VIENNA UNIVERSITY OF TECHNOLOGY

105.625 PR ADVANCED ECONOMICS PROJECT

Double Sided Matching

9702170, Karin Leithner
0725439, Florin Bogdan Balint
1025735, Clemens Proyer
1027143, Mattias Haberbusch
1027433, Thomas Solich

June 20, 2016

Contents

Introduction

Double sided matching is about finding a match between two sets of elements. The first algorithm which should solve this problem was described by David Gale and Lloyd S. Shapley in 1962 [1]. The focus of this report is on the examination of this algorithm and the creation of a simulation model for it, which can be applied to different economic issues. The selected economic issues will be presented in the following chapters. Due to another important project in the course of this lecture only the partner-matching algorithm was implemented. The other two economic issues, the university application and labour supply/demand problem, could not be implemented.

Motivation and Problem Description

Many everyday situations contain hidden double sided matching problems. For example, students usually have the opportunity to apply to a number of schools. In this case students and schools have to be matched together. Both parties have preferences which need to be taken into consideration. Yet, in our everyday life we usually do not have the time to find an appropriate solution for our problems.

In some cases these problems even go by without solving them. In other situations however, it is crucial to solve them, e.g. if we consider medical school graduates and open positions in hospitals. Due to multiple reasons like different preferences or a high number of elements in the two sets this problem cannot be solved intuitively or in linear time.

Aim and Objectives

The aim of this paper is to develop a simulation model prototype which can solve similar problems, like the ones discussed above. Questions, which should be answered with the simulation model are:

- Do there exist "stable" solutions for these kinds of problem?
- How can it be applied to all prospective students and universities in Austria?
- Is there also a solution for the current labour supply and demand in the European Union?

Structure

In the second chapter a theoretical overview is provided regarding the double sided matching problem and what stability means in this context. The focus lies on the Gale and Shapley Algorithm, basic information about the game theory and the relationship between them.

The third chapter describes the prototype of the simulation model, which puts the previously discussed theory into practice. As a simulation environment NetLogo [1] will be used. The prototype will be firstly evaluated on a simple discotheque example. In the disco example it is assumed, that an equal number of men and women are in a discotheque and the objective is to form couples.

Finally the most important findings and results are summarized and compared in the last chapter. Additionally a short outlook to further research areas will be provided.

[1] Available at https://ccl.northwestern.edu/netlogo/, accessed April 28, 2016

Theory

In this chapter the necessary theoretical background for the realization of this project is discussed.

The Double Sided Matching Problem

The double sided matching problem focuses on two groups of elements. These sets are disjoint and may be equal in size, but this condition is not mandatory. The purpose of the double sided matching algorithm is to find a solution, so every element of the first set has a corresponding counterpart in the other set. This can be best described in an example: Assume that the elements are men and women and that individuals have only heterosexuals preferences. Consequently, the two sets are male and female persons. The double sided matching algorithm shall create couples, which consist of a man and woman. Optimally, every person has a partner who has a high position in their list of preferences.

The solutions of the matchings can have different properties, which might be very important (e.g. stability). The output of the algorithm of men and women is not allowed to be unstable, i.e. there are two men α and β who are assigned to women A and B although β prefers A to B and A prefers α to β. If this situation does not occur, the assignment is called *stable*. If there exists more than one stable solution the *optimal* one can be of particular interest.
Optimality means that a solution has the best outcome for an individual or set. In this case optimality can be achieved from the perspective of the first set of elements or from the second one. There can also be a trade-off from the perspective of the both sets. In this case optimality is achieved by weighting the expected value of both sets. These properties have been described in a more detailed way in the paper by Gale and Shapley [1].
Generally in economics this is also known as pareto efficiency [2, p. 46]. Pareto efficiency is a characteristic for a solution where any change which would improve one property on one hand, but will worsen another one on the other hand.

Gale and Shapley Algorithm

In the paper of Gale and Shapley [1] an algorithm was proposed for solving the double sided matching problem. The assumption is that there are two sets of elements: one contains men and the other one women. The size of the two sets is equal, the preferences are heterosexual and there are no polygamies (i.e. one element is only allowed to be matched with one element from the other set). The aim of the algorithm is to find a stable way of marrying men and women. Each man and woman has own preferences of the marriage partner.

First Iteration

In the first iteration each man proposes to the woman who he ranked first. Afterwards a woman has a list of men that proposed to her. This list might be zero, if not she rejects

every men, except her highest preferred man on this list. If a man is not rejected he and the woman create a temporary couple. All man who are not in a couple are single.

Second Iteration

In the second interaction each man who is single, proposes to his highest ranked woman, who he didn't already proposed to. Each woman now chooses the man which she ranks the highest out of her received proposals. If a man is not rejected he and the woman create a temporary couple (Note: after the second iteration some men, who have been in a couple from the first iteration might be now single).

Further Iterations

The third iteration is the same as the second one. This continues until no men is single any more. If this condition is satisfied a stable solution is found for the double sided matching problem. The stability of the solution of this algorithm has been proven by contradiction [1].

Relevance to Game Theory

General

"Game theory is about what happens when people - or genes, or nations - interact" [3, p. 1]. An important aspect for participating parties is to anticipate how the opposite party will react on certain actions. Mathematics shall help to analyse, understand and estimate outcomes of such games. Depending on the information participants have, they decide how to act basing on rules contained in their strategy. Game theory is widely applied in economics. Companies use game theory to estimate e.g. reactions of competitors or behaviour of employees. The major advantages of game theory are its precision and its universal applicability to all kind of games. [3, pp. 1-3]

In addition there are some assumptions which are made during preparation and execution by every individual who is interacting with another one:

- Well-specified choices
- Well-defined end-state
- Specified pay-off
- Perfect knowledge
- Rationality

However, these perfect preconditions will never be met in real scenarios. In the context of double sided matching, it is important to have well-specified choices defined. Nevertheless, there might be chances to get into a nearly worst-case scenario instead. But

furthermore, as a consequence of these listed assumptions above, there are two different kind of games:

- Static ones
- Dynamic ones

The two items can be combined with the following kind of knowledge

- Incomplete
- Complete (there is no private information)

Depending on the nature of the game, whether it is static or dynamic and the knowledge is complete or incomplete four different combinations can be made. For a better understanding, following examples only consist of two players (every combination can be used for a higher amount of players too):

- Static and (in)complete information
  As of complete information, the players action set can be presented in an N x M matrix. Therefore all information (also private) is known to every player and each player is able to easily eliminate the pay-offs of the other players. The pay-off is the outcome of a chosen action of a player which depends on the state or previous actions of the game. The pay-off can change dynamically while the game is in progress. As of incomplete information players are forced to be uncertain about the other player's actions and pay-offs. Besides that, there are two basic steps
  1. Player 1 and 2 are choosing actions out of a set, which were predefined by them before.
  2. After their actions, they receive their pay-offs.
- Dynamic and complete information
  When basic assumptions are made on complete information, players will create a strategy to ensure to get the highest return of each action. Although the strategy might be changed during the game after each turn. Incomplete information leads to communication of uninformed parties (i.e. closed private information) which might end up to under- or overestimate the others pay-offs. In general there will be a Nash Equilibrium between those parties. This equilibrium is created by players because they choose a strategy which do not depend on the other players. There are three basic steps:
  1. Player 1 chooses an action out of his/her predefined set.
  2. Player 2 observes player 1's action and chooses an action based on his/her observation.
  3. After their actions, they receive their pay-offs.

