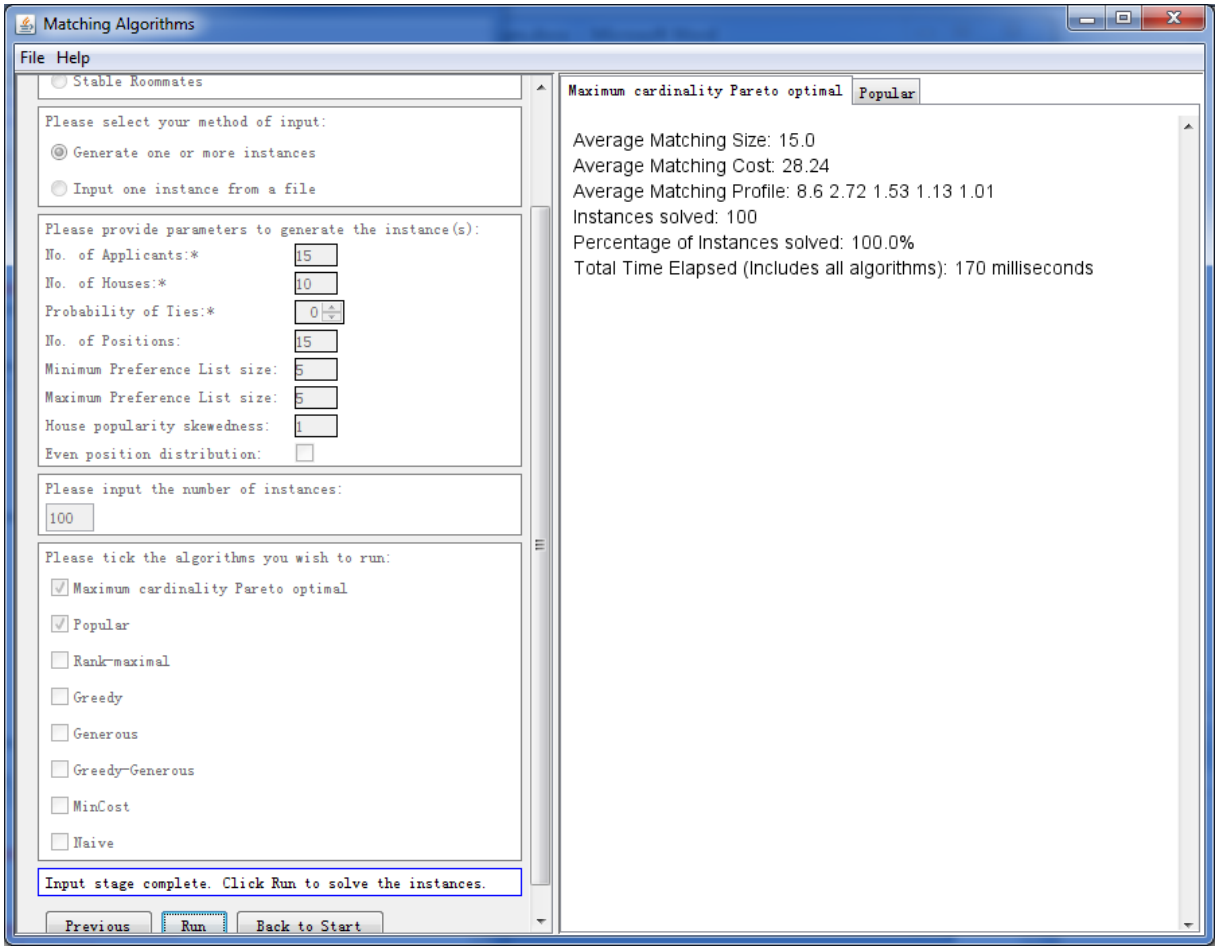

Figure F.9: Results (multiple instances)

