MatchMaker: A DSL for Game-Theoretic Matching

Prashant Kumar and Martin Erwig[0000−0002−7471−4554]

Oregon State University, Corvallis OR 97330, USA
{kumarpra,erwig}@oregonstate.edu

Abstract. Existing tools for solving game-theoretic matching problems are limited in their expressiveness and can be difficult to use. In this paper, we introduce MatchMaker, a Haskell-based domain-specific embedded language (DSEL), which supports the direct, high-level representation of matching problems. Haskell's type system, particularly the use of multi-parameter type classes, facilitates the definition of a highly general interface to matching problems, which can be rapidly instantiated for a wide variety of different matching applications. Additionally, as a novel contribution, MatchMaker provides combinators for dynamically updating and modifying problem representations, as well as for analyzing matching results.

1 Introduction

A large class of problems are instances of matching problems. Examples include the assignment of children to different schools, students to universities and campus housing, doctors to hospitals, kidney transplant patients to donors, and many others. In each of these problems, the participants in the matching process typically have *preferences* over the entities they are matched to, and the task is to find a matching that is, in some sense, optimal with respect to these preferences. The importance of matching is also highlighted by the fact that the 2012 Nobel Prize in Economics was awarded to Lloyd S. Shapley and Alvin E. Roth for their work on stable matching problems.

Despite its apparent usefulness, the actual software support for expressing and solving matching problems is surprisingly limited in a number of ways. For example, the currently available software tools for solving matching problems are limited in expressiveness and often difficult to use. Almost all the available matching libraries use strings to encode the matching problem, which affects readability and maintainability of the encoded problems. Employing untyped representations limits the options for checking the validity of the encoding and producing meaningful error messages. As we will demonstrate, MatchMaker leverages Haskell's rich type system and its type class system to facilitate high-level representations of matching problems that are readable, easily modifiable, and provide good error checking.

Arthur	Sunny	Joseph	Latha	Darrius
City	City	City	Mercy	City
	Mercy	General	City	Mercy
		Mercy	General	General

(a) Applicants' hospital preferences

Mercy	City	General
Darrius	Darrius	Darrius
Joseph	Arthur	Arthur
	Sunny	Joseph
	Latha	Latha
	Joseph	

(b) Hospitals' ranking of applicants

Fig. 1: Matching hospitals with applicants: a two-sided stable matching example.

MatchMaker already implements algorithms for a large class of matching problems. More specifically, we implement *bipartite stable matching with two-sided preferences*, *bipartite stable matching with one-sided preferences*, and *same-set matching problems with one-sided preferences*. Together, these represent the most important and widely applicable matching problems [16, 11, 7].

Our DSL makes the following main contributions. It:

- offers a high-level, type-safe, extensible representation for matching problems.
- defines a scalable mechanism for describing preferences based on function definitions and abstract criteria.
- provides functions to analyze and compare the results of various matchings.
- is easily extensible to represent new matching problems.

The remainder of this paper is structured as follows. In Section 2, we introduce stable bipartite matching problems with two-sided preferences and encode them in MatchMaker with explicit preferences. In Section 3, we illustrate how to represent preferences implicitly using Haskell's abstract data types and functions. In Section 4, we introduce combinators to update the existing matching representations plus functions for comparing the results of two matchings. In Section 6, we compare MatchMaker to other tools for matching. Finally, in Section 7 we provide conclusions.

2 Bipartite Stable Matching With Two-Sided Preferences

Consider the task of assigning medical residency applicants to hospitals. Figure 1 shows an example taken from the National Resident Matching Program's (NRMP) website [12] in which five applicants apply to the three hospitals. Hospitals and applicants list their preferences, and each hospital can accept at most two applicants. The stable matching algorithm (also called *delayed acceptance algorithm* [14, 4]) produces a match with the following two characteristics:

(1) Each applicant is assigned to only one hospital, and no hospital is assigned more applicants than its quota.

(2) The resulting match is *stable*.

This stability condition is described in the delayed acceptance algorithm as the match not having a *blocking pair*, which is a pair of a hospital and an applicant currently assigned to different partners but who prefer each other more than their current assignment. The presence of such pairs undermines the effectiveness of the matching process, as these pairs can make private arrangements, leaving behind their partners assigned by the matching algorithm. Roth and Sotomayor show an example of an unstable matching mechanism for matching doctors to hospitals in Birmingham and Newcastle used in the 1960s and 1970s [16, Chapter 5]. The instability of the outcome led to doctors and hospitals entering private negotiations outside the matching process, which left many doctors without a position and many hospitals without a resident. This culminated in the abandonment of the mechanism. Gale and Shapley [4] showed that a special property of bipartite matching markets is that stable matchings always exist.

Let us try to match hospitals with applicants, taking into account their preferences and quotas. Consider the preference list of Darrius. He prefers City Hospital the most, and City Hospital also ranks him the highest amongst the candidates. It is easy to deduce that Darrius will end up at City Hospital. Now, if we look at the preference list of Sunny, we see that she considers only City and Mercy for her residency. However, Sunny is not included in Mercy Hospital's rankings, so she cannot be assigned there. Her first option, City Hospital, does rank her third. However, notice that the two people ranked above her, Darrius and Arthur, have listed City as their first choice. If they are assigned the two positions, then Sunny is left without an offer, as Mercy is the only hospital in her preference set that also ranks her.

Could we have accommodated Darrius at Mercy Hospital, leaving room for Sunny at City Hospital? Although this does lead to Sunny getting accommodated at City Hospital, it results in instability in the matching process due to the formation of a blocking pair between Darrius and City Hospital. Darrius still prefers City Hospital to Mercy Hospital, and City Hospital still prefers Darrius to Sunny. In other words, they both benefit from forming their own match, leaving behind their assigned matches.

In this section, we demonstrate how to represent this example in MatchMaker and generate a stable matching. To motivate the different design choices, it is instructive to look at the formal model of stable matching.