The previously described theory is based on information of the paper "An introduction to applicable game theory" [4].

Example for the Application of Game Theory

Game theory is also very popular in the field of economics. Many companies primarily apply game theory to estimate the reactions of their competitors. In the last auction for broadband internet frequencies in Germany, all major providers hired game theorists out of two reasons. Firstly, they wanted to win the auction for the desired frequency and secondly, they also did not want to overpay for the frequency. The bidding behaviour of a provider indicated the interest rate for a frequency. As a consequence, other providers retained their bids for parts of this frequency so the interested provider did not need to overpay for it. The positive result for the providers was, that the 700-MHz frequency parts were sold for the minimum bidding price. [5]

Double Sided Matching Problem and Game Theory

Like described in chapter 2, the goal of the three main problems, which will be simulated in the course of this project, is to bring different parties together. In the original marriage problem discussed by Shapley, the goal was to match men and women so the overall situation is stable. This idea can also be applied to other areas like the labour market or university applications. In each of these three problems the parties need to have a strategy how to react to moves of the counterpart. If, for example, a university offers a place to a student and this is not the student's favourite one, he/she has to decide whether to accept the place (to be on the safe side) or still hope to get accepted from the favourite university (and as a consequence accept to be on the waiting list there). Not only the student needs to have a strategy how to deal with these situations but also universities have to incorporate different reactions of students into their application process (e.g. how many students to invite) [1].
The double sided matching problem is a dynamic game. The players of the two sets have incomplete information. The algorithm uses the information of both sets, therefore it has complete information.

An important aspect of the game or matching is that the rules have to be clear, which was mentioned by Roth and Sotomayor. The way agents are matched to each other influences the analysis of the problem. A possible rule might be that individuals like a student and university are only brought together if both parties agree to the matching. Other norms of a game might contain the way an individual proposes to another one or whether there exists a moderating individual [6, p. 492].

Double Sided Matching Problem and Optimal Stopping

The double sided matching problem also relates to the optimal stopping problem. The optimal stopping problem concerns itself with choosing the optimal time to stop an undertaken action and at the same time to minimize the expected costs and maximize the profit. A concrete example for the optimal stopping problem could be a men trying to find a women to spend the rest of his life with. He can meet several different women but when is the optimal time to stop looking?

A few algorithms leading to a possible solution for the optimal stopping problem have been presented by Christian & Griffiths 2016 [7]. For the previously mentioned example of partner search, one algorithm proposes to spend 37% of the time in building standards. Afterwards, the man should immediately propose/choose the next woman he meets who satisfies or over-exceeds his standards. 37% is the provably optimal solution [7, p. 2]. This algorithm is recommended in the case of *no-information games* (e.g. in the case of a partner search, it is hard to quantify the information, because it is hard to compare people to each other. There is no concrete criteria, one can only assume that one person is better or more suitable than another person, but not by how much) [7, p. 18].

Another possible solution for the previously mentioned problem is to set a threshold from the beginning and to take the first option that exceeds it. Christian & Griffiths 2016 [7] recommended this approach in the case of *full-information games*. A suitable example for this case is the sale of a house: the value of the house can be specified clearly and there exists also information about the state of the market. The combined knowledge allows to predict a range of offers. In this case a threshold is set and the first offer that exceeds this threshold can be accepted. This example does not include waiting costs. If waiting costs exist, they also have to be taken into account. Therefore, it is recommended to decline an offer if the chance of a better offer multiplied by how much better it is compensates the waiting costs.

Finally Christian & Griffiths 2016 [7, pp. 28-30] recommend to always stop looking at a certain point.

Prototype

A prototype for solving the double sided matching problem was developed in NetLogo[2]. In this chapter the prototype is evaluated on a discotheque example. In this case, it is assumed that an equal number of men and women are in a discotheque and the objective is to form couples based on the individual preferences. Afterwards the prototype will be evaluated on more complex problems (e.g., matching students and universities and matching labour supply and labour demand). In the following, the data model, the data generation script, the graphical user interface (GUI) and code from NetLogo will be described.

Data Model

According to the problem description we defined a generic data model for every individual. This data model is valid for the discotheque example, the university and labour market example, and contains the following attributes:

- id: unique object identifier
- name: object name (e.g. male1 or female1)
- maxMatchesInt: integer value representing the maximum number of matchings for this individual (e.g. in the disco example each person is maximally matched with one other person)
- sideInt: integer helper variable assigning an individual to one of the two participant groups
- partnerList: list of identifiers of the partners, ordered according to the preference of an individual
- rankList: list of ranks (values between 1 to 0) for the partners from the partnerList; the list is ordered from highest to lowest preferences. If the value is equal to 1 this partner is a perfect match.
- hasProposedToList: list of identifiers representing the individuals to which an individual has proposed
- gotProposedByList: list of identifiers representing the individuals from which an individual received proposals. The gotProposedByList is only valid for one iteration.
- tmpMatchList: list containing the identifiers of individuals to whom a temporary match was established
- activeFlag: boolean value stating if an individual is matched or not respectively if the individual gave up (they already proposed to all individuals from the partnerList but only received rejections)

[2] Available at https://ccl.northwestern.edu/netlogo/, accessed April 28, 2016

Data Generation

In order to generate random data, which represents the individuals and their preferences the statistical computing language R[3] has been used. The generated data is exported as a CSV list, which is then read from NetLogo. The R script accepts the following parameters:

- seed: integer value, which is used as a seed by the number generator. The default value is 123.
- numberOfMen: integer value representing the number of men which should be created. The default value is 10.
- numberOfWomen: integer value representing the number of women which should be generated. The default value is 10.
- pickyLower: float value between 0 and 1 representing the lower bound which will be used in order to generate a number that represents how picky a person is. The value 0 means accepts every possible match, the value 1 excepts none.
- pickyUpper: float value between 0 and 1 representing the upper bound which will be used in order to generate a number that represents how picky a person is. The value 0 means excepts every possible match, the value 1 excepts none.

The script can be called as follows from the command line:

```
Rscript initDisco.R 123 10 10 0 0
```

The script generates a CSV which looks as follows:

```
1 "";"id";"name";"maxMatchesInt";"sideInt";"partnerList";"rankList"
2 "1";1;"male1";1;1;"13#18#14#17#16#11#20#19#12#15";
      "0.96#0.95#0.9#0.68#0.57#0.45#0.33#0.25#0.1#0.04"
3 ...
4 "11";11;"female1";1;2;"3#9#5#4#10#8#2#1#6#7";
      "0.92#0.82#0.7#0.67#0.48#0.41#0.35#0.25#0.22#0.05"
```

[3] Available at https://www.r-project.org/, accessed April 28, 2016

GUI

One of the main goals in the design phase was to keep the GUI simple and clear. The GUI was structured in two main parts - the control and the information section.

