Article

Automated Test Assembly in R: The eatATA Package

Benjamin Becker [1,*,†], Dries Debeer [2,3,†], Karoline A. Sachse [1] and Sebastian Weirich [1]

[1] Institute for Educational Quality Improvement, Humboldt University Berlin, 10117 Berlin, Germany; karoline.sachse@iqb.hu-berlin.de (K.A.S.), sebastian.weirich@iqb.hu-berlin.de (S.W.)
[2] Faculty of Psychology and Educational Sciences, KU Leuven, 3000 Leuven, Belgium; dries.debeer@kuleuven.be
[3] Imec Research Group Itec, KU Leuven, 3000 Leuven, Belgium
* Correspondence: b.becker@iqb.hu-berlin.de
† These authors contributed equally to this work.

Abstract: Combining items from an item pool into test forms (test assembly) is a frequent task in psychological and educational testing. Although efficient methods for automated test assembly exist, these are often unknown or unavailable to practitioners. In this paper we present the `R` package `eatATA`, which allows using several mixed-integer programming solvers for automated test assembly in `R`. We describe the general functionality and the common work flow of `eatATA` using a minimal example. We also provide four more elaborate use cases of automated test assembly: (a) The assembly of multiple test forms for a pilot study; (b) the assembly of blocks of items for a multiple matrix booklet design in the context of a large-scale assessment; (c) the assembly of two linear test forms for individual diagnostic purposes; (d) the assembly of multi-stage testing modules for individual diagnostic purposes. All use cases are accompanied with example item pools and commented `R` code.

Keywords: automated test assembly; mixed integer programming; R

check for updates

Citation: Becker, B.; Debeer, D.; Sachse, K.A.; Weirich, S. Automated Test Assembly in R: The eatATA Package. *Psych* **2021**, *3*, 96–112. https://doi.org/10.3390/psych3020010

Academic Editor: Philipp Doebler and Mosad Zineldin

Received: 19 April 2021
Accepted: 17 May 2021
Published: 21 May 2021

1. Theoretical Background

In psychological or educational testing, assembling test forms from an existing item pool is a frequent challenge [1]. Sometimes a single test form is constructed, which has to be maximally informative for a certain classification decision. Sometimes test security is a concern and therefore multiple, parallel test forms are created to prevent test-takers from answer-copying and from sharing test content across test sessions [2]. In large-scale assessments, multiple test forms are used to cover a broader range of test content and to increase measurement precision [3,4]. In multi-stage testing (MST), test modules with different ability target groups are created [5], whereas in computer adaptive testing (CAT), after every item, a new item is added to the test [6]. Because assembling test forms by hand can be cumbersome and error prone, automated test assembly (ATA) methods have been developed, which rely on mathematical programming techniques such as mixed-integer programming (MIP). In many, if not all cases where items should not just be randomly selected, ATA can help, and will likely lead to better solutions than manual test assembly.

In ATA, test specifications, similar to the number of items per test form or item type distributions across test forms, are formulated as mathematical constraints. The optimization goal (for instance, maximizing test information at a certain ability level) is formulated as a mathematical objective function. Mathematical programming solvers can be used to find an optimal solution for the given combination of mathematical constraints and objective, meaning that the optimal item-to-test-form assignment for the test assembly problem can be found. A general introduction to automated test assembly is, for example, available in van der Linden [1].

Unfortunately, in practice, ATA approaches are not utilized as often as they could. Due to conceptual and technical barriers, practitioners frequently opt for manual trial and error approaches instead, leading to sub-optimal solutions. With the `R` package `eatATA` (educational assessment tools: Automated Test Assembly) and this tutorial we try to give easy access to ATA to more practitioners. The paper is structured as follows: First, we give a short introduction to why an `R` package is suitable in this context. We then give an overview of the functionalities of `eatATA` and which solvers are accessible via the package, and illustrate the general work flow when using `eatATA` for automated test assembly with a minimal example. Subsequently, we provide four practical use cases alongside detailed and commented `R` code to illustrate the package functionality in depth.