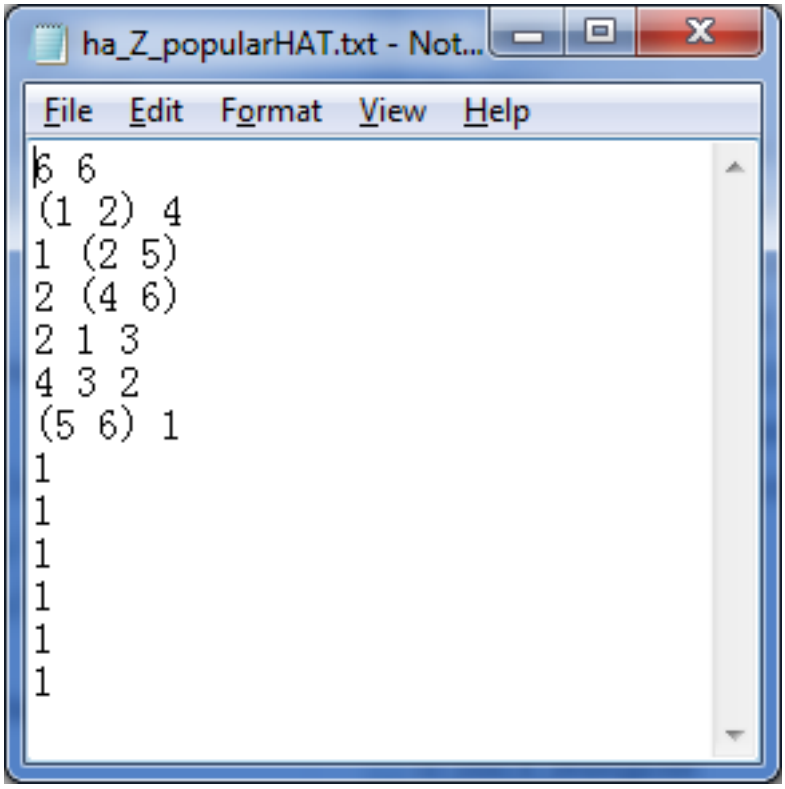

Figure F.10: The format of an input text file for House Allocation problem

are displayed (Figure F.9). This is the only output format available in the toolkit at the moment.

An example of text file input is given in Figure F.10. The two numbers in the first line are the number of applicants and the number of houses. In this instance, there are 6 applicants and 6 houses. The preference lists are displayed starting from the second line, with each line representing the preference list of an applicant, starting from a_1. In this case the preference lists are placed from Line 2 to Line 7 because there are 6 applicants. Also tied houses are shown by brackets. Following the preference lists are the capacities of houses. In the case every house is of capacity 1.

Bibliography

[1] Atila Abdulkadiroğlu and Tayfun Sönmez. Random serial dictatorship and the core from random endowments in house allocation problems. *Econometrica*, 66(3):689–701, 1998.

[2] David J Abraham, Katarína Cechlárová, David F Manlove, and Kurt Mehlhorn. Pareto optimality in house allocation problems. In *Algorithms and Computation*, pages 1163–1175. Springer, 2005.

[3] David J Abraham, Robert W Irving, Telikepalli Kavitha, and Kurt Mehlhorn. Popular matchings. *SIAM Journal on Computing*, 37(4):1030–1045, 2007.

[4] John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on computing*, 2(4):225–231, 1973.

[5] Aanund Hylland and Richard Zeckhauser. The efficient allocation of individuals to positions. *The Journal of Political Economy*, pages 293–314, 1979.

[6] Robert W Irving, Telikepalli Kavitha, Kurt Mehlhorn, Dimitrios Michail, and Katarzyna Paluch. Rank-maximal matchings. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 68–75. Society for Industrial and Applied Mathematics, 2004.

[7] David Manlove. *Algorithmics of matching under preferences*. World Scientific Publishing, 2013.

[8] David F Manlove and Colin TS Sng. Popular matchings in the capacitated house allocation problem. In *Algorithms–ESA 2006*, pages 492–503. Springer, 2006.

[9] Dimitrios Michail. Rank-maximal matching.

[10] MIT. How we assign housing, March 2013.

[11] MD Plummer and L Lovász. *Matching theory*. Access Online via Elsevier, 1986.

[12] Aleš Remta. A java api for matching problems. Master's thesis, University of Glasgow, 2010.

[13] Alvin E Roth. Incentive compatibility in a market with indivisible goods. *Economics letters*, 9(2):127–132, 1982.

[14] Alvin E Roth and Andrew Postlewaite. Weak versus strong domination in a market with indivisible goods. *Journal of Mathematical Economics*, 4(2):131–137, 1977.

[15] Lloyd Shapley and Herbert Scarf. On cores and indivisibility. *Journal of mathematical economics*, 1(1):23–37, 1974.

[16] Colin Thiam Soon Sng. *Efficient Algorithms for Bipartite Matching Problems with Preferences*. PhD thesis, Faculty of Information and Mathematical Sciences at the University of Glasgow, 2008.

[17] Philip Yuile. Adding to a library of matching algorithms, 2011. Level 4 project.