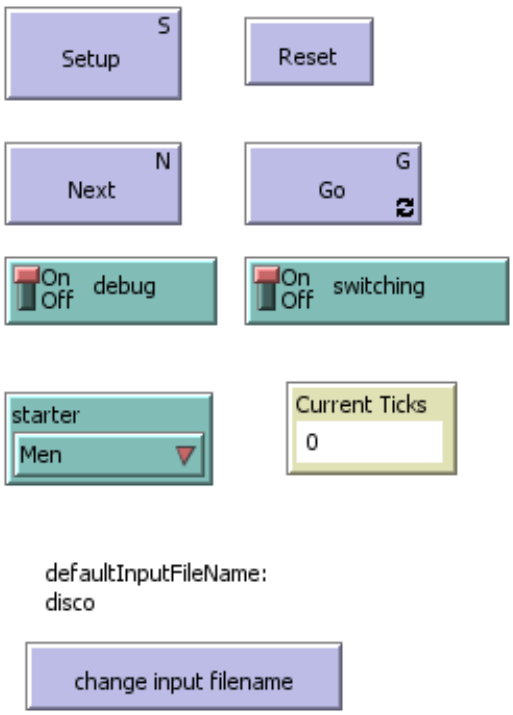

Figure 3.1: The control section for the disco example

Figure 3.1 shows the control section in the GUI. With the buttons **Setup** and **Reset** the simulation model can be initialized or reset after a simulation run. The button **Next** advises the program to perform one simulation step, if **Go** is activated the simulation continues until a termination criterion is met. In the disco example the main termination criterion is if every proposing person is already matched to another person. Another criterion for termination is if the proposing person already proposed to every person in his partnerList but got rejected every time. The **debug** switch enables or disables debug messages which are printed during the setup phase and simulation run. The speed of a simulation run can be increased if the **debug** switch is set to Off. The second switch, **switching**, controls, if the two parties propose to each other alternatively (On) or if only one party proposes to the other one (Off). With the drop-down list **starter** the user can determine which party starts proposing. The reporting box **Current Ticks** shows the current number of simulation steps. The button change input file name raises a dialogue box where the user can enter the filename of an input file which shall be used in the simulation. Together with the number of ticks the file name will also be used as the name for the CSV export file.

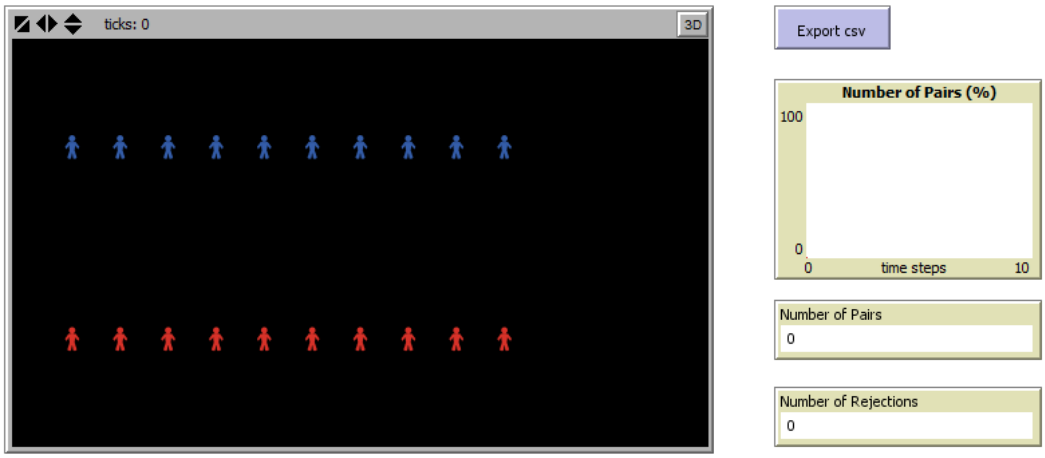

Figure 3.2: The information section for the disco example

Figure 3.2 shows the information section of the disco example GUI. All created humans are shown in the world area. Men are coloured blue and women are red. On setup of the simulation the proposing site is shown in the top row. As soon as two people are matched their colour changes to green and a link between the two is established. If a couple is divorced the link is removed and the colour changes back to the original one. The plot named **Number of Pairs (%)** illustrates at each point in time how many of the possible couples exist. If the simulation contains 10 men and 10 women and currently 4 couples exist, the plot shows that 40% are matched. The reporting box **Number of Pairs** shows the absolute number of couples. The other reporting box, **Number of Rejections**, indicates how many couples were divorced over time. With the button **Export CSV** the current properties of all humans are exported to a CSV file. The file name is a combination of the input file name, the starting side, if the switching is activated and the current number of ticks.

NetLogo Code

At the beginning of the NetLogo code the extensions, the data model and the needed global variables are defined. The rest of the code is based on procedures (i.e. a sequence of NetLogo commands which begins with *to* and ends with *end*). The most important procedures are the following:

- *setup* and *reset*: both clear all results and reset the whole model (especially the needed variables) to an initial empty state.
- *open-file*: procedure used to open and import a CSV list which represents the individuals and their preferences. The turtles for the simulation are then created based on the read information.
- *setup-globals*: procedure used to set up all global variables (e.g., current number of pairs, shape of the humans, etc.)
- *step*: procedure which represents going one step in the double sided matching algorithm. Within this step one side proposes to the other one. The proposal-receiving site processes (accepts/rejects) the proposals.
- *go*: calls the procedure step until a termination criterion is met.
- *calc-stats*: calculate some statistical information (e.g., number of pairs, number of rejections).
- *clear-before-match*: empties the list of the proposals as a preparation for the next matching round.
- *propose-to[sender receiver]*: sends the proposals (e.g. manages lists containing IDs of individuals an individual already proposed to/an individual got proposed by).
- *process-proposals [tmpId]*: processes the proposals (e.g. orders incoming proposals according to the own preference, calls the method create-tmpCouples to form a couple with the most desired partner, calls method reject-proposals to reject the other proposers).
- *reject-proposals [tmpId rejectList]*: rejects the proposals (e.g. removes the partner-ID from the tmpMatchList, sets the status of the turtle to active - the individual shall try to find another partner in the next round, sets the colour of the rejected individual to the original one and removes an existing link).
- *create-tmpCouples [tmpId acceptList]*: creates temporary couples (e.g. puts the ID of the partner in the tmpMatchList, sets the colour of the individual to green, creates a link to the partner).

The Netlogo program itself contains comments which explain the procedures and performed steps in a more detailed way.

Simulation Runs

The discotheque problem was simulated three times: one time with a simple and small dataset of 20 persons (10 men and 10 women) and twice with a dataset of 200 persons (100 men and 100 women).

Simulation Run 1: Small Dataset

In the first simulation run a dataset containing 20 individuals (10 males and 10 females) was used. This dataset was generated using the previously described R script. After the first simulation step already 6 couples were formed (see figure 3.3).