2. eatATA

In the context of psychological and educational testing, `R` [7] is a common tool for psychometric and statistical analyses. `R` is an open source and free software environment and its extensive and actively maintained libraries offer tools for a rich diversity of data analysis use cases. Furthermore, a variety of mathematical programming solvers are available through `R`, including both open source and commercial solvers. These solvers can usually be accessed via packages that function as APIs (application programming interfaces) to a specific solver, and the solver itself is often included directly in the respective package (e.g., `lpSolveAPI` is an API to the solver `lpSolve`). This, in principal, enables researchers to use `R` for test assembly purposes. For example, a short tutorial on how ATA can be used in `R` using `lpSolveAPI` [8] can be seen in Diao and van der Linden [9]. However, while such an implementation is possible, the translation of test specifications into mathematical constraints can be an interesting challenge for some, but a cumbersome task for others.

The mathematical programming solvers and APIs are often very flexible and applicable to a wide range of optimization problems that go far beyond ATA. Although valuable, this flexibility also increases the complexity for users, especially because the APIs, the solvers, and the available documentation are generally not targeting ATA applications. Hence, for researchers without a background in mathematical programming, applying the solvers to ATA problems is far from straightforward. In addition, there exist considerable differences between the APIs of different solvers. Therefore, when an educational measurement practitioner has invested the time to become familiar with one specific solver API, switching to a different solver will likely require an additional effort. From our experience, we believe that there are two main reasons why ATA methods are highly underused in practice: First, a lot of practitioners are not aware that there are efficient assembly techniques that could be helpful in their daily practice. Second, practitioners that are aware of the efficient assembly techniques often lack the time to work out all the operational details.

Through the `R` package `eatATA` [10] and this tutorial paper we want to promote ATA methods and provide easier access to ATA for measurement practitioners. The package facilitates the access to mathematical programming and its potential for ATA-problems without worrying about how test specifications are formulated mathematically. In the spirit of the `R` programming language, the functionality of the package is based on functions. Every test specification can be expressed by a function, thereby enabling a work flow in `R` that will feel familiar for practitioners with `R` experience.

Available solvers within the package are `GLPK` [11], `lpSolve` [12], `SYMPHONY` [13], and `Gurobi` [14]. These solvers are used via the R package APIs `Rglpk` [15], `lpSolve` [16], `Rsymphony` [17], and `gurobi` [18]. For a general overview of different available open source and commercial MIP solvers see, for example, Donoghue [19] and Luo [20].

2.1. Work Flow

Regardless of the context of a specific test assembly problem, the common challenge when assembling test forms is that a variety of requirements for the resulting test form(s) have to be fulfilled, otherwise known as the test specifications [1]. The schematic work flow of automatically assembling test forms and incorporating the required test specifications using `eatATA` is the following:

1. Item Pool: A `data.frame` including all information on the item pool is loaded or created. If the items have already been calibrated (e.g., based on data from a pilot study) this will include the calibrated item parameters.
2. Test Specifications: Usually a combination of: (a) Typically one objective and (b) multiple constraints.
   (a) Objective Function: Usually a single object corresponding to the optimization goal, created via one of the *objective function* functions. This refers to a test specification where we have no absolute criterion, but where we want to minimize or maximize something.
   (b) Further Constraints: Further constraint objects, created using various constraint functions. These refer to test specifications with a fixed value or an upper and/or lower bound.
3. Solver Call: The `useSolver()` function is called using the constraint objects to find an optimal solution.
4. Solution Processing: The solution can be inspected using the `inspectSolution()` and `appendSolution()` functions.

To illustrate this general work flow, we provide a small illustrative example. More complex extensions are discussed in the specific use cases later in the paper.

2.2. Minimal Example

In this minimal example, the goal is to assemble a single test form with maximum test information function (TIF) for a medium ability level. Furthermore, the test form should consist of exactly ten distinct items and have an average test time of approximately 8 min. The complete R syntax for this example can be found in Supplement S0 and the mathematical formulation of the test assembly problem is described in Appendix A. Further details regarding mathematical formulations of MIP problems and how MIP solvers operate can be found in van der Linden [1].

