py-school-match Documentation

Iacopo Garizio

Feb 04, 2019

First steps

Py-school-match is a Python library designed to help researchers compare different matching algorithms to assign students to schools.

Why? Different algorithms have different strengths and weaknesses regarding efficiency, fairness and other theoretical characteristics. Because of this, the algorithms should be evaluated in simulations in order to asses these characteristics.

CHAPTER 1

First steps

1.1 Installation guide

1.1.1 Installing py-school-match

Ubuntu

Py-school-match uses graph-tool for most algorithms. You can install it by following this guide.

After this, you can simply install py-school-match with `pip`:

```
pip install py-school-match
```

Or by doing:

```
git clone https://github.com/igarizio/py-school-match
cd py-school-match
python setup.py install
```

Windows

Windows is not currently supported, but if you are still interested, I recommend using Windows Subsystem for Linux (read this guide) and then configure remote interpreters via SSH (if you are using Pycharm, you can read this this guide).

1.2 Getting started

1.2.1 Py-school-match's basic structure

Py-school-match tries to expose flexible and easy-to-understand structures and functions.

In order to use the library, you should create at least one element of each category:

- `Student`: Represents one student.
- `School`: Represent one school. A school can have many seats.
- `SocialPlanner`: This entity runs the selected algorithm.

You should also select one algorithm:

- `TTC`: Top trading cycles.
- `DAMTB`: Deferred acceptance with multiple tie-breaking.
- `DASTB`: Deferred acceptance with single tie-breaking.
- `SIC`: Stable improvement cycles.
- `MSIC`: Deferred Acceptance with multiple tie-breaking, and then searches for stable cycles.
- `NSIC`: Deferred Acceptance with single tie-breaking, and then searches for non-stable cycles.

Finally, you can also define:

- `Criteria`: Associates an adjective and a value type.
- `Characteristic`: Associates a criteria with a value.
- `Rule`: Is used to determine the priority of a student.
- `Ruleset`: Aggregates a set of rules in a prioritized manner.

1.3 Examples

1.3.1 Basic example

The following example simulates three students (with one marked as `vulnerable`) and three schools (who prioritize `vulnerable` students). The code runs `SIC` (Stable Improvements Cycles) as an example, but it can be replaced with any of the stated algorithms.

```
# Importing py-school-match
import py_school_match as psm

# Creating three students.
st0 = psm.Student()
st1 = psm.Student()
st2 = psm.Student()

# Creating a criteria. This means 'vulnerable' is now a boolean.
vulnerable = psm.Criteria('vulnerable', bool)

# Assigning st1 as vulnerable
student_vulnerable = psm.Characteristic(vulnerable, True)
st1.add_characteristic(student_vulnerable)

# Creating three schools, each with one seat available.
sc0 = psm.School(1)
sc1 = psm.School(1)
sc2 = psm.School(1)

# Defining preferences (from most desired to least desired)
```

(continues on next page)

(continued from previous page)

```
st0.preferences = [sc0, sc1, sc2]
st1.preferences = [sc0, sc2, sc1]
st2.preferences = [sc2, sc1, sc0]

# Creating a lists with the students and schools defined above.
schools = [sc0, sc1, sc2]
students = [st0, st1, st2]

# Defining a ruleset
ruleset = psm.RuleSet()

# Defining a new rule from the criteria above.
rule_vulnerable = psm.Rule(vulnerable)

# Adding the rule to the ruleset. This means that a 'vulnerable' student has a higher␣
↪priority.
# Note that rules are added in order (from higher priority to lower priority)
ruleset.add_rule(rule_vulnerable)

# Creating a social planner using the objects above.
planner = psm.SocialPlanner(students, schools, ruleset)

# Selecting an algorithm
algorithm = psm.SIC()

# Running the algorithm.
planner.run_matching(algorithm)

# inspecting the obtained assignation
for student in students:
    if student.assigned_school is not None:
        print("Student {} was assigned to School {}".format(student.id, student.
↪assigned_school.id))
    else:
        print("Student {} was not assigned to any school".format(student.id))
```

1.3.2 Example using quotas

The following example simulates three students (with one marked as `vulnerable`) and three schools (who prioritize `vulnerable` students). This time, a minimum quota of 50% of vulnerable students is required. The code runs `SIC` (Stable Improvements Cycles) as an example, but it can be replaces with any of the stated algorithms.

```
# Importing py-school-match
import py_school_match as psm

# Creating three students.
st0 = psm.Student()
st1 = psm.Student()
st2 = psm.Student()

# Creating a criteria. This means 'vulnerable' is now a boolean.
vulnerable = psm.Criteria('vulnerable', bool)

# Assigning st1 as vulnerable
student_vulnerable = psm.Characteristic(vulnerable, True)
```

(continues on next page)