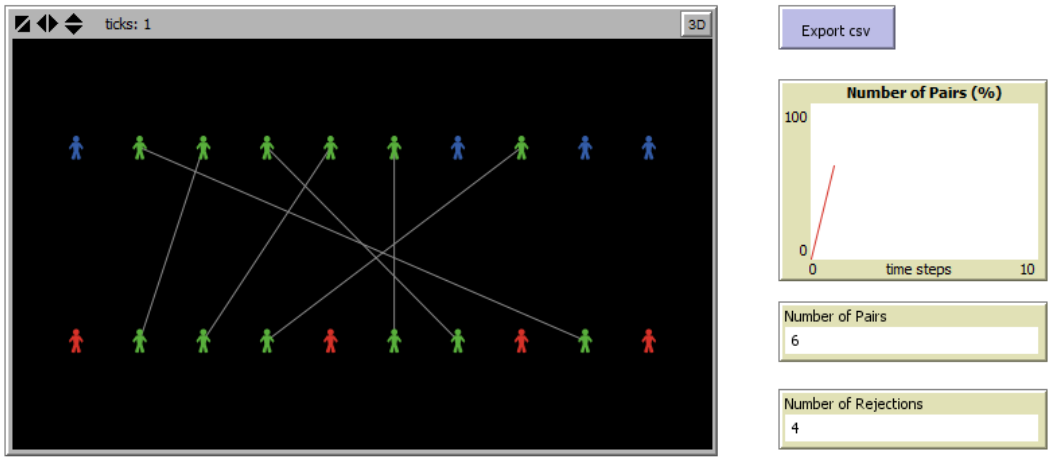

Figure 3.3: Simulation 1, step 1

In the next few steps the number of couples continued to increase by 1 at each step. After the third step 8 couples were formed (see figure 3.4).

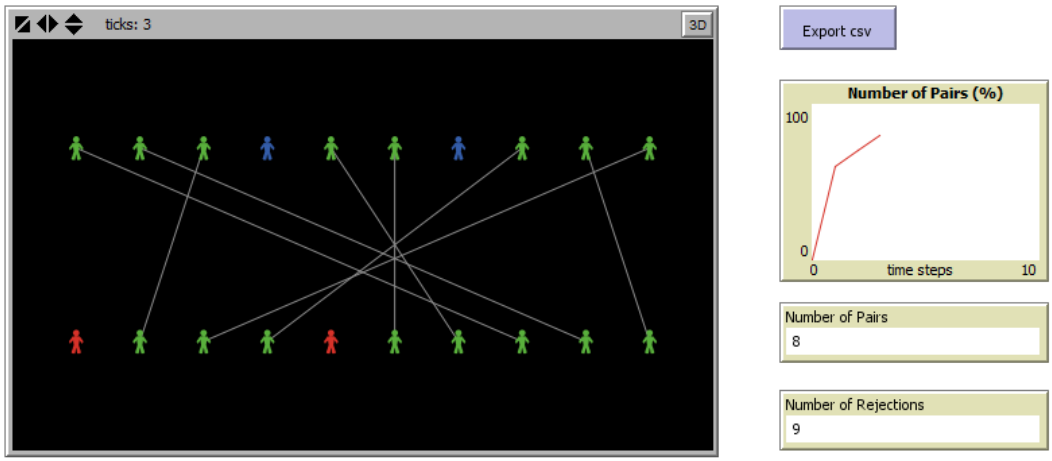

Figure 3.4: Simulation 1, step 3

Afterwards the number of couples continued to increase slowly and some of them changed according to the individuals preferences. As we can see in figure 3.5 the male number 10 (upper right corner) was in a couple with female number 3. However, male number 10 is now single, because female number 3 changed the partner and is now a couple with male number 6. Female number 3 changed partner because male number 6 better suits her preferences.

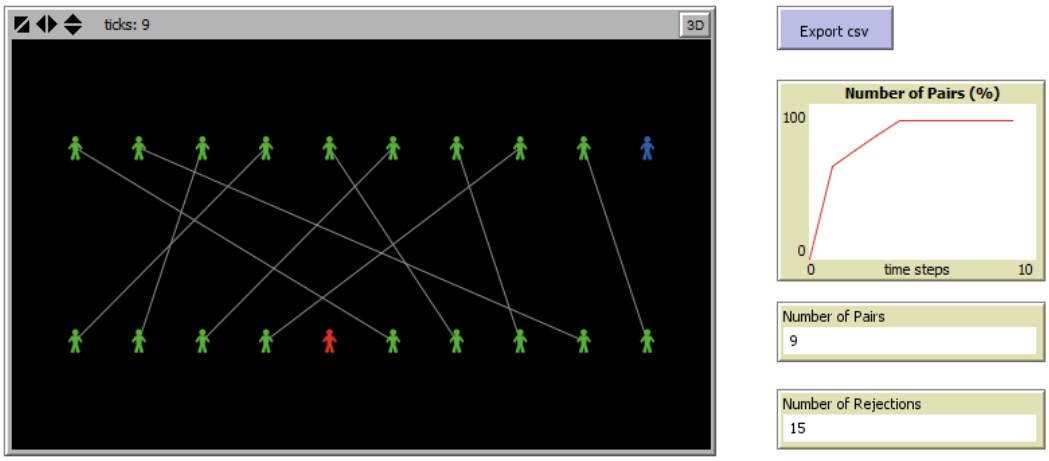

Figure 3.5: Simulation 1, step 9

In the further steps, the couples continued to change according to the individual's preferences. Finally, after 27 steps the double sided matching problem was solved and 10 couples have been created (see figure 3.6).

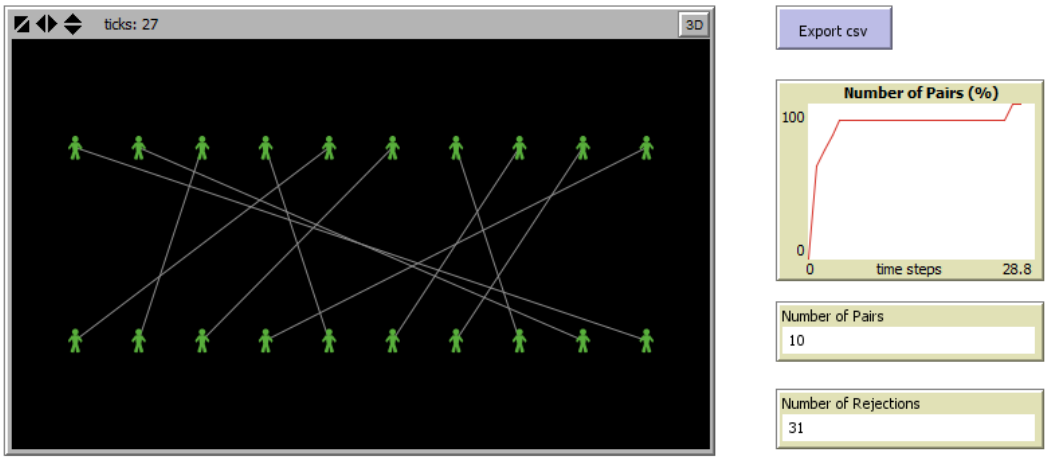

Figure 3.6: Simulation 1, step 27

Simulation Run 2: Large Dataset

In the second simulation run, the algorithm was applied to a dataset containing 100 men and 100 women. In the first step of the simulation 49 couples were formed (see figure 3.7).