(1) Item Pool

For the illustrative example, we use a small simulated item pool of 30 items, which is included in the `eatATA` package (`items_mini`). In general, item pool information should be stored in a single `data.frame` with each row representing an item. In the example item pool, items are characterized by their format (“`format`”), average response times (“`time`”), and a difficulty parameter (“`difficulty`”), based on a calibration according to a Rasch model [21]. To calculate the *item information function* (IIF) we use the `calculateIIF()` function (see Figure 1). Alternatively, the `calculateIIF()` function could be used to calculate the IIF for the item parameters from the 2 and 3 parameter logistic models. (In principal, any response model can be used within `eatATA` and there exist various `R` packages to calculate IIFs for a wide range of response models.) We provide the item parameters and one or multiple ability points (`theta`) at which the item information function should be calculated. In our case, we are interested only in the information function at a medium ability, so we set `theta = 0` and append the IIF to our item pool `data.frame`. The resulting first five rows of the item pool can be seen in Table 1.

```
items_mini$IIF_0 <- calculateIIF(B = items_mini$difficulty, theta = 0)
```

Figure 1. Calculate item information function.

(2a) Objective Function

As a first object, we define the objective function. This corresponds to a test specification where we have no absolute criterion that needs to be exactly fulfilled, but where the goal is to minimize or maximize something. This can vary greatly depending on the goal of the test assembly. Common optimization goals include maximizing the TIF of a test form, minimizing test time or test length, or minimizing differences in TIF between test forms.

Table 1. First five items of the simulated item pool.

Item	Format	Time	Difficulty	IIF_0
1	mc	27.79	−1.88	0.11
2	mc	15.45	0.84	0.45
3	mc	31.02	1.12	0.33
4	mc	29.87	0.73	0.50
5	mc	23.13	−0.49	0.61

In our example, we seek to maximize the TIF at a medium ability level using the `maxObjective()` function (see Figure 2). This is achieved via maximizing the sum of the IIFs of the items in the test form. (As a Rasch model has been used for calibration, maximizing the TIF at ability level 0 corresponds to minimizing the difference of the average item difficulty from 0.) Note that item identifiers should be supplied to all objective function and constraint functions. This guarantees that all constraints relate to the same set of items and provides a more readable solver output. Other available functions for defining optimization goals are: `minObjective()`, `maximinObjective()`, `minimaxObjective()`, and `cappedMaximinObjective()`.

```
testInfo <- maxObjective(nForms = 1, itemValues = items_mini$IIF,
                         itemIDs = items_mini$item)
```

Figure 2. Define objective function: Maximize item information function at average ability.

(2b) Constraints

In the next step, we translate our further test specifications for the test assembly into function calls. These constraints are not optimization goals but specifications with fixed target values or upper and/or lower bounds. For this, we create multiple constraint objects (see Figure 3). In our example we want to fix the number of items in the test form to exactly ten items, which is performed by the `itemsPerFormConstraint()` function. This function specifically serves the purpose of setting the test length for the test assembly. Using the `operator` and the `targetValue` arguments we can set a fixed target value or an upper or lower bound. The total test time is constrained to approximately eight minutes using the `itemsValuesDeviationConstraint()` function. This function belongs to a family of functions that can be used to set constraints using numerical item values. By setting `allowedDeviaton = 5` we allow the testing time to vary between 7 min and 55 s and 8 min and 5 s. Note that setting the test time to be exactly eight minutes would be overly restrictive and not necessary from a practical stand point. An overview over the available constraint functions and their functionality can be found here: https://CRAN.R-project.org/package=eatATA/vignettes/overview.html (accessed on 20 May 2021).

(3) Solver Call

Finally, we collect all constraint objects defined above in a `list` and hand these to the solver of our choice via the `useSolver()` function (see Figure 4). The order in which the constraints (including the objective function) are created or ordered does not have any impact on the solution of the test assembly problem.