2.1 Modeling Stable Matching

A two-sided stable matching problem between applicants and hospitals is a 6 tuple $(A, H, P_A, P_H, Q_A, Q_H)$ where $A = \{a_1, \ldots, a_m\}$ and $H = \{h_1, \ldots, h_n\}$ represent the finite disjoint sets of applicants and hospitals, respectively [16]. The preference of each applicant $a \in A$ is represented by an ordered list of preferences $P(a)$ on set H. Similarly, the preference of each hospital $h \in H$ is represented by an ordered list of preferences $P(h)$ on set A. The set of all preference lists is captured by the functions $P_A : A \to H^*$ and $P_H : H \to A^*$.

```
import qualified Data.Map as M

type Capacity = Int

forall :: Capacity -> a -> Capacity
forall c _ = c

class (Bounded a,Enum a,Ord a) => Set a where
    members :: [a]
    members = enumFromTo minBound maxBound

    quota :: a -> Capacity
    quota = forall 1

data Rec b c    = Rec  {unRec :: M.Map b c}
data Info a b c = Info {unInfo :: M.Map a (Rec b c)}

data Rank = Rank {unRank :: Int}

class Preference a b c | a b -> c where
  gather :: Info a b c

type Ord2 a b = (Ord a, Ord b)
type Preference2 a b c d = (Preference a b c,Preference b a d)

info    :: Ord2 a b => [(a,[(b,c)])] -> Info a b c
choices :: Ord2 a b => [(a,[b])] -> Info a b Rank

data Match a b  = Match  {unMatch:: [(a,[b], Capacity)]}

ranks :: (Set2 a b,Norm c,Weights a) => Info a b c -> Match a b

twoWay :: (Preference2 a b c d,Set2 a b,Norm2 c d) => Match a b

twoWayWithCapacity :: (Preference2 a b c d,Set2 a b,Norm2 c d) =>  Match a b

twoWayWithPref :: (Preference2 a b c d,Set2 a b,Norm2 c d) => Info a b c ->
                  Info b a d -> Match a b
```

Fig. 2: Definitions for encoding and storing preferences in MatchMaker.

Each hospital h is also assigned a positive integer $Q(h)$, also called its *quota*, that represents the maximum number of applicants it could admit. Similarly, each applicant is also assigned a quota. This information is collected in the two functions $Q_A : A \to \mathbb{N}$ and $Q_H : H \to \mathbb{N}$. For the applicant-hospital matching problem, it is obvious that applicants have a quota of 1, since they can work at only one hospital. However, in other examples of matching problems with two-sided preferences, both the sets can have quotas greater than 1.

A *matching* is a relation $\mu \subseteq A \times H$ that satisfies the following two conditions: (1) $\forall a \in A : |\mu(a)| \leq Q_A(a)$ and $\forall h \in H : |\mu(h)| \leq Q_H(h)$ and (2) $(a, h) \in \mu \Rightarrow a \in P(h) \wedge h \in P(a)$. The first condition ensures the matching satisfies the quota restrictions, and the second condition ensures consistency. In our current example that means that a hospital is in an applicant's match only if they are in each other's preference list.

The formal model guides the design of our DSL, which we demonstrate with the help of our example next.

2.2 DSL Representation of Matching Problems

The first step in encoding our example is to represent the two sets to be matched as Haskell data types.

```
data Applicant = Arthur | Sunny | Joseph | Latha | Darrius
data Hospital  = City | Mercy | General
```

The definitions and type signatures of various data types, type classes, and functions used in this section are summarized in Figure 2.

To store the preferences of the applicants and hospitals we use two mappings: `Rec` and `Info`, which is a collection of `Rec`s, represented as a mapping[1]. Specifically, `Info Applicant Hospital Rank` maps every applicant to a record `Rec Hospital Rank`, which maps hospitals to their ranks as specified by the applicant. Similarly, the ranking of applicants by hospitals is represented in `Info Hospital Applicant Rank`, where the individual preferences of each hospital are recorded in the mapping `Rec Hospital Rank`.

We define `Functor` instances of the `Rec` and `Info` types in Figure 4. We use these instances to define their corresponding `zap` functions (also in Figure 4), which are useful for creating combinators like `zipInfo` that we later use in the paper.[2] It can be thought of as a generalization of the `zip` function, where instead of just combining two functorial structures (like lists), we can combine any number of functorial structures using corresponding functions.

The multi-parameter type class `Preference a b c` provides an interface for specifying preferences. An interesting aspect of the class definition is the functional dependency specification, which signifies that types `a` and `b` uniquely determine type `c`. The `gather` function of the type class captures the preference of elements of set `a` for elements of set `b` using a type `c` and stores it in the mapping `Info a b c`. The smart constructors `info` and `choices` are used to construct the preference mappings from list of tuples.[3]

To keep the number of class constraints manageable in Figure 2, we use a Haskell language extension called `ConstraintKinds`, which allows us to define class constraints more succinctly using type synonyms. For instance, instead of using `(Ord a, Ord b)` as the class constraint, we can use its type synonym `Ord2 a b`. We have similar definitions for `Norm2`, `Set2` and `Preference2 a b c d`.

Our current example requires two-sided specification of preferences. This entails two instance definitions of the `Preference` class, one for hospitals and one for applicants. However, before presenting those definitions, we discuss a particular design choice for the type class. One question is whether we should have simplified the definitions of the `Info` mappings and consequently the `Preference` class by removing their last type argument and hard-coding the `Rank` in the definitions instead. This would mean that the rank of an item is specified by the position of that item in a list. While this does simplify the design, the constraint

[1] The mappings are represented by the `Data.Map.Map` data type from the standard `containers` package of Haskell

[2] Thanks to the one of the reviewers for suggesting the use of `zap`.

[3] We mostly show only the type signatures and present implementations only when they contribute to a better understanding. For the complete code, see `https://github.com/prashant007/MatchMaker`.

to relate the items being matched in just one way also limits the expressivity of the domain. The advantages of our design choice become apparent in Section 3 where we instantiate the third argument of `Info` and `Preference` with richer types than `Rank` that allow agents to implicitly rank other agents, which eases the cognitive burden and effort in defining preference lists.

The ranked preference lists of applicants for hospitals can be specified with an appropriate instance of `Preference` type class using the `choices` function as shown below. The infix operation `-->` is simply syntactic sugar for building pairs. In this representation, rankings are based on positions. For example, the fact that City precedes Mercy in the preference list of Sunny means that she prefers City to Mercy.

```
instance Preference Applicant Hospital Rank where
  gather = choices [Arthur  --> [City],
                    Sunny   --> [City,Mercy],
                    Joseph  --> [City,General,Mercy],
                    Latha   --> [Mercy,City,General],
                    Darrius --> [City,Mercy,General]]
```

The ranked preference lists of applicants for hospitals can be similarly encoded.

```
instance Preference Hospital Applicant Rank where
  gather = choices [Mercy   --> [Darrius,Joseph],
                    City    --> [Darrius,Arthur,Sunny,Latha,Joseph],
                    General --> [Darrius,Arthur,Joseph,Latha]]
```

Finally, to encode the quota information, we define a type class called `Set` with `quota` as a member function as shown in Figure 2. We also define a function `members` that can list all the elements of a set. The function `forall` is used to assign the same quota to every member of the set to be matched.