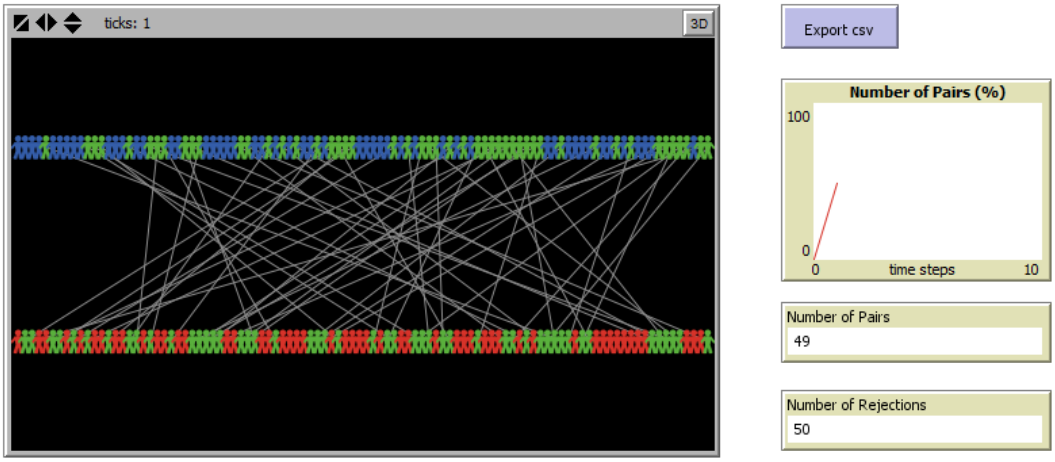

Figure 3.7: Simulation 2, step 1

As observed in the previous simulation, in the next few steps the number of formed couples continued to increase (see figure 3.8).

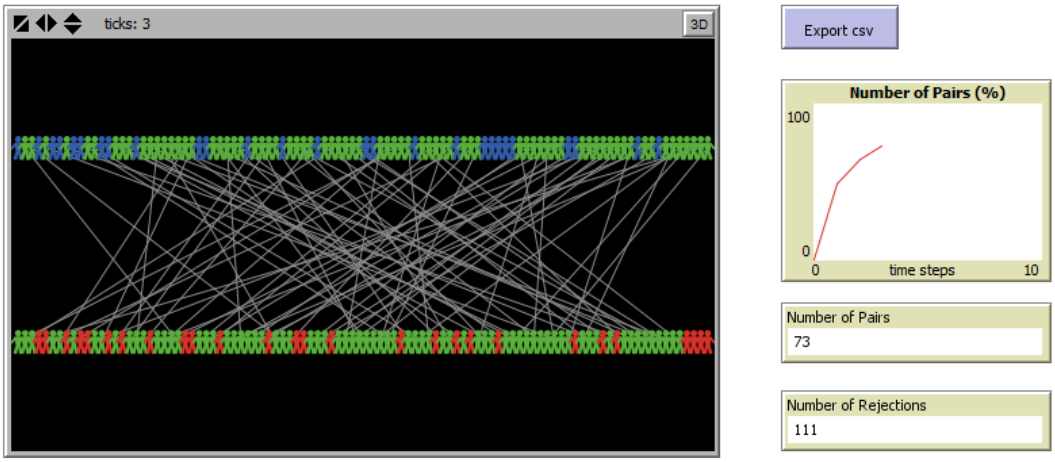

Figure 3.8: Simulation 2, step 3

As the number of steps grew, the number of couples increased only slowly and a change of couples could be observed (see figure 3.9).

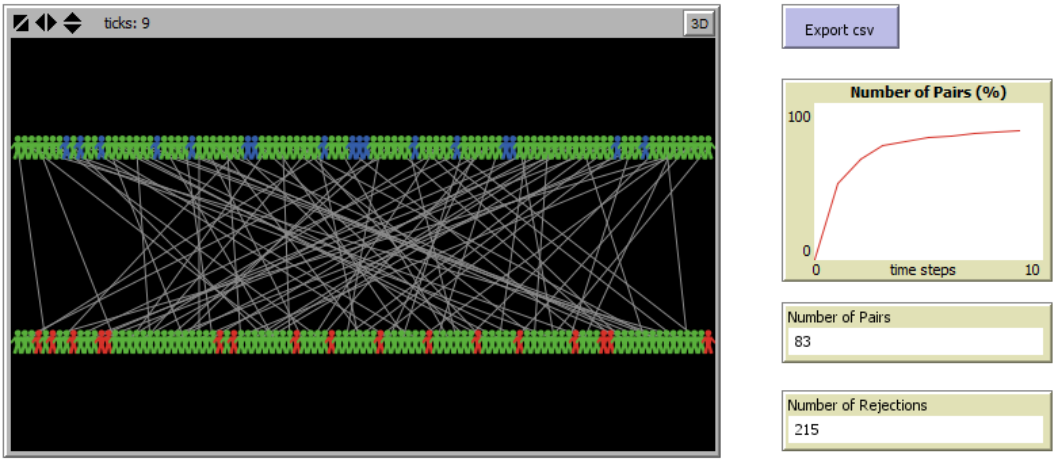

Figure 3.9: Simulation 2, step 9

As the simulation went on, the model started to reach a stable point, where the number of couples increased only slowly and the number of rejections increased much faster. Finally, after 127 steps, the model has reached a final state where 94 couples were formed. Not every individual could find a partner. This might happen if a person was in a partnership, got rejected and could not find another partner who was not already in a partnership with a higher-ranked person. The final state of the model is illustrated in figure 3.10.

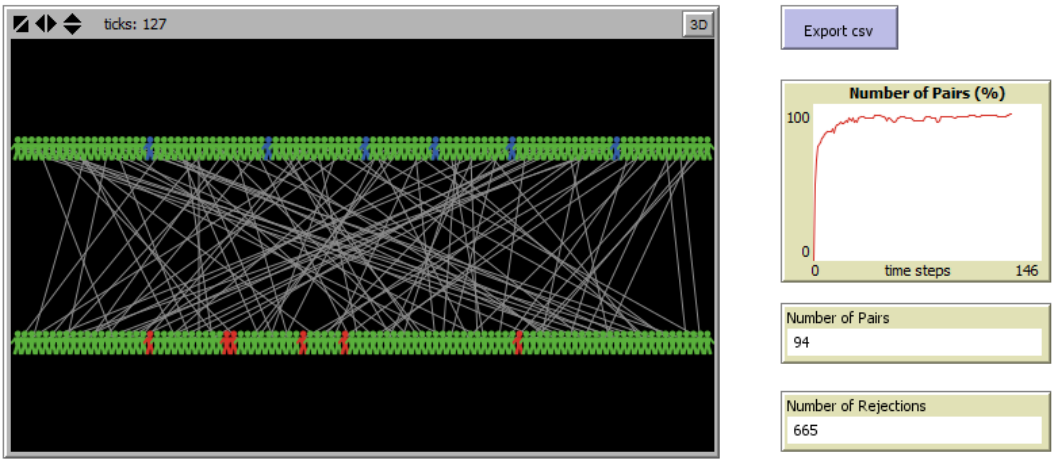

Figure 3.10: Simulation 2, step 127

Simulation Run 3: Large Dataset and Lower Preferences

The second simulation was repeated with a similar dataset. This time however, the individuals did not have too many preferences (i.e., they were not so picky) regarding their partner. During this simulation the couples were formed faster. After the first step (see figure 3.11) there were 59 couples formed (compared to 49 from the previous simulation - see figure 3.7).