As complex test assembly problems with large numbers of possible solutions can lead to long computation times it is often reasonable to set a time limit for the solver via the `timeLimit` argument. In cases where the time limit is reached and where at least one feasible solution is found, but the search of the total solution space is incomplete, the function returns the best available solution. In most practical applications the quality of this solution will be absolutely sufficient. As this illustrative example is a very simple ATA problem for which `GLPK` finds a solution almost instantly, it is not necessary to set a time limit for the solver. The solver used for ATA can be specified via the `solver` argument, with ‘‘`GLPK`’’ being the default.

```
# Number of items (test length)
itemNumber <- itemsPerFormConstraint(nForms = 1, operator = "=",
                                     targetValue = 10,
                                     itemIDs = items_mini$item)

# Test time
testTime <- itemValuesDeviationConstraint(nForms = 1,
                                          itemValues = items_mini$time,
                                          targetValue = 8 * 60,
                                          allowedDeviation = 5,
                                          relative = FALSE,
                                          itemIDs = items_mini$item)
```

Figure 3. Define constraints: Number of items in the test form, number of times an item can be used, and total average testing time.

```
solver_out <- useSolver(list(itemNumber, testTime, testInfo),
                        solver = "GLPK")
```

Figure 4. Solve MIP problem.

Note that sometimes combinations of constraints can lead to infeasibility issues. That is, it is possible that for a given set of test specifications for a specific item pool no feasible solution exists. If this is the case, `useSolver()` will issue a corresponding message. To identify which (combination of) constraints causes the infeasibility, it can be helpful to remove constraints from the ATA problem step by step until feasibility is achieved. Alternatively, constraints can be added to the ATA problem step by step starting with just the objective function until the problem becomes infeasible [22].

(4) Solution Processing

`eatATA` provides two functions to process the output of `useSolver()`: `inspectSolution()` to directly view the assembled test forms presented in a list that only contains the items in the assembled test form(s), and `appendSolution()`, which appends the assignment matrix containing 0 (item not in this test form) and 1 (item in this test form) to the item pool `data.frame` (see Figure 5).

```
inspectSolution(solver_out, items = items_mini, idCol = "item")
item_mini_out <- appendSolution(solver_out, items = items_mini,
                                idCol = "item")
```

Figure 5. Inspect the solver solution and append it to the item pool `data.frame`.

3. Use Cases

In the following section we present four different applications of the `eatATA` package to test assembly problems. The use cases were chosen to cover a broad range of contexts for ATA application: (1) A pilot study setting in which we assemble multiple test forms while depleting the item pool (without prior item calibration), (2) a typical large-scale assessment situation, in which calibrated items are assembled to blocks for a multiple matrix booklet design, (3) the assembly of multiple parallel test forms for a high-stakes assessments from a calibrated item pool, and (4) the assembly of modules from a calibrated item pool for a multi-stage assessment. To illustrate the accessibility of `eatATA` compared to plain solver API's use case (3) and (4) correspond to two of the problems used in the tutorial paper by Diao and van der Linden [9]. Because the solver calls and the solution processing do not differ much between the minimal example and the different use cases, we primarily focus on how the constraint and objective function definitions have to be altered from application to application. Complete syntaxes for all uses cases can be found in the corresponding Supplementary Files.

3.1. Pilot Study

Usually, when conducting a pilot study, little is known about the empirical characteristics of the item pool. Instead, the goal of a pilot study is to gather such information (e.g., response times, and missing rates) and calibrate the items. Hence, the test specifications for pilot studies often deviate substantially from test assembly specifications for operational tests. For this use case we use a simulated item pool `items_pilot`, which is included in the `eatATA` package. The item pool consists of 100 items with various characteristics, for example the expected response times in seconds (''`time`''), the item format (''`format`''), and a rough estimate of the item difficulty (''`diffCategory`''), grouped into five categories. The first five items of the item pool can be seen in Appendix B Table A1. From this item pool, we want to assemble test forms that meet the following requirements: (1) each item should appear in exactly one test form (this implies no item overlap between test forms), (2) all items should be used (*item pool depletion*), (3) the expected test form response times should be as close to 10 minutes as possible, (4) the number of test forms should be determined accordingly, (5) item difficulty categories and items formats should be distributed as evenly as possible across test forms, (6) each content domain should be at least once in each test form, and (7) item exclusions should be incorporated.