The instances of `Set` for the `Applicant` and the `Hospital` types are shown below, where each hospital is assigned a quota of 2 and each applicant is assigned a default quota of 1.

```
instance Set Applicant
instance Set Hospital where quota = forall 2
```

2.3 Generating Stable Matchings

In general, a matching problem can have multiple stable matchings. However, two of them are especially significant. For our problem, these are the *hospital-optimal stable match* and the *applicant-optimal stable match*. (Sometimes the adjective "stable" is omitted for brevity.) In a hospital-optimal match, hospitals do as well as they possibly can. While not intended, the structure of the matching problem entails that a stable match where hospitals perform their best is also a stable match where applicants perform their worst [16, Chapter 2, Corollary 2.14]. Similarly, in an applicant-optimal match, applicants perform their best and hospitals their worst. Interestingly, the NRMP program was shown to be hospital optimal [21] before it was changed to be applicant optimal in 1997 [15].

A stable match can be computed with the function `twoWayWithPref`, which takes two preference encodings of the `Info` type and yields a value of type `Match a b` that stores all the elements of set `b` matched to an element of set `a`. The

overloaded value `twoWay` triggers the computation by inferring the `Info` arguments from its type annotation. For example, the annotation `Match Applicant Hospital` generates the applicant-optimal matching.

```
> twoWay :: Match Applicant Hospital
{Sunny --> [], Darrius --> [City], Latha --> [General], Joseph --> [General], Arthur --> [City]}
```

Similarly, `Match Hospital Applicant` generates a hospital-optimal matching.

```
> twoWay :: Match Hospital Applicant
{City --> [Arthur,Darrius], Mercy --> [], General --> [Latha,Joseph]}
```

We can observe that the two matchings are the same. However, this need not always be the case. The DSL also provides a function `twoWayWithCapacity` to find the remaining quotas in a matching. The next example shows that General and City have exhausted their quotas of applicants, whereas Mercy's quota of 2 is untouched, since no residents have been assigned to it.

```
> twoWayWithCapacity :: Match Hospital Applicant
{City --> [Arthur,Darrius] : 0, Mercy --> [] : 2, General --> [Latha,Joseph] : 0}
```

2.4 The Role of Type Classes in the DSL Design

After examining a problem encoding in our DSL, we can now discuss an important aspect of the DSL's design: the use of type classes. In our opinion, the type classes enhance the clarity of the encoding that is generated as well as guide users during the encoding process. Using the `twoWayWithPref` function, which is the function for matching we would use in the absence of a type class, would require users to encode hospitals' and applicants' preferences without guidance from the DSL, making it more challenging. Our DSL explicitly sets user expectations, as shown below, with the first instance gathering hospitals' choices for applicants using ranks, and the second doing the reverse.

```
instance Hospital Applicant Rank where
    gather = ....

instance Applicant Hospital where
    gather = ....
```

Moreover, the declaration `twoWay :: Match Applicant Hospital` provides an easy way for users to specify that they want an applicant-optimal match. In contrast, without the type class users must call the `twoWayWithPref` function with the arguments `infoApplicant` and `infoHospital` and consider input order, which can yield different results. Finally, as we will see later, original preference encodings (`Info` values) can evolve over time. Having the initial preference list determined by an instance declaration (of the class `Preference`) improves clarity; otherwise, users would need to rely on naming conventions to identify starting preference lists.

3 Representational Ranking

The encoding of the NRMP example in the previous section is not ideal, particularly when the number of applicants or hospitals to be ranked becomes large. Instead of ranking through ordered lists, it's often more practical to use a function that computes ranks based on attributes of the elements being ranked. For example, a hospital might prefer to rank candidates using weighted criteria, such as MCAT scores, interview performance, prior experiences, and whether their previous degree is from their hospital. Each candidate's score can be generated using a formula, and the reciprocal of this score can be used to determine their rank. Different hospitals may assign various weights to these criteria, with some even omitting certain factors. MatchMaker facilitates this form of ranking. To this end, we define a data type `AInfo` for storing the relevant applicant data.

```
data AInfo = Appl {examScore      :: Double,
                   experience     :: Double,
                   interviewScore :: Double,
                   sameSchool     :: Bool }
```

Similarly, a candidate might prefer to specify the ranking of hospitals implicitly based on the livability of the city the hospital is in, reputation of the programs and their personal desire to attend a particular program. Again, these criteria are assigned appropriate weights. The applicants' model of hospital preferences is captured by the data type `HInfo`, defined as follows.

```
data HInfo = Hptl {hospitalRank      :: Rank,
                   cityLivability    :: Int,
                   desirabilityScore :: Double}
```

Next we need to express the information in a form that supports the computation of preference lists.

3.1 Normalization and Weighting of Criteria

To generate rankings, we normalize values of a representation type to numbers using the type class `Norm`. Figure 3 shows the definition of this type class as well as some of its instances. The primary purpose of this type class is to transform an element of type `a` into a number between 0 and 1. For straightforward types like `Rank` and `Bool`, we can create a direct instance of this type class. Note that the `Norm` instance for the `Rank` type, which represents relative preferences, conveys that a numerically lower rank corresponds to a higher preference, and vice versa.

In most cases, a constant is required for normalization. For instance, to normalize an exam score of 80, we need to know the maximum possible score. Assuming this to be 100, the score can be normalized as $\frac{80}{100} = 0.8$. To handle this, we introduce a new data type `BoundedVal` for managing normalization with a bound. The `outOf` function is utilized to create a `BoundedVal` value for normalization. We represent this normalization as `80 `outOf` 100`.[4]

[4] Thanks to one of the reviewers for recommending the use of the `BoundedVal` data type, which enables our `norm` function to be total.

```haskell
class Norm a where
    components :: a -> [Double]
    components _ = []

    norm :: a -> Double
    norm = sum . components

instance Norm Bool where
  norm x = if x then 1.0 else 0.0

instance Norm Rank where
    norm (Rank r) = 1/fromIntegral r

instance Norm Double where
    norm v = v

instance Norm Int where
    norm v = fromIntegral v


data Polarity = Pos | Neg

class Valence a where
    valence :: a -> Polarity
    valence _ = Pos

instance Valence Int
instance Valence Double
...
```

```haskell
data BoundedVal a = a `OutOf` Double

outOf :: (Norm a,Num a,Valence a) =>
         a -> Double -> Double
outOf x y = norm (x `OutOf` y)

instance Valence NDouble where
    valence (ND _) = Neg

instance (Valence a, Num a,Norm a) =>
         Norm (BoundedVal a) where
    components (x `OutOf` y)
        | valence x == Pos = [norm x/y]
        | otherwise        = [y/norm x]

class Weights a where
    weights :: a -> [Double]
    weights _ = [1.0]

class Weights a =>
    Preference a b c | a b -> c where
    gather :: Info a b c
```

Fig. 3: Support for Representational Rankings in MatchMaker.