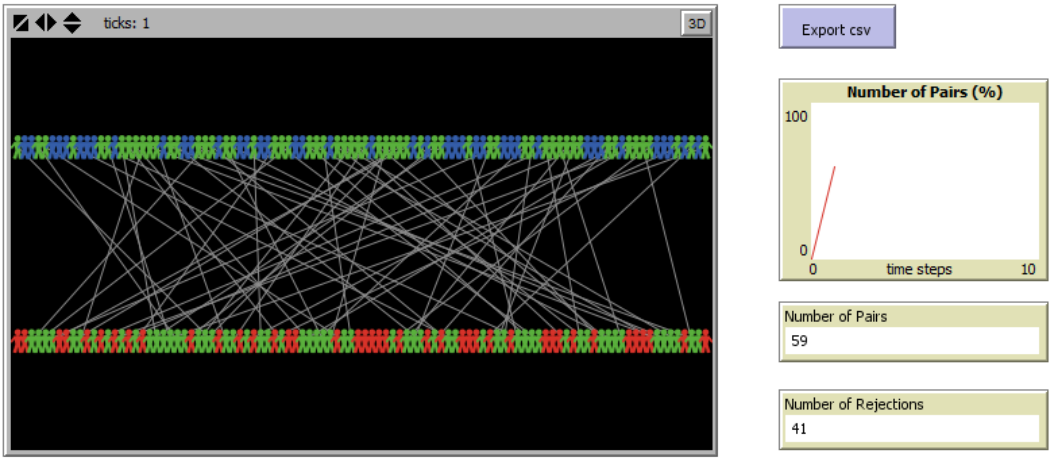

Figure 3.11: Simulation 3, step 1

After the third step there were 79 (compared to 73 from the previous simulation) and after the ninth step there were 90 (compared to 83) couples. Finally the model reached a stable state after 118 steps, where 100 couples were formed. Another observation with the new dataset was, that the number of rejections was much lower than with the other data set. In the end there were 465 rejections (see figure 3.12) compared to 665 from the previous run (see figure 3.10).

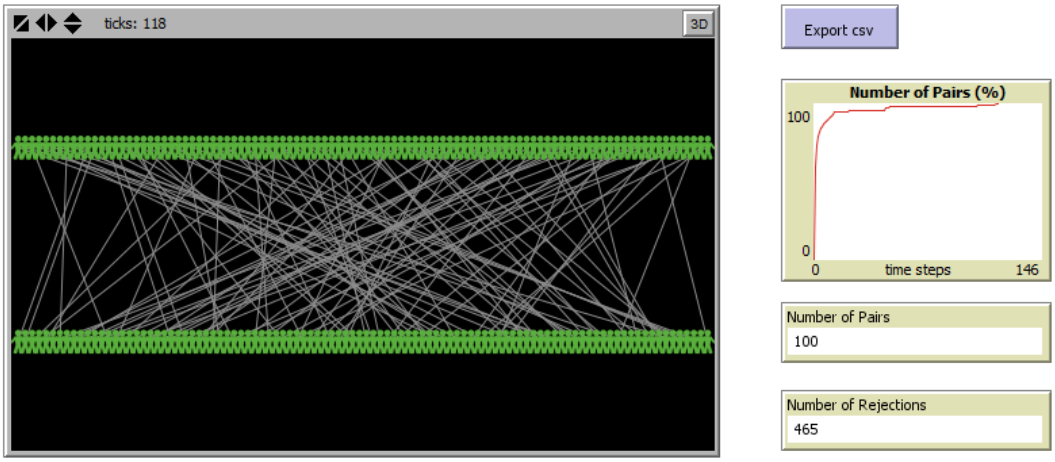

Figure 3.12: Simulation 3, step 118

Prototype Evaluation: College Applications in Austria

Subsection

Text...

Prototype Evaluation: Labour Market in the European Union

Subsection

Text...

Summary

Subsection

Text...