The definition of the objective function and all constraints can be seen in Figure 6. Among the test specifications listed above, (3) is the specification most suitable for formulation as an objective function. This means that we want to optimize the test takers' mean test taking time and keep it as close as possible to 10 min per test form. In order to achieve this, we first transform the expected item response times to minutes. Then we calculate the ideal number of test forms by dividing the sum of all expected item response times (which is 74 min) by ten. As we prefer test forms below our target test time to test forms above our target test time, we choose the next integer above via the `ceiling()` function, resulting in eight test forms. The actual objective function is defined via the `minimaxObjective()` function, which allows us to specify a `targetValue`. The maximum difference of test form times from this target value is then minimized.

Second, we implement test specifications (1) and (2) (item pool should be depleted and no item overlap) as constraints using a single function call to `itemUsageConstraint()`. The operator argument is set to ''='' which means that every item will occur exactly once across all test forms. Test specification (5) refers to the difficulty column ''`diffCategory`'' as well as to the item format column ''`format`''. For item difficulty, we define the column ''`diffCategory`'' to be a factor variable, as we do not want the numerical mean value to be equal across test forms but the distribution of distinct difficulty levels. We use the function `autoItemValuesMinMaxConstraint()` to determine the required `targetValues` automatically, after which the function directly calls the respective constraint functions using the calculated `targetValues`. By default, the function returns the resulting minimum and maximum levels. For example, for item difficulty, items of difficulty category 1 will occur once or twice in each test form. Alternatively, for item formats, the ''`cmc`'' format will occur four or five times in each test form. Test specification (6) requires that each domain occurs at least once in each test form. Using the `itemCategoryMinConstraint()` and the `min` argument, we define for each of the three categories (levels) of `domain` (''`listening`'', ''`reading`'', ''`writing`'') the minimum occurrence frequency.

Finally, we implement test specification (7), the item exclusion constraints that are captured in the ''`exclusions`'' column of the `items_pilot data.frame`. The column contains item exclusions as a single character string for each item. The items in the data set have either no exclusions (`NA`), only one exclusion (e.g., ''`76`''), or multiple exclusions (e.g., ''`70, 64`''). As there are items with multiple exclusions, we need to separate the string into discrete item identifiers via the function `itemTuples()`, which produces pairs (*tuples*) of exclusive items (also called *enemy items*). Using the `sepPattern` argument in the `itemTuples()` function, the user must specify the pattern, which separates the item identifiers within the string. These tuples can be used to define exclusion constraints in the `itemExclusionConstraint()` function. The complete code for the pilot study use case, including the solver call and the solution inspection, can be seen in Supplement S1.

```
# Determine number of test forms
items_pilot$time_in_min <- items_pilot$time / 60
nForms    <- ceiling(sum(items_pilot$time_in_min) / 10 )

# Objective function (response times)
timeCons <- minimaxObjective(nForms = nForms,
            itemValues = items_pilot$time_in_min,
            targetValue = 10, itemIDs = items_pilot$item)

# Item pool depletion
noItemOverlap <- itemUsageConstraint(nForms, targetValue = 1,
                 operator = "=", itemIDs = items_pilot$item)

# Difficulty and format
items_pilot$diffCategory <- as.factor(items_pilot$diffCategory)
equal_diff  <- autoItemValuesMinMaxConstraint(nForms = nForms,
                              itemValues = items_pilot$diffCategory,
                              itemIDs = items_pilot$item)
equal_format <- autoItemValuesMinMaxConstraint(nForms = nForms,
                 itemValues = items_pilot$format,
                              itemIDs = items_pilot$item)

# Content categories
domainCons <- itemCategoryMinConstraint(nForms = nForms,
              itemCategories = items_pilot$domain,
              itemIDs = items_pilot$item, min = c(1, 1, 1))

# Exclusions
exclusionTuples <- itemTuples(items_pilot, idCol = "item",
                   infoCol = "exclusions", sepPattern = ",␣")
excl_constraints <- itemExclusionConstraint(nForms = nForms,
                    itemTuples = exclusionTuples,
                    itemIDs = items_pilot$item)
```

Figure 6. Define constraints for pilot study test assembly.