Occasionally, an attribute may have negative valency, indicating that a lower value of the attribute is considered more favorable than a higher value. In Figure 3 we define a type class `Valence` along with a data type `Polarity`. Using this type class, we can specify the desired valency of a type. For negative valence double values, we define a data type `NDouble` and its corresponding normalization. Note that the positions of the numerator and denominator are switched compared to the `Norm` instance of the `Double` type.

The type class `Norm` also offers a `components` function, which provides a list of normalized values corresponding to the various arguments of a constructor of an abstract data type. As shown in the class definition, once we have defined `components` for a data type, the normalized values can be easily deduced from it.

With the help of `norm` and `outOf`, we can define the normalization for the applicant and hospital preference representations as follows.

```haskell
instance Norm AInfo where
    components (Appl e x i c) = [e `outOf` 800, x `outOf` 10, i `outOf` 10, norm c]

instance Norm HInfo where
    components (Hptl h c d) = [norm h, c `outOf` 10, d `outOf` 5]
```

However, before we compute preferences using the normalization of representation types, we need to address the situation where applicants or hospitals may weight criteria differently. To that end, MatchMaker provides a class `Weights`, shown in Figure 3, which can be used to assign different weight profiles for various criteria corresponding to different constructors of type `a`. This type class is then placed as a class constraint in the definition of the `Preference` type class (shown in Figure 2), which specifies that the first type argument of the `Preference` class

```
instance Functor (Rec b) where
  fmap f (Rec m) = Rec (fmap f m)

instance Functor (Info a b) where
  fmap f (Info m) = Info (fmap (fmap f) m)

zapRec :: Rec a (b -> c) -> Rec a b -> Rec a c
zapRec (Rec fMap) (Rec xMap) = Rec (M.intersectionWith ($) fMap xMap)

zapInfo :: (Ord a,Ord b) => Info a b (c -> d) -> Info a b c -> Info a b d
zapInfo (Info i1) (Info i2) = Info (M.intersectionWith zapRec i1 i2)

zipInfo  :: (Ord2 a b) => Info a b c -> Info a b d -> Info a b (c,d)
zipInfo i1 = zapInfo (fmap (,) i1)

zipInfo2 :: (Ord2 a b) => Info a b c -> Info a b d -> Info a b e -> Info a b (c,d,e)
zipInfo2 i1 i2 i3 = zapInfo (fmap (\x (y,z) -> (x,y,z)) i1) (zipInfo i2 i3)


completedWith  :: Ord a => (b -> c -> d) -> Info a b c -> Info a b d
completedWith2 :: Ord a => (b -> c -> d -> e) -> Info a b (c,d) -> Info a b e
```

Fig. 4: Combinators for combining `Info` values and generating them from profiles.

should also be a member of the `Weights` class. This allows us to generate rankings for various hospitals and applicants using different distributions of criteria weights.

The weight distributions of the criteria for various hospitals and applicants are specified as instances of the `Weight` class. We observe that Mercy assigns greater importance to exam and interview scores than to previous work experiences compared to other hospitals. Furthermore, unlike other hospitals, Mercy gives some weight to whether or not applicants have previously studied there.

```
instance Weights Hospital where
    weights Mercy = [0.3,0.3,0.3,0.1]
    weights _     = [0.2,0.2,0.6,0.0]
```

For applicants we assume that they all use the same weights for the various criteria.

```
instance Weights Applicant where
    weights = forall [0.2,0.2,0.6]
```

3.2 Representational Rankings in Action

Now we can derive a rank from preference representations. Specifically, we can replace the third argument of the `Preference` type class, `Rank`, with `AInfo` and `HInfo`, allowing us to record the preferences for hospitals and applicants, respectively. Before we look at the actual preference encodings of applicants, note that values of some criteria remain unchanged for different applicants. For example, rankings of the hospitals and the livability of the cities they are located in are not applicant dependent but intrinsic to the hospitals and cities themselves. We can exploit this fact to factor out this shared information, which can then be used by all applicants. The function `hProfile` constructs a hospital/city profile for each hospital as a partial `HInfo` value with fixed ranking and livability score

information but still unassigned desirability scores of type `DScore` (which is a type synonym for `Double`).

```
hProfile :: Hospital -> DScore -> HInfo
hProfile Mercy   = Hptl (Rank 2) 9
hProfile City    = Hptl (Rank 1) 10
hProfile General = Hptl (Rank 3) 8
```

Next, we represent the desirability scores of hospitals for the different applicants in the form of an `Info` value. Of course, it may be the case that applicants use different sources for getting the ranking and livability information, resulting in non-uniform rankings of hospitals and livability scores of cities. In such a case, we could have two additional `Info` values, one each for rank and livability, similar to what we have for the desirability scores. However, for our current example we consider them to be uniform.

```
desirability :: Info Applicant Hospital DScore
desirability =
info [Arthur  --> [City --> 3],
      Sunny   --> [Mercy --> 4,City --> 3],
      Joseph  --> [Mercy --> 1,City --> 5,General --> 4],
      Latha   --> [Mercy --> 5,City --> 1,General --> 1],
      Darrius --> [Mercy --> 5,City --> 5,General --> 4]]
```

We can combine the fixed and variable criteria values to generate the overall representation of applicants' preferences using the `completedWith` combinator. As the type of `completedWith` (shown in Figure 4) indicates, it takes a function with output type `d` and an `Info` value with type `c` as its third type argument representing the value type of the variable criterion. It returns as output a completed `Info` value for matching set `a` with respect to `b` using the type `d`.

```
instance Preference Applicant Hospital HInfo where
  gather = hProfile `completedWith` desirability
```

We can represent the preferences for hospitals in a similar way. Again, we begin by defining the profile of applicants `aProfile` for the fixed information, which includes the applicants' exam scores and their work experience.

```
aProfile :: Applicant -> IScore -> SStatus -> AInfo
aProfile a = case a of
               Arthur -> Appl 700 2
               Sunny  -> Appl 720 2
               Joseph -> Appl 750 1
               Latha  -> Appl 650 5
               Darrius-> Appl 790 2
```