(continued from previous page)

```
st1.add_characteristic(student_vulnerable)

# Creating three schools, each with one seat available.
sc0 = psm.School(1)
sc1 = psm.School(1)
sc2 = psm.School(1)

# Defining preferences (from most desired to least desired)
st0.preferences = [sc0, sc1, sc2]
st1.preferences = [sc0, sc2, sc1]
st2.preferences = [sc2, sc1, sc0]

# Creating a lists with the students and schools defined above.
schools = [sc0, sc1, sc2]
students = [st0, st1, st2]

# Defining a ruleset
ruleset = psm.RuleSet()

# Defining a new rule from the criteria above.
# This time, a flexible quota is imposed.
# This means that each school should have at least 50% percent
# vulnerable students. The "flexible" part means that if there are
# no vulnerable students left, even if the quota is not met, the
# school can now accept non-vulnerable students.
rule_vulnerable = psm.Rule(vulnerable, quota=0.5)

# Adding the rule to the ruleset. This means that a 'vulnerable' student has a higher␣
↪priority.
# Note that rules are added in order (from higher priority to lower priority)
ruleset.add_rule(rule_vulnerable)

# Creating a social planner using the objects above.
planner = psm.SocialPlanner(students, schools, ruleset)

# Selecting an algorithm
algorithm = psm.SIC()

# Running the algorithm.
planner.run_matching(algorithm)

# inspecting the obtained assignation
for student in students:
    if student.assigned_school is not None:
        print("Student {} was assigned to School {}".format(student.id, student.
↪assigned_school.id))
    else:
        print("Student {} was not assigned to any school".format(student.id))
```

1.3.3 Comparing algorithms

The following example simulates the same conditions for two different algorithms. This allows a direct comparison of the results.

```
# Importing py-school-match
import py_school_match as psm

# Defining a list of algorithms
algorithms = [psm.TTC, psm.DAMTB]

# Simple dictionary to store the results
results = {}

# Iterating over each algorithm and defining the conditions
for algorithm in algorithms:

    random.seed(42)

    vulnerable = psm.Criteria("vulnerable", bool)

    st0 = psm.Student()
    st1 = psm.Student()
    st2 = psm.Student()
    st3 = psm.Student()

    st1.add_characteristic(psm.Characteristic(vulnerable, True))

    students = [st0, st1, st2, st3]

    sc0 = psm.School(1)
    sc1 = psm.School(1)
    sc2 = psm.School(1)

    schools = [sc0, sc1, sc2]

    st0.preferences = [sc0, sc1, sc2]
    st1.preferences = [sc0, sc2, sc1]
    st2.preferences = [sc2, sc1, sc0]
    st3.preferences = [sc0, sc1, sc2]

    ruleset = psm.RuleSet()
    rule_vulnerable = psm.Rule(vulnerable)
    ruleset.add_rule(rule_vulnerable)

    planner = psm.SocialPlanner(students, schools, ruleset)

    # Running each algorithm
    planner.run_matching(algorithm())

    # Storing the results in the dictionary.
    # Note that ``get_positions_stat`` takes the SocialPlanner object and returns
    # a dictionary with the following format: {position: number of students}
    # For example, {1: 25, 2:14, 'NA': 5} means that 25 students were assigned to
    # their most preferred school, 14 to their second-most preferred school
    # and 5 were not assigned no any school.
    results[algorithm.__name__] = get_positions_stat(planner)

print(results)
```

1.3.4 Visualizing algorithms

> **Warning:** Experimental code.

In iterative algorithms you can visualize each iteration.

In order to generate images, simply add `generate_images=True` to the algorithm definition. See the following example:

```
algorithm = psm.SIC(generate_images=True)
planner.run_matching(algorithm)
```

Note that if an algorithm does not find any cycle or cannot make any iteration, no image will be created.

1.4 Testing guide

1.4.1 Testing py-school-match

To run the test suite, simply run:

```
python3 -m unittest discover
```

1.5 Detailed documentation

1.5.1 py_school_match package

Subpackages

py_school_match.algorithms package

Submodules

py_school_match.algorithms.abstract_matching_algorithm module

py_school_match.algorithms.da module

py_school_match.algorithms.da_mtb module

py_school_match.algorithms.da_stb module

py_school_match.algorithms.mnsic module

py_school_match.algorithms.msic module

py_school_match.algorithms.nsic module

py_school_match.algorithms.pi module

py_school_match.algorithms.sic module

py_school_match.algorithms.ttc module

Module contents

py_school_match.entities package

Submodules

py_school_match.entities.characteristic module

py_school_match.entities.criteria module

py_school_match.entities.rule module

py_school_match.entities.ruleset module

py_school_match.entities.school module

py_school_match.entities.slot module

py_school_match.entities.social_planner module

py_school_match.entities.student module

Getting started Learn about py-school-match's structure.

Examples Learn to use py-school-match by example.

Testing guide Test your py-school-match installation.