3.2. LSA Blocks for Multiple Matrix Booklet Designs

Many large-scale assessments (LSAs) test forms (typically referred to as booklets) consist of multiple item blocks (also referred to as clusters). In the test assembly process, first, item blocks are assembled from the item pool and later these item blocks are combined into test forms according to so called multiple matrix booklet designs [23]. Examples of this approach can be found in the PISA studies [4] or are described by Kuhn and Kiefer for the Austrian Educational Standards Assessment [3]. The present use case illustrates the first step—assembling test items to eight item blocks that fit in multiple matrix booklet designs. The second step—combining item blocks to test forms – is currently beyond the scope of `eatATA` and the reader is referred to the literature on booklet designs [24,25].

For this purpose we assume that a pilot study has been conducted and that all required parameter estimates from an item calibration are available. We use a simulated item pool of 209 items with typical properties, which is included in the `eatATA` package (`items_lsa`). The first 10 items of this item pool can be seen in Appendix C Table A2. The assembled item blocks should conform to the following test specifications: (1) blocks should contain as many well fitting items as possible, (2) hierarchical stimulus item structures should be incorporated, (3) no item overlap, (4) a fixed set of anchor items has to be included in the block assembly (if LSAs intend to measure trends between different times of measurement, new assessment cycles partially reuse items from former studies, so-called *anchor items*, to establish a common scale [26]. Usually, anchor items are chosen beforehand based on their advantageous psychometric properties), (5) the average item block times should be around 20 min, (6) difficulty levels should be distributed evenly across item blocks, (7) all blocks should contain at least three different item formats, and (8) maximally two items per block should have an average proportion of correct responses below 8 or above 92 percent.

The definition of the objective function and all constraints can be seen in Figure 7. Test specification (1) is chosen as the objective function. The infit (weighted MNSQ) is among the most widely used diagnostic Rasch fit statistics [27] and can be found in column "`infit`". As we are only interested in absolute deviations from 1 (otherwise positive and negative deviations could cancel each other out) we create a new variable, `infitDev`. The deviation of this variable from 0 is then minimized using the `minimaxObjective()` function.

```
# Objective function (infit)
infitDev <- abs(items_lsa$infit - 1)
infitCons <- minimaxObjective(nForms = 8, itemValues = infitDev,
                               targetValue = 0, itemIDs = items_lsa$item)

# Shared stimuli
incluTup <- stemInclusionTuples(items_lsa, "item", "testlet")
incluCons <- itemInclusionConstraint(nForms = 8, itemTuples = incluTup,
                                     itemIDs = items_lsa$item)

# Item overlap
overlapCons <- itemUsageConstraint(nForms = 8, targetValue = 1,
                                   operator = "<=",
                                   itemIDs = items_lsa$item)

# Anchor items
anchorCons <- itemUsageConstraint(nForms = 8, targetValue = 1,
                            operator = "=",
                            whichItems=items_lsa$item[items_lsa$anchor==1],
                            itemIDs = items_lsa$item)

# Block times
timeCons <- itemValuesDeviationConstraint(nForms = 8,
                                itemValues = items_lsa$time,
                                targetValue = 1170, allowedDeviation = 150,
                                relative = FALSE, itemIDs = items_lsa$item)
# Difficulty
diffLevels <- as.factor(items_lsa$level)
levelCons <- itemCategoryMinConstraint(nForms = 8, diffLevels,
                             itemIDs = items_lsa$item,
                             min = c(1,2,2,1))

# Item format
formatLv <- as.factor(ifelse(grepl("complex",items_lsa$format)|
                          grepl("matching",items_lsa$format),
                          "mix",ifelse(grepl("open",items_lsa$format)|
                                         grepl("sentence",items_lsa$format),
                             "open", "closed")))
formatCons <- itemCategoryMinConstraint(nForms = 8, formatLv,
                              itemIDs = items_lsa$item,  min = c(2,2,2))

# Proportion correct
freqLv <- as.factor(ifelse(items_lsa$frequency > .92, "above",
                        ifelse(items_lsa$frequency < .08, "below", "okay")))
freqCons <- itemCategoryMaxConstraint(nForms = 8, freqLv,
                            itemIDs = items_lsa$item,
                            max = c(2, 2, 200))
```

Figure 7. Define constraints for LSA test assembly.