This leaves applicants' hospital-dependent attributes, such as interview scores `IScore` and prior student status `SStatus` at a hospital, to be filled in by the individual hospitals. The interview scores of applicants at various hospitals are recorded again in a corresponding `Info` value.

```
interview :: Info Hospital Applicant IScore
interview = info
    [Mercy   --> [Joseph --> 8,Darrius --> 9],
     City    --> [Arthur -->10,Sunny --> 9,Joseph --> 4,Latha --> 6,Darrius--> 10],
     General --> [Arthur --> 9,Joseph --> 8,Latha --> 5,Darrius --> 10]]
```

Similarly, the student status of applicants at a given hospital is also represented by an `Info` value.

```
school :: Info Hospital Applicant SStatus
school = info
    [Mercy   -->[Joseph --> False,Darrius --> True],
     City    -->[Arthur --> True,Sunny --> False,Joseph --> False,Latha --> False,
                 Darrius --> False],
     General -->[Arthur --> False,Joseph --> True,Latha --> False,Darrius --> False]]
```

Finally, we can combine the applicants' profiles with their interview scores and student status information to generate an `Info` value with complete information about students. We do this by first "zipping" together `interview` and `school` using the `zipInfo` function, which results in an `Info` value where the interview score and school status information for every candidate is paired up. The function `zipInfo` is analogous to Haskell's `zip` function in that it has the effect of pairing `Info` values. We also provide functions `zipInfo2`, `zipInfo3`, and so on, for combining multiple `Info` values, corresponding to Haskell's `zip2` and `zip3`. The function `completedWith2` is a function which takes as input a profile with two unassigned fields and an `Info` value that contains these variable values and produces a completed `Info` value. We provide different variants of the `compeletedWith` function to join multiple `Info` values.

```
instance Preference Hospital Applicant AInfo where
  gather = aProfile `completedWith2` (interview `zipInfo` school)
```

This completes the specification of applicant and hospital preferences. It is instructive to see that we can get concrete rankings from our preference representations. We can do so using the `ranks` function (defined in Figure 2) as shown below. Note that the preference lists of hospitals are unchanged from Figure 1. Similarly, we can verify that the preference lists for applicants have not changed either.

```
> ranks (gather :: Info Hospital Applicant AInfo)
{City    --> [Darrius,Arthur,Sunny,Latha,Joseph] : 2,
 Mercy   --> [Darrius,Joseph] : 2,
 General --> [Darrius,Arthur,Joseph,Latha] : 2}
```

The stable matchings can be generated in the same way as we did with explicit rankings.

```
> twoWay :: Match Hospital Applicant
{City --> [Arthur,Darrius], Mercy --> [], General --> [Latha,Joseph]}
```

Since the inferred preference lists for applicants and hospitals didn't change, the stable matchings don't change either.

4 Evolution and Analysis of Matches

So far we have seen matching problems with a fixed initial set of agents. Let's assume now that some hospitals or applicants decide to amend their preferences or maybe some hospitals or applicants are added late in the NRMP cycle and need to be accommodated in the match. The straightforward thing to do would be to manually modify the preference lists and rerun the matching algorithm on this amended list. Not only is this approach prone to errors during the update, but we would also lose track of the history of the different stages of the

```
modWithRanks :: Ord2 a b => Info a b Rank -> (a,[b]) -> Info a b Rank
modWithInfo  :: Ord2 a b => Info a b c -> Info a b c -> Info a b c
modWithRow   :: Ord2 a b => Info a b c -> (a,[(b,c)]) -> Info a b c

updateWithRow   :: Ord2 a b => Info a b c -> (a,[(b,c)]) -> Info a b c
updateWithInfo  :: Ord2 a b => Info a b c -> Info a b c -> Info a b c
updateWithInfos :: Ord2 a b => Info a b c -> [Info a b c] -> Info a b c

modWithRanksDef :: (Ord2 a b,Preference a b Rank) => (a,[b]) -> Info a b Rank
...

data CompMatch a b = CompMatch {unCompMatch :: [(a,[b],[b])]}

diffMatch  :: Eq2 a b => Match a b -> Match a b -> CompMatch a b
twoWayDiff :: Info a b c -> Info a b c -> CompMatch a b
```

Fig. 5: Combinators to modify encodings and compare results.

process, which can reveal how changes in the data lead to changes in matches. An alternative is to keep the original and amend it using functions provided by the DSL. This approach makes the changes explicit, allowing users to track the evolution of data and corresponding matchings. The type signatures for some of the relevant functions for these tasks are shown in Figure 5.

4.1 Updating Ranks and Adding Agents

Assume that a new applicant Bob is added to the matching process. Like other applicants, Bob will have his preference list of hospitals. Hospitals will also need to accommodate him in their preference lists. Let's further assume that City decides not to rank him. Situations like this are of special interests to game theorists who are interested in finding out how the addition of a new applicant or a hospital might change the resulting match. For example, is it more favorable to the applicants or the hospitals? In this section, we look at how MatchMaker can be used to support such investigations.

We begin by updating the `Applicant` data type to include the `Bob` constructor.

```
data Applicant = Arthur | Sunny | Joseph | Latha | Darrius | Bob
```

We can update the preference list of applicants by adding Bob's preferences using the `modWithRanks` function. It takes as input the original preference list of applicants as well as the new applicant to be added with his preference list. The function `gather` provides the original encoding of the preferences for applicants.

```
updatedAppl = gather `modWithRanks` (Bob --> [Mercy, City, General])
```

We also update the preference lists for hospitals. Note how we can chain together multiple updates. A difference between the two values `updatedAppl` and `updatedHosp` is that, while the former creates a new record for Bob, the latter simply updates the already existing preference lists for Mercy and General.

```
updatedHosp = gather `modWithRanks` (Mercy   --> [Darrius,Bob,Joseph])
                     `modWithRanks` (General --> [Bob,Darrius,Arthur,Joseph,Latha])
```

When we need to modify the preference lists of multiple agents, rather than making one change at a time by chaining together multiple `modWithRanks` calls, it is more convenient to collect all the changes in an `Info` value and update the original encoding with it in one go. This can be done with the `modWithInfo` function, as shown below. The updated preferences of Mercy and General are stored in an `Info` value called `deltaInfo`, which can then be used to update the original preference encoding of the applicants. Note that since City doesn't appear in `deltaInfo`, its preferences are not changed in `updatedHosp`.

```
deltaInfo = choices [Mercy   --> [Darrius,Bob,Joseph],
                     General --> [Bob,Darrius,Arthur,Joseph,Latha]]

updatedHosp = gather `modWithInfo` deltaInfo
```

The function `modWithInfo` is useful for various reasons. When the number of elements being matched is large, we can keep the original data and the intended changes separate. If we need to make iterative changes, this approach keeps track of the changes performed in each iteration. We can also contemplate alternative changes to the data. We also have a `modWithInfos` combinator, which can be used to modify the original data with a list of iterative changes stored as `Info` values themselves. For example, the following expression modifies the data by four updates `i1`, ..., `i4`.

```
updated = gather `modWithInfos` [i1,i2,i3,i4]
```

If at any point we need to undo some of the changes, we can simply remove the corresponding `Info` value from the list.

