Student-University Matching Tool

Developer Guide

January 30, 2026

Contents

1 Architecture Overview

The Student-University Matching Tool is a pure frontend application with no backend dependencies. The architecture consists of:

- **index.html:** Main HTML structure and UI layout
- **style.css:** Styling and responsive design
- **script.js:** Core application logic, data processing, and algorithm interface
- **libs/:** External libraries (solver, CSV/Excel parsers)

This is all based on a bachelor thesis project, see https://purl.utwente.nl/essays/100794 for more details.

1.1 Hosting and Deployment

The application is hosted at the University of Twente server:

https://nsh-01.utsp.utwente.nl

All files can overwritten and are immediately live. No build step is required (except for LaTeX)

1.2 Data Flow

1. User uploads CSV/Excel files
2. Files are parsed using PapaParse (CSV) or XLSX (Excel)
3. Data is validated and normalized into standardized formats
4. Global data structures store universities (`idToUni`) and students (`idToStud`)
5. Linear programming model is built
6. Solver (javascript-lp-solver) finds optimal assignment
7. Results are displayed and can be exported to CSV

2 Global Data Structures

2.1 idToUni

Maps agreement IDs to university objects:

```
idToUni[agreementId] = {
  agreement_ID: number,
  partner_name: string,
  study_field: string,
  study_abbr: string,
  total_places: number,
  max_BCs: number,           // Max BSc students
  max_MCs: number,           // Max MSc students
  spots_first_s: number,     // Semester 1 capacity
  spots_second_s: number,    // Semester 2 capacity
  max_BMS: number,           // Faculty constraints
  max_EEMCS: number,
```

```
  max_ET: number,
  max_ITC: number,
  max_ST: number,
  max_UCT: number,
  key: string                // Unique identifier
}
```

2.2 idToStud

Maps student IDs to student objects:

```
idToStud[studentId] = {
  stud_id: number,
  student_no: string,
  name: string,
  study_level: string,
  semester: number,
  study_field: string,
  all_preferences: [id1, id2, id3, ...] // Agreement IDs
}
```

2.3 manualAssignments and forbiddenAssignments

```
manualAssignments = [
  { studentId: 123, agreementId: 666, choice: 1 },
  ...
]

forbiddenAssignments = [
  { studentId: 123, agreementId: 666 },
  ...
]
```

3 Key Functions

3.1 File Processing

3.1.1 parseFile(file)

Detects file type (.xlsx, .xls, or .csv) and calls the appropriate parser. Returns a Promise resolving to an array of row objects.

3.1.2 validateUniversitiesRawData(rows)

Checks for required columns (ID, Abbr. of study field) and at least one capacity field. Returns {`ok: boolean, errors: []`}.

3.1.3 validateApplicantsRawData(rows)

Checks for required columns (ID of application, Abbreviation of study field, Study field, Study level, Semester). Returns validation object.

3.1.4 processUniversitiesData()

Transforms raw CSV data into the standardized `idToUni` structure. Handles:

- Aggregating agreements by ID and study field
- Creating special dummy universities for unmatched students
- Extracting semester capacities
- Mapping faculty constraints

3.1.5 processApplicantsData()

Transforms raw CSV data into the standardized `idToStud` structure. Handles:

- Extracting student preferences from Agreement-ID columns
- Converting study levels (BSc/MSc) to numeric identifiers
- Parsing semester information
- Mapping students by ID

3.2 Data Validation

3.2.1 cleanupStudentPreferences()

Cleans up student preference arrays by:

- Removing preferences for non-existent universities
- Eliminating duplicate preferences
- Padding arrays with `null` to maintain consistent size (6 preferences)

3.2.2 checkManualAssignmentsFeasibility()

Validates that manual assignments don't violate:

- University capacity constraints
- Faculty-specific capacity constraints
- Duplicate student assignments
- Forbidden assignment rules

3.3 Algorithm: Building the Linear Programming Model

3.3.1 buildModel(prefWeights = [1,2,3,4,5,6])

Constructs a linear programming model for the javascript-lp-solver library. The model minimizes total assignment cost while respecting all constraints.

Variables: For each student-university-semester pair:

```
x_s_u_sem = {
  obj: weight,  // Preference weight (1-6)
  student_sem_id: 1,
  cap_ag_sem_agreementId: 1,
  cap_u_sem_uniqueKey: 1,
  cap_u[level]_sem_uniqueKey: 1
}
```

Constraints:

1. **Student assignment:** Each student assigned to exactly one university
2. **Agreement capacity:** Total assignments $\leq$ agreement capacity
3. **Study level capacity:** BSc/MSc assignments $\leq$ respective limits
4. **Faculty constraints:** Faculty-specific seat limits (if defined)
5. **Forbidden pairs:** Forbidden student-university assignments set to 0
6. **Manual assignments:** Pre-assigned student-university pairs fixed to 1

3.3.2 buildVar(weight, studID, uni, sem, stud)

Creates a variable object representing one possible student-university assignment with:

- Preference weight for the objective function
- Constraint coefficients for capacity checks
- Faculty constraint references (if applicable)

3.4 Result Processing

3.4.1 displayResults(results)

Parses the solver output and generates:

- Summary statistics (choice distribution, unmatched count, average preference)
- Sortable results table with student-university pairings
- List of unmatched students
- Export button for CSV download

3.4.2 exportAssignmentsToCSV(assignments)

Creates and downloads a CSV file with all assignment details including:

- Student identifiers
- Assigned university
- Preference ranking achieved
- Study program and level
- Faculty and agreement type

4 Modifying the Application

4.1 Adding a New Constraint

To add a new university capacity constraint (e.g., by agreement type):

1. Add a field to the university data structure in `processUniversitiesData()`
2. Create a constraint in `buildModel()` following the existing pattern
3. Update `checkManualAssignmentsFeasibility()` to validate the constraint

Example: Adding agreement-type-specific limits

```
// In processUniversitiesData():
max_exchange_i: r['Max Exchange-I Students'],

// In buildModel():
const typeKey = 'cap_type_' + sem + '_' + u.agreement_type;
model.constraints[typeKey] = {max: u.max_exchange_i};
// In buildVar():
if (u.max_exchange_i) v[typeKey] = 1;
```

4.2 Changing Preference Weights

Modify the default weights in the HTML and `getPreferenceWeights()` function:

```
// In index.html, change the value attributes:
<input type="number" id="weight_1" value="1">

// In getPreferenceWeights(), update defaults:
const defaults = [1, 2, 3, 4, 5, 6];
```

4.3 Modifying File Format Requirements

1. Update column name expectations in data parsing functions
2. Update validation functions to check for new required columns
3. Update the USER_GUIDE.tex file

4.4 Adding Manual Assignment Logic

The manual assignment system is controlled by:

- `setupStudentSearch()`: Handles UI interactions
- `showStudentPreferences()`: Displays student options
- `updateManualAssignmentsList()`: Updates display table
- `checkManualAssignmentsFeasibility()`: Validates constraints

To modify enforcement: edit `checkManualAssignmentsFeasibility()` to return specific error messages, then update the UI handlers to display them.

4.5 Modifying the Solver

The application uses `javascript-lp-solver` library. The solver object expects:

```
{
  optimize: 'obj' | 'min' | 'max',
  opType: 'min' | 'max',
  constraints: {...},
  variables: {...}
}
```

To use a different solver, replace the `solver.js` library and update the call site in `displayResults()`.