Appendix

Discoteque code

```
1  extensions [array]
2  extensions [string]
3  breed [humans human]
4
5  ; used for drawing a line between the couples
6  undirected-link-breed [ pairs pair ]
7
8  ; humans are turtles (common breed)
9
10 ; NetLogo is case-insensitive
11
12 humans-own [
13   id
14   name
15   maxMatchesInt
16   sideInt
17   partnerList
18   rankList
19   hasProposedToList
20   gotProposedByList
21   tmpMatchList
22   activeFlag
23 ]
24
25 ;;global variables
26 globals [
27   csv fileList ; fileList named csv
28   startSideInt ; set in setup-globals from GUI-slider
29   switchingFlag ; set in setup-globals from GUI-switch
30   debugFlag ; set in setup-globals from GUI-switch
31   user-input-filename ;
32   input-filename; set default in setup-globals or from GUI-button
33   current_nr_of_pairs
34   current_nr_of_pairs_percent
35   current_nr_of_rejects
36 ]
37
38 ;; method which is called from the setup button
39 to setup
40   clear-all
41   if debugFlag = true [
42     show "after clear-all"
43     show "clear-all"
```

```
44      show count humans]
45    setup-globals
46    if debugFlag = true [show "after open-file"
47       show "count humans after open-file"
48       show count humans]
49    reset-ticks
50  end
51
52  to reset
53    clear-ticks
54    clear-turtles
55    clear-patches
56    clear-drawing
57    clear-all-plots
58    clear-output
59    if debugFlag = true [show "after clear-all"
60       show "clear-all"
61       show count humans]
62    setup-globals
63    if debugFlag = true [show "after open-file"
64       show "count humans after open-file"
65       show count humans]
66    reset-ticks
67  end
68
69
70  ;; stackoverflow.com/questions/27096948/how-to-read-a-csv-filve-with
       -netlogo
71  to open-file
72    file-open (word input-filename ".csv")
73    set fileList []
74
75    while [not file-at-end?] [
76      set csv file-read-line
77      set csv word csv ";"  ; add comma for loop termination
78
79      let mylist []  ; list of values
80      let i 0
81      while [not empty? csv]
82      [
83        let $x position ";" csv
84        if i > 0 [
85          let $item substring csv 0 $x  ; extract item
86          carefully [set $item read-from-string $item][] ; convert if
              number
87          set mylist lput $item mylist  ; append to list
88        ]
```

```
89        set csv substring csv ($x + 1) length csv  ; remove item and
              comma
90        set i i + 1
91      ]
92      if debugFlag = true [show mylist]
93      if item 0 mylist != "id"[
94        create-humans 1 [
95          set id item 0 mylist
96          set name item 1 mylist
97          set maxMatchesInt item 2 mylist
98          set sideInt item 3 mylist
99          let tmpPartnerListString string:split-string item 4 mylist
                "#"
100         set partnerList read-from-string (word tmpPartnerListString)
101         ;     set partnerList string:split-int item 4 mylist "#"
102         let tmpRankListString string:split-string item 5 mylist "#"
103         set rankList read-from-string (word tmpRankListString)
104         ;     set rankList string:split-int item 5 mylist "#"
105         set hasProposedToList []
106         set gotProposedByList []
107         set tmpMatchList []
108         set activeFlag true
109       ]
110     ]
111     set fileList lput mylist fileList
112     if debugFlag = true [show "count humans at end of open-file"
113      show count humans
114      show "fileList at end of open-file"
115      show fileList]
116   ]
117   file-close
118 end
119
120 to setup-globals
121   ifelse starter = "Men" [set startSideInt 1] [set startSideInt 2]
122   set debugFlag debug
123   set switchingFlag switching
124   ifelse user-input-filename = 0 [set input-filename "
          disco100NotPicky"] [set input-filename user-input-filename]
125   set current_nr_of_rejects 0
126   set current_nr_of_pairs 0
127   set current_nr_of_pairs_percent 0
128   let current_world_width world-width
129
130   open-file ; and read initialisation data from csv file
131   ; define starting position and start color
132   let number_women count humans with [sideInt = 2]
```

```
133   let xposHumansStart current_world_width / 2 ; starting position
        for humans
134   let i 1
135   foreach sort humans with [sideInt = startSideInt] [
136     ask ? [
137       let number_starting count humans with [sideInt = startSideInt]
138       set shape "person"
139       set size 1
140       set heading 0
141       ; positioning and color
142       set ycor 4
143       ifelse sideInt = 1 [set color blue] [set color red]
144       let xposHumans i / (number_starting + 1) * current_world_width
            - xposHumansStart
145       set xcor xposHumans
146       set i i + 1
147     ]
148   ]
149   set i 1
150   foreach sort humans with [sideInt != startSideInt] [
151     ask ? [
152       let number_notstarting count humans with [sideInt !=
            startSideInt]
153       set shape "person"
154       set size 1
155       set heading 0
156       ; positioning and color
157       set ycor -4
158       ifelse sideInt = 1 [set color blue] [set color red]
159       let xposHumans i / (number_notstarting + 1) *
            current_world_width - xposHumansStart
160       set xcor xposHumans
161       set i i + 1
162     ]
163   ]
164 end
165
166 to go
167   if count humans with [activeFlag = true and (sideInt =
        startSideInt or switchingFlag = true)] = 0 [stop]
168   step
169 end
170
171
172
173 to step
174   if debugFlag = true [show "---------------- begin of step
        ----------------"
```

```
175       show "count humans at begin of step"
176       show count humans]
177     clear-before-match
178     foreach sort humans with [activeFlag = true and sideInt =
          startSideInt] [ ; start of proposing
179       ask ? [
180         if debugFlag = true [show "count humans at begin of ask humans
              with activeFlag=true and sideInt=startSideInt"
181           show count humans with [activeFlag = true and sideInt =
                startSideInt]
182           show "myId"
183           show id
184           show "myName"
185           show name
186           show "partnerList"
187           show partnerList
188           show "hasProposedToList"
189           show hasProposedToList]
190         let tmpPotentialPartnersList list-difference partnerList
              hasProposedToList ; set-difference of partnerList \
              hasProposedToList
191         if length tmpPotentialPartnersList = 0 [
192           set activeFlag false ; this human has no potential partners
                to propose to
193           stop ; break
194         ]
195         if length tmpMatchList >= maxMatchesInt [
196           set activeFlag false ; this human has enough current matches
197           stop
198         ]
199         let myPreferredPartner item 0 tmpPotentialPartnersList ; most
              preferred partner from tmp...List
200         propose-to id myPreferredPartner
201       ]
202     ] ; end of proposing
203     if debugFlag = true [show "end of proposing"]
204     foreach sort humans with [length gotProposedByList > 0 and sideInt
          != startSideInt] [
205       ask ? [
206         process-proposals id
207       ]
208     ]
209     if switchingFlag = true [set startSideInt ((startSideInt + 1) mod
          2)]
210     if startSideInt = 0 [set startSideInt 2]
211      if debugFlag = true [show "switchingFlag"
212        show switchingFlag
213        show "startSideInt"
```

```
214       show startSideInt]
215     calc-stats
216     tick
217     if debugFlag = true [show "############### end of step
          ###############"]
218   end
219
220   to calc-stats
221     set current_nr_of_pairs 0
222     ask humans with [sideInt = startSideInt] [
223       if length tmpMatchList = maxMatchesInt [
224         set current_nr_of_pairs current_nr_of_pairs + 1
225       ]
226     ]
227    let countedHumans count humans with [sideInt = startSideInt]
228     ifelse countedHumans = 0 [
229       set current_nr_of_pairs_percent 0
230     ] [
231       set current_nr_of_pairs_percent current_nr_of_pairs /
            countedHumans
232       set current_nr_of_pairs_percent current_nr_of_pairs_percent *
            100
233       if current_nr_of_pairs = 0 [set current_nr_of_pairs_percent 0]
234     ]
235     if debugFlag = true [ show "Current Nr of Pairs"
236      show current_nr_of_pairs
237      show "Current Nr of Pairs Percent"
238      show current_nr_of_pairs_percent
239    ]
240   end
241
242   to clear-before-match
243     ask humans [
244       set gotProposedByList [] ; delete gotProposedbyList (in
            preparation for next matching-round)
245     ]
246   end
247
248   to propose-to[sender receiver]
249     if debugFlag = true [show "---------------- begin of propose-to
          ----------------"]
250     ask humans with [id = sender] [
251       set hasProposedToList lput receiver hasProposedToList
252       if debugFlag = true [show "hasProposedToList"
253         show hasProposedToList
254         show "receiver"
255         show receiver]
256       ask humans with [id = receiver] [
```

```
257         set gotProposedByList lput sender gotProposedByList
258         if debugFlag = true [show "female side"
259           show "gotProposedByList"
260           show gotProposedByList]
261       ]
262     ]