Test specification (2) is a common challenge for cognitive tests in LSAs, where items are usually not distinct units. Instead, multiple items share a common stimulus (e.g., a text, a picture or an auditive stimulus). Such item sets are often called *testlets*. In general, testlet structures can be dealt with in different ways: (a) In the assembly, testlets can be treated as fixed structures and used as the actual units in the test assembly, (b) testlet structures can be incorporated using fixed inclusion constraints (e.g., whenever item A is chosen, items B and C that belong to the same stimulus have to be chosen, too), (c) hierarchical structures can be incorporated in the test assembly (see chapter 7 in [1]). In the `eatATA` package, options (a) and (b) are implemented and option (b) is chosen for this specific use case. Option (b) can indeed be implemented very similarly to the item exclusion constraints that were introduced in the pilot study use case. Inclusion tuples are built using the function `stemInclusionTuples()` and then provided to the `itemInclusionConstraint()` function.

Test specification (3) is implemented similarly as in the previous cases using the `itemUsageConstraint()` function. The "less than or equal" operator ‘‘<=’’ is used, because complete depletion of the item pool is not required. Test specification (4) refers to the forced inclusion of certain items in the block assembly, which can also be implemented using the `itemUsageConstraint()` function. In this specific case we specify the `whichItems` argument, which lets us choose to which items this constraint should apply. For this specification, the `operator` argument is set to ‘‘=’’ as the items have to appear once across the blocks.

The further test specifications are implemented in line with similar constraints in previous examples: block times, referring to test specification (5), are constrained using the `itemValuesDeviationConstraint()` function. Test specifications (6), (7), and (8) are implemented by transforming the respective variables to factors so we can apply the `itemCategoryMinConstraint()` or the `itemCategoryMaxConstraint()` functions. As every block should contain at least some items at the intermediate difficulty levels and also in each block at least one item at the adjacent difficulty levels, we set the `min` argument for this test specification to `c(1, 2, 2, 1)`. For test specification (7), item formats are grouped into three different groups, which then are constrained by setting the minimum number of items of each group per block to two. In some LSA studies, items are flagged that have empirical proportions correct below and/or above a certain value (cf., test specification (8)). Therefore, we limit the inclusion of items that range below 8 percent and above 92 proportion correct to a maximum of two items per category per block.

The complete code for the LSA use case, including the solver call and the solution inspection, can be seen in Supplement S2. Note that for this test assembly problem, the GLPK Simplex Optimizer finds a feasible solution very quickly but the complete integer optimization process takes a substantial amount of time, due to the large item pool, multiple assembled item blocks, and various constraints. This showcases that often setting a time limit and using a feasible but not optimal solution is sufficient in practice.

3.3. High-Stakes Assessment

This use case corresponds to Problem 1 in the paper by Diao and van der Linden [9]. Because the item pool used in Diao and van der Linden [9] is not freely available, an item pool was generated with similar characteristics (`items_diao` in the `eatATA` package). The item pool consists of 165 items following the three-parameter logistic model (3PL). Each item belongs to one of six content categories. The first five items of the generated item pool can be seen in Appendix D Table A3. In this example, the goal is to assemble two parallel test forms with the following test specifications: (1) absolute target values for the TIFs set as $T_\theta = 5.4, 10, 5.4$ at $\theta = -1.5, 0, 1.5$; minimize the distances of the TIFs of the two new forms with respect to the target at these ability values, (2) distribute the number of items per content category evenly across test forms; Appendix E Table A4 presents the numbers of items per content category that are available in the complete item pool as well as the numbers required in each of the two forms, (3) no overlapping items, and (4) each test form should contain exactly 55 items. These specifications are directly copied from Diao and van der Linden [9]. The code for calculating the IIF, and setting up the minimax objective function as well as the other constraints can be seen in Figure 8.