Now that we have the amended preference lists for hospitals and applicants, we can use them to get new matchings using the `twoWayWithPref` function, which was introduced in Section 2.2.

```
> twoWayWithPref updatedHosp updatedAppl
{City --> [Arthur,Darrius], General --> [Latha,Joseph], Mercy --> [Bob]}
```

Notice how the matching is different from the original matchings, repeated here for convenience.

```
> twoWay :: Match Hospital Applicant
{City --> [Arthur,Darrius], General --> [Latha,Joseph], Mercy --> []}
```

Clearly, Mercy has benefited by gaining a resident. While figuring out the difference was trivial in our current example, spotting changes in even a moderately large example is more difficult. To do so systematically, we provide a function called `diffMatch`, which compares two `Match` values and reports the difference between the two matchings. In our current example, we obtain the following.

```
> diffMatch twoWay (twoWayWithPref updatedHosp updatedAppl)
{Mercy --> [] => [Bob]}
```

The result `Mercy --> [] => [Bob]` shows that that Mercy went from not having any resident in the original match to having Bob in the updated match. An interesting thing to note here is that even though we didn't annotate the type of the first argument `twoWay`, it can be inferred from the type of the second argument of `diffMatch`.

What can we say about the performance of various hospitals and applicants in the updated match, compared to the original match? Intuitively, it seems that most hospitals, namely City and General, have performed as well as they did before, while Mercy has improved its performance. Similarly, it appears that no applicants have performed worse than in the original match. Do these observations always hold? Game theory informs us that no hospital will be worse off, and some hospitals are better off compared to the original match [16, Theorem 2.26]. At the same time, none of the original applicants are better off, while some can be worse off than in the original match. In any case, MatchMaker can be employed as a tool for gaining a deeper understanding of a wide range of matching scenarios.

4.2 Updating Representational Ranks

Assume that we want to update the representational ranks of our example from Section 3.2. More concretely, suppose Mercy wants to add Sunny and Arthur, and City wants to add Sunny to their preference lists. They only need to provide the interview scores and school status for the applicants, as the other information can be obtained from the applicants' profiles. The interview scores can be updated for the two hospitals using the `updateWithRow` combinator, which takes an `Info` value to be updated along with the information to update it with. An entry such as `City --> [Sunny --> 9]` indicates that City assigns an interview score of 9 to Sunny, which is then appended to its already existing score assignments for other applicants. The function `updateWithRow` can be chained together to update the records for multiple hospitals.

```
interview1 = interview `updateWithRow` (City  --> [Sunny --> 9])
                       `updateWithRow` (Mercy --> [Sunny --> 8,Arthur --> 8])
```

And the school status also needs to be updated.

```
school1 = school `updateWithRow` (Mercy --> [Sunny --> True,Arthur --> False])
                 `updateWithRow` (City  --> [Sunny --> False])
```

Again, we also have the option to collect all changes in an `Info` value, which is then used by the `updateWithInfo` combinator.

```
deltaInterview = info [Mercy --> [Sunny --> 8,Arthur --> 8],City --> [Sunny --> 9]]

interview1 = interview `updateWithInfo` deltaInterview
```

Finally, we can use the modified interview scores and school status information to update the preferences for hospitals.

```
updatedHosp = aProfile `completedWith2` (interview1 `zipInfo` school1)
```

The changed data leads to the following preference lists for various hospitals.

```
> ranks updatedHosp
{Mercy   --> [Darrius,Sunny,Arthur,Joseph] : 2,
 City    --> [Darrius,Arthur,Sunny,Latha,Joseph] : 2,
 General --> [Darrius,Arthur,Joseph,Latha] : 2}
```

```
class Preference a b c => Exchange a b where
  endowment :: Match a b

type SameSetMatch a = Maybe (Match a a)

data CompRanks a b = CompRanks {unCompRanks :: [(a,[(b,Rank)],[(b,Rank)])]}

oneWay :: (Preference a b c,Set2 a b,Norm c) => Match a b

oneWayWithOrder :: (Preference a b c,Set2 a b,Norm c) => [a] -> Match a b
oneWayWithPref  :: (Preference a b c, Set2 a b,Norm c) => Info a b c -> Match a b

trade   :: (Preference a b c,Set2 a b,Norm c) => Match a b
sameSet :: (Preference a a b,Set a,Norm b) => SameSetMatch a

diffRanks :: (Eq2 a b,Preference a b c,Set2 a b,Norm c) => Match a b -> Match a b ->
             CompRanks a b
```

Fig. 6: Some type and function definitions for various matching problems.

We can now generate the updated match using the `twoWayWithPref` function. But perhaps it will be more interesting to see how this matching differs from the original match. As shown, the only difference in the two matchings is that Mercy which was not assigned a resident initially, now has Sunny assigned to it.

```
> twoWayDiff updatedHosp gather
{Mercy --> [] => [Sunny]}
```

5 Other Matching Problems

In addition to the two-way stable matching problem, MatchMaker also allows for the modeling of other interesting matching problems like one-sided matchings, one-sided matching with exchange, and same-set matchings, which we briefly discuss in this section. The various types and function definitions used in this section are shown in Figure 6.

5.1 Bipartite Matching With One-Sided Preferences

The first important example of a one-sided matching problem is known as the *house allocation problem* in the economics literature. In this type of matching, only the elements in the source set have preferences for the elements in the target set. The preferences of the target sets are not taken into account. Some of its applications have been allocating graduates to trainee positions, students to projects, professors to offices, and clients to servers.

As a concrete example, let us consider the problem of selecting kidney donors for various transplant patients. Assume that the donors are altruistic and don't care who their kidney goes to. Patients, on the other hand, have a preference over the kidneys: a good kidney for a patient depends on the tissue compatibility of the donor-recipient pair as well as the donor's age and their overall health

P_1	P_2	P_3	P_4
Bob	Alice	Alice	Alice
Dan	Dan	Bob	Bob
Dillon	Dillon	Dillon	Dan

(a) Preference lists of Donors to Patients.

```
data Donor = Alice | Bob | Dan | Dillon
data Patient = P1 | P2 | P3 | P4

instance Preference Patient Donor Rank where
gather = choices [P1 --> [Bob,Dan,Dillon],
                  P2 --> [Alice,Dan,Dillon],
                  P3 --> [Alice,Bob,Dillon],
                  P4 --> [Alice,Bob,Dan]]
```

(b) Encoding the example in DSL.