263     if debugFlag = true [show "############### end of propose-to
          ###############"]
264   end
265
266   to process-proposals [tmpId]
267     if debugFlag = true [show "----------------- begin of process-
          proposals -----------------"
268       show "count humans at begin of process-proposals"
269       show count humans]
270     ask humans with [id = tmpId] [
271       if debugFlag = true [show "gotProposedByList"
272         show gotProposedByList
273         show "sideInt"
274         show sideInt
275         show "maxMatchesInt"
276         show maxMatchesInt]
277       let tmpPotentialCoupleList []
278       ifelse length tmpMatchList = 0 [
279         set tmpPotentialCoupleList gotProposedByList
280       ] [
281       set tmpPotentialCoupleList list-union-set tmpMatchList
            gotProposedByList
282       ]
283       if debugFlag = true [show "tmpPotentialCoupleList"
284         show tmpPotentialCoupleList]
285       let tmpRejectList []
286       let tmpCoupleList []
287       ifelse length tmpPotentialCoupleList > maxMatchesInt [
288         set tmpCoupleList list-order tmpPotentialCoupleList rankList
              partnerList maxMatchesInt
289         if debugFlag = true [show "has more proposals than willing to
              accept"
290           show "tmpCoupleList"
291           show tmpCoupleList]
292         set tmpRejectList list-difference tmpPotentialCoupleList
              tmpCoupleList
293       ] [
294       set tmpCoupleList tmpPotentialCoupleList
295       ]
296       if length tmpRejectList > 0 [
297         reject-proposals id tmpRejectList
298       ]
```

```
299       if length tmpCoupleList > 0 [
300         create-tmpCouples id tmpCoupleList
301       ]
302     ]
303     if debugFlag = true [show "############### end of process-
          proposals ###############"]
304   end
305
306   to reject-proposals [tmpId rejectList]
307     if debugFlag = true [show "----------------- begin of reject-
          proposals -----------------"]
308     let rejecter 0
309     ask humans with [id = tmpId] [
310       set rejecter self
311       set tmpMatchList  list-difference tmpMatchList rejectList
312       if debugFlag = true [show "tmpMatchList"
313          show tmpMatchList]
314       foreach rejectList [
315         ask humans with [id = ?] [
316           set tmpMatchList  list-difference tmpMatchList lput tmpId []
317           set activeFlag true
318           ifelse sideInt = 1 [set color blue] [set color red]
319           set current_nr_of_rejects current_nr_of_rejects + 1
320           remove-link self rejecter
321         ]
322         ifelse sideInt = 1 [set color blue] [set color red]
323       ]
324     ]
325     if debugFlag = true [show "############### end of reject-proposals
          ###############"]
326   end
327
328   to create-tmpCouples [tmpId acceptList]
329     if debugFlag = true [show "----------------- begin of create-
          tmpCouples -----------------"]
330     ask humans with [id = tmpId] [
331       set tmpMatchList acceptList
332       if debugFlag = true [show "tmpMatchList"
333          show tmpMatchList]
334       foreach acceptList [
335         ask humans with [id = ?] [
336           set tmpMatchList  list-union-set tmpMatchList lput tmpId []
337           set color green
338           create-link-with myself
339         ]
340         set color green
341       ]
342     ]
```

```
343     if debugFlag = true [show "############### end of create-
            tmpCouples ###############"]
344   end
345
346
347   to-report list-order [listToOrder ranking partners maxMatches]
348     if debugFlag = true [show "---------------- begin of list-order
            ----------------"]
349     set listToOrder list-overlap listToOrder partners
350     if debugFlag = true [show "listToOrder"
351        show listToOrder]
352     if length listToOrder > maxMatches [
353       let tmpRanking get-rating-of-list partnerList listToOrder
              ranking
354       let tmpList listToOrder
355       if debugFlag = true [show "tmpRanking"
356          show tmpRanking
357          show "tmpList"
358          show tmpList]
359       set listToOrder []
360       let i 0
361       loop [
362         if i >= maxMatches [report listToOrder]
363         let j position (max tmpRanking) tmpRanking
364         if debugFlag = true [show "j"
365            show j
366            show "item j tmpList"
367            show item j tmpList
368            show "tmpList"
369            show tmpList]
370         set listToOrder lput item j tmpList listToOrder
371         set tmpRanking remove-item j tmpRanking
372         set tmpList remove-item j tmpList
373         if debugFlag = true [show "listToOrder"
374            show listToOrder
375            show "tmpList"
376            show tmpList]
377         set i i + 1
378       ]
379
380     ]
381     report listToOrder
382     if debugFlag = true [show "############### end of list-order
            ###############"]
383   end
384
385   ;; symmetrical difference: fullList \ toRemoveList
386   to-report list-difference [fullList toRemoveList]
```

```
387   report filter [not member? ? toRemoveList] fullList
388 end
389
390 ;; intersection of listA and listB
391 to-report list-overlap [listA listB]
392   if debugFlag = true [show "############### begin of list-overlap
          ###############"
393     show listA
394     show listB
395     show filter [member? ? listB] listA]
396   if debugFlag = true [show "############### end of list-overlap
          ###############"]
397   report filter [member? ? listB] listA
398 end
399
400 ;; union of listA and listB
401 to-report list-union-set [listA listB]
402   let tmpList list-difference listB listA
403   foreach tmpList [ set listA lput ? listA]
404   report listA
405 end
406
407
408 ;; gets ranking associated with fullList for partialList
409 to-report get-rating-of-list [fullList partialList ranking]
410   let tmpList []
411   foreach partialList [
412     set tmpList lput item (position ? fullList) ranking tmpList
413   ]
414   report tmpList
415 end
416
417
418
419 ;;;;;;;;;;;;;;;;;;;;;;;;
420 ;;; Input filename   ;;;
421 ;;;;;;;;;;;;;;;;;;;;;;;;
422
423 to change-input
424   set user-input-filename user-input "Type input filename (without
          extension)."
425   if debugFlag = true [show "user-input-filename"
426     show user-input-filename]
427 end
428
429
430
431
```

```
;;;;;;;;;;;;;;;;;;;;;;
;;; Output matches   ;;;
;;;;;;;;;;;;;;;;;;;;;;

;; write-to-file taken from
;; Wilensky, U. (1999). NetLogo. http://ccl.northwestern.edu/netlogo/.
;; Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL
;; code from File/Models Library/Code Examples/File Output Example

to export-to-csv
  let output-filename (word input-filename "_Starter_" starter "_Switch_" switching "_Ticks_" ticks ".csv")
  foreach sort humans [
    ask ? [
      let delimPartnerList list-concat-with-delim partnerList "#"
      let delimRankList list-concat-with-delim rankList "#"
      let delimHasProposedToList list-concat-with-delim hasProposedToList "#"
      let delimGotProposedByList list-concat-with-delim gotProposedByList "#"
      let delimTmpMatchList list-concat-with-delim tmpMatchList "#"
      write-csv output-filename (list (id) (name) (maxMatchesInt) (sideInt) (delimPartnerList) (delimRankList) (delimHasProposedToList) (delimGotProposedByList) (delimTmpMatchList) (activeFlag))
    ]
  ]
end


;; http://stackoverflow.com/questions/22462168/netlogo-export-tableau-issues
to write-csv [ #filename #items ]
  ;; #items is a list of the data (or headers!) to write.
  if is-list? #items and not empty? #items
  [ file-open #filename
    ;; quote non-numeric items
    set #items map quote #items
    ;; print the items
    ;; if only one item, print it.
    ifelse length #items = 1 [ file-print first #items ]
    [file-print reduce [ (word ?1 ";" ?2) ] #items]
    ;; close-up
    file-close
  ]
```

```
471 end
472
473 to remove-link [human1 human2]
474   ask links with [(end1 = human1 and end2 = human2) or (end1 =
        human2 and end2 = human1)] [
475     die]
476 end
477 ;; http://stackoverflow.com/questions/22462168/netlogo-export-
      tableau-issues
478 to-report quote [ #thing ]
479   ifelse is-number? #thing
480   [ report #thing ]
481   [ report (word "\"" #thing "\"") ]
482 end
483
484 ;; https://groups.yahoo.com/neo/groups/netlogo-users/conversations/
      topics/6490
485 ;; intersperse listA with delim
486 to-report list-concat-with-delim [listA delim]
487   if length listA > 0 [report reduce [(word ?1 delim ?2)] listA]
488   report ""
489 end
```

List of Figures

References

[1] David Gale and Lloyd S. Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[2] Nicholas Barr. *Economics of the Welfare State*. Oxford University Press, 2012.

[3] Colin Camerer. *Behavioral game theory: Experiments in strategic interaction*. Princeton University Press, 2003.

[4] Robert Gibbons. An introduction to applicable game theory. *The Journal of Economic Perspectives*, 11:127–149, 1997.

[5] Benedikt Fuest. Frequenzauktion droht zum mini-geschaeft zu werden, 2015.

[6] Alvin E Roth and Marilda A Oliveira Sotomayor. *Two-sided matching: A study in game-theoretic modeling and analysis*. Number 18. Cambridge University Press, 1992.

[7] Brian Christian and Tom Griffiths. *Algorithms to Live By: The Computer Science of Human Decisions*. Henry Holt and Co., 2016.