To implement test specification (1) we create minimax objects at each of the specified ability values, and combine them in one objective function. Hence, when solving the ATA problem, the solver will try to minimize the maximal distance between the target and the two forms at the three ability values. The implementation of the further constraints directly corresponds to formulations of test specifications in the use cases above. Therefore, further explanations on these specifications are omitted. The complete code for the high-stakes assessment use case, including the solver call and the solution inspection, can be seen in Supplement S3.

3.4. Multi-Stage Testing

This use case covers the case of multi-stage testing and corresponds to Problem 3 in Diao and van der Linden [9]. The use case uses the same items as use case (3), but the item pool is doubled: all items are duplicated. Hence, the item pool in this example contains 330 items following the 3PL model. Here, the goal is to assemble a two-stage multi-stage test with one routing module in the first stage and three modules in the second stage. The test specifications are: (1) the TIF of the first-stage module is required to be relatively uniform between $\theta = -1$ and $\theta = 1$, (2) the TIFs of the second-stage modules are required to be single-peaked at $\theta = -1$, $\theta = 0$ and $\theta = 1$, respectively, (3) the number of item per content category should be evenly distributed across the test forms according to Appendix E Table A4, (4) no item overlap, (5) for the first-stage module the test length should be 30 items, and (6) for each of the second-stage modules the test length should be 20 items.

```
# Theta values
theta_values <- c(-1.5, 0, 1.5)

# Calculate item information for each theta-value
items_diao[, paste0("IIF_", theta_values)] <- calculateIIF(A = items_diao$a,
                                                           B = items_diao$b,
                                                           C = items_diao$c,
                                                           theta = theta_values)

# Specify target values for each theta-value
target_values <- structure(c(5.4, 10, 5.4),
                           names = paste0(theta_values))

# Objective function: minimize maximum difference between the TIF and
#  the target values
minimaxTif <- combineConstraints(lapply(theta_values,
                                        function(theta_value) {
  minimaxObjective(
    nForms = 2,
    itemValues = items_diao[, paste0("IIF_", theta_value)],
    targetValue = target_values[as.character(theta_value)],
    itemIDs = items_diao$item
  )
}))

# Other constraints
contentConstraints <- itemCategoryConstraint(
  nForms = 2,
  itemCategories = items_diao$Category,
  operator = ">=",
  targetValues = c(9, 9, 7, 9, 9, 11),
  itemIDs = items_diao$item)

noOverlap <- itemUsageConstraint(
  nForms = 2,
  itemIDs = items_diao$item)

testLength <- itemsPerFormConstraint(
  nForms = 2,
  operator = "=",
  targetValue = 55,
  itemIDs = items_diao$item)
```

Figure 8. Constraint definitions for high-stakes assessment test assembly.

3.4.1. Original Approach

Diao and van der Linden [9] split this assembly problem in two separate problems. In a first step, the routing module for the first stage is assembled. Thereafter, the three modules for the second stage are assembled using only the remaining items in the pool. In order to assemble the routing module with a uniform relative target, a maximin approach is used—that is, the minimum value of the TIF at the three ability values is maximized. At the same time, the TIF values at the three ability values are required to be close to each other, in order to create a TIF with a relatively flat plateau. More specifically, the TIF at the three ability values is required to be within a distance of 0.5 of each other. Hence, the `allowedDeviation` is set to 0.5. The syntax for the implementation of the objective function and the constraints can be found in Appendix F Figures A1 and A2.

3.4.2. Combined Capped Approach

Using the `eatATA` package, it is possible to assemble the modules for the two stages in one combined assembly. Especially in situations with multiple stages, a simultaneous assembly may prevent infeasibility at later stages—that is, when the modules for the stages are assembled sequentially, the assembly of the first stages may deplete the item pool so that it becomes impossible to meet certain test specifications at later stages. In addition, from a practitioners perspective, a simultaneous assembly may also be easier, as the item pool does not need to be adjusted after every assembly step.

The `R` syntax for the combined capped approach can be seen in Figure 9. To implement test specification (1) we specify the maximization of the minimum TIF values at the