Fig. 7: Assigning donors to patients: Bipartite matching with one-sided preferences.

condition. Thus, the transplant team of a patient may have a ranked preference list of donors. Figure 7a shows patients with their preference lists.

Formally, the donor assignment problem is a three-tuple (T, D, P), where $T = \{t_1, \ldots, t_k\}$ is a finite set of transplant patients and $D = \{d_1, \ldots, d_n\}$ is a finite set of donors. P is a preference map such that the preference of each patient $t \in T$ is represented by an ordered list of preferences $P(t)$ on set D. We assume that each patient has a quota of 1, that is, they can be assigned just one donor. A matching $\mu : T \to D$ in this case is a partial function that assigns every patient to 1 donor.

We can represent the patient-donor example with the machinery already developed for two-sided matching. Figure 7 shows an encoding of the problem using explicit ranks. In a more realistic setting, the agency tasked with performing the match might prefer to rank the donors using meaningful representation such as age and the blood and tissue compatibility between the donor-patient pair.

How do we assign donors to the patients based on their preferences? The strategy we use here is the so-called *serial dictatorship mechanism* [1]. It is a straightforward greedy algorithm that takes each patient in turn and assigns them to the most preferred available donor on their preference list. The order in which the patients are processed will, in general, affect the outcome. In applications where elements have a quota of n, they are assigned to n objects when their turn comes for processing. For our example here, we expect that a matching agency will come up with an order of processing based on factors such as the urgency of a patient's situation, their age, or their time on the waiting list. The function `oneWayWithOrder` performs serial dictatorship with a given order as shown below where patient P_3 gets its first choice donor, P_4 gets its first choice amongst the remaining donors, and so on.

```
> oneWayWithOrder [P3,P4,P2,P1] :: Match Patient Donor
{P1 --> [Dillon],P2 --> [Dan],P3 --> [Alice],P4 --> [Bob]}
```

Oftentimes users might prefer that the matching function infer a preferred order based on position of the constructor in the data definition for donors, that is, the `Donor` data definition implies an order of `[P1,P2,P3,P4]`. The function `oneWay` generates a one-way match with this implicit order.

```
> oneWay :: Match Patient Donor
{P1 --> [Bob],P2 --> [Alice],P3 --> [Dillon],P4 --> [Dan]}
```

Finally, there is also a third variant of the function `oneWayWithPref` that takes explicit preference encoding like its counterpart `twoWayWithPref`.

As we can see, these two matches are different because they are generated using different orders. Is one better than the other? What is the best possible match among the various possibilities? Manually comparing one match with another is cumbersome because for every patient we have to look at the two matchings and compare the relative ranks of the two donors in that patient's preference list. This task is simplified by the function `diffRanks`, which compares the ranks of the two matchings using a type called `CompRanks`. This type represents for every agent the element assigned to them in those matchings as well the elements' ranks for comparison. In the following expression, we use `x = oneWayWithOrder [P3,P4,P2,P1]`.

```
> diffRanks oneWay x :: CompRanks Patient Donor
{P1 --> Bob : 1 > Dillon : 3, P2 --> Alice : 1 > Dan : 2,
 P3 --> Dillon : 3 < Alice : 1, P4 --> Dan : 3 < Bob : 2}
```

The first match is advantageous for patients P_1 and P_2, whereas the second match is advantageous for patients P_3 and P_4. Informally, a matching is *Pareto optimal* if there is no other matching in which some patient is better off, whilst no patient is worse off. It is used as a metric to compare the quality of outcomes in game theoretic matchings. The deceptively simple-looking *serial dictatorship* algorithm results in Pareto optimal matchings, which implies that for any two matchings, there are some patients for whom one match is better and for some, the second match is better. In other words, a unique best match doesn't exist.

5.2 Bipartite Matching With One-Sided Preferences and Exchange

We assumed the presence of altruistic donors in our last example. However, kidneys are valuable commodities, and altruistic donors alone can't fulfill the vast demand for them. A more realistic scenario is a family member or a friend donating one of their kidneys to a loved one. However, sometimes this donation may not happen due to reasons like tissue or blood group incompatibility. An elegant solution was developed in the field of economics. Suppose (d_1, r_1) and (d_2, r_2) are two donor-receiver pairs such that d_i wants to donate to r_i but can't do so. However, if d_1 could donate to r_2 and d_2 to r_1, then both the patients would be able to receive kidneys. This could be easily scaled to multiple pairs generating large numbers of compatibility pairs. The actual *kidney exchange mechanism* [17] is a little more complicated, but the exchange between multiple donor-receiver pairs is at the heart of it. This exchange characterizes our next matching algorithm, the so-called *top trading cycle* (TTC) matching mechanism for one-way matching where every element has an initial endowment and a preference list [18]. The resulting match takes both of these into account.

Take the patient-donor example we considered in the last section. At the start, some donor, presumably family or friends willing to donate a kidney, is assigned to each patient. These initial set of donors are sometimes also called the *initial endowment*, or just *endowment*, of a patient. Assume that patients $P_1, \ldots, P_4$ are endowed with Bob, Dan, Alice, and Dillon, respectively, such that

all the patients are compatible with the donors they are endowed with. In this case, TTC tries to find out if the patients can do better than the donor they are assigned to, based on their preference lists. We start by representing endowments for which we define the multi-parameter type class `Exchange`, which has a `Preference` class constraint (see Figure 6). The instance definition of `Exchange` for our example is as follows.

```
instance Exchange Patient Donor where
    endowment = assign [P1 --> Bob,P2 --> Dillon,P3 --> Alice,P4 --> Dan]
```

Now we can use the function `trade` provided by the DSL to generate the matching.

```
> trade :: Match Patient Donor
{P1 --> [Bob], P2 --> [Dan], P3 --> [Alice], P4 --> [Dillon]}
```

Did any patient gain as a result of the change? We can use the `diffRanks` function we saw in the previous section to find out. We discover that patients P_2 and P_4 do indeed profit by exchanging their donors.

```
> diffRanks endowment trade :: CompRanks Patient Donor
{P2 --> Dillon : 3 < Dan : 2, P4 --> Dan : 3 < Dillon : 2}
```

5.3 Same-Set Matching

This variation of the problem is the so-called *stable roommate problem* [6, 9] where the source and the target sets being matched are the same. For example, a set of students living in the dormitory can supply a ranked preference list of other students they want to be roommates with. An example is shown below.

```
data Student = Charlie | Peter | Kelly | Sam

instance Preference Student Student Rank where
    gather = choices [Charlie --> [Peter,Sam,Kelly],   Peter --> [Kelly,Sam,Charlie],
                      Kelly   --> [Peter,Charlie,Sam], Sam   --> [Charlie,Kelly,Peter]]
```

We can obtain a stable matching of roommates using Irving's algorithm [8]. In order to capture the fact that the source and target sets are the same, we define a type synonym `SameSetMatch` that assigns the same type `a` to both the source and the target sets in the `Match` type (see Figure 6). Even though same-set matchings are stable matching problems like the bipartite two-sided matching problems, they are different in that a stable match always exists for the former, whereas it may not always exist for the latter. This fact is reflected by the `Maybe` constructor in the type definition of `SameSetMatch`. Finally, we can generate the same-set matching using the `sameSet` function, which produces the following result for our example from above.

```
> sameSet :: SameSetMatch Student
Just {Charlie --> [Sam], Peter --> [Kelly]}
```

6 Related Work

Matching [20] is a library for Python that allows users to encode simple matching problems in a straightforward manner. An issue with the library is that all the encoding are done using strings, which makes error handling difficult and thus complicates the maintenance and debugging of larger examples. In comparison, MatchMaker avails the strongly typed feature of the host language Haskell to detect the various errors in encoding.

Similarly, *matchingMarkets* [10] is a matching library for R. The advantage of the library is that it implements a wide variety of matching algorithms developed in the matching theory. Additionally, it implements statistical tools to correct for the sample selection bias from observed outcomes in matching markets, which is something that MatchMaker doesn't do. The library encodes the preference relation between the sets of elements being matched in the form of a matrix. While an efficient way to encode the preferences, the matrix encoding is clunky and is thus difficult to understand, update and maintain. Another stable matching library for R and C++ *matchingR* is [19], which uses matrices to encode the preference relations and thus suffers from the same problems as *matchingMarkets*.

MatchMaker allows users to specify their preferences more abstractly in terms of attributes that they understand, while all the previous libraries only allow specification of preference in terms of ranks. Additionally, none of these libraries offers either the primitives for systematic modification of representations or primitives to compare and contrast different matchings.

Matching problems can also be solved using constraint programming [13, 5] or SMT solving [3]. Moreover, integer linear programming can be used to solve NP-hard stable marriage problems, including ones with ties and incomplete lists as well as the many-to-one generalization [2]. While powerful, a potential downside is that encoding matching problems as constraints might be challenging for users. In contrast, MatchMaker facilitates high-level representations of matching problems and can thus be used without any specialized knowledge.

7 Conclusions

MatchMaker is an embedded DSL in Haskell for expressing, solving, and analyzing game-theoretic matching problems. Our implementation leverages advanced type system features of Haskell to facilitate high-level representations of matching problems, expressed in terms of domain elements. MatchMaker also supports the maintenance and evolution of the problem representation and provides some limited support for analyzing computed results, making it a useful tool for end users as well as game theorists.

The design of our DSL emphasizes the significance of strong typing in detecting errors at compile time. Additionally, employing multi-parameter type classes promotes a lucid mental representation of the problem, helping users to comprehend problem structures and implement complex functions.

References

1. Bogomolnaia, A., Moulin, H.: A new solution to the random assignment problem. Journal of Economic Theory **100**(2), 295–328 (2001)
2. Delorme, M., García, S., Gondzio, J., Kalcsics, J., Manlove, D., Pettersson, D.: Mathematical models for stable matching problems with ties and incomplete lists. European Journal of Operational Research **277**(2), 426–441 (2019)
3. Drummond, J., Perrault, A., Bacchus, F.: Sat is an effective and complete method for solving stable matching problems with couples. In: Proceedings of the 24th International Conference on Artificial Intelligence. p. 518–525. IJCAI'15, AAAI Press (2015)
4. Gale, D., Shapley, L.S.: College admissions and the stability of marriage. The American Mathematical Monthly **69**(1), 9–15 (1962)
5. Gent, I.P., Irving, R.W., Manlove, D., Prosser, P., Smith, B.M.: A constraint programming approach to the stable marriage problem. In: Proc. of the 7th International Conference on Principles and Practice of Constraint Programming. p. 225–239. CP '01, Springer-Verlag (2001)
6. Gusfield, D.: The structure of the stable roommate problem: Efficient representation and enumeration of all stable assignments. SIAM Journal on Computing **17**(4), 742–769 (1988)
7. Gusfield, D., Irving, R.W.: The Stable Marriage Problem: Structure and Algorithms. MIT Press, Cambridge, MA, USA (1989)
8. Irving, R.W.: An efficient algorithm for the "stable roommates" problem. Journal of Algorithms **6**(4), 577–595 (1985)
9. Irving, R.W., Leather, P.: The complexity of counting stable marriages. SIAM Journal on Computing **15**(3), 655–667 (1986)
10. Klein, T., Aue, R., Giegerich, S., Sauer, A.: matchingMarkets: Analysis of Stable Matchings in R (2020), `https://matchingmarkets.org/`
11. Manlove, D.F.: Algorithmics of Matching Under Preferences. World Scientific (2013)
12. NRMP: National Resident Matching Program (2022), `https://www.nrmp.org/intro-to-the-match/how-matching-algorithm-works/`
13. Prosser, P.: Stable roommates and constraint programming. In: CPAIOR (2014)
14. Roth, A.E.: Deferred acceptance algorithms: History, theory, practice, and open questions. Working Paper 13225, National Bureau of Economic Research (2007)
15. Roth, A.E., Peranson, E.: The redesign of the matching market for american physicians: Some engineering aspects of economic design. American Economic Review **89**(4), 748–780 (September 1999)
16. Roth, A.E., Sotomayor, M.A.O.: Two-Sided Matching: A Study in Game-Theoretic Modeling and Analysis. Econometric Society Monographs, Cambridge University Press (1990)
17. Roth, A.E., Sönmez, T., Ünver, M.U.: Kidney exchange. The Quarterly Journal of Economics **119**(2), 457–488 (2004)
18. Shapley, L., Scarf, H.: On cores and indivisibility. Journal of Mathematical Economics **1**(1), 23–37 (1974)
19. Tilly, J., Janetos, N.: matchingR: Matching Algorithms in R and C++ (2020), `https://github.com/jtilly/matchingR/`
20. Wilde, H., Knight, V., Gillard, J.: Matching: A python library for solving matching games. Journal of Open Source Software **5**(48), 2169 (2020)
21. Williams, K.J., Werth, V.P., Wolff, J.A.: An analysis of the resident match. New England Journal of Medicine **304**(19), 1165–1166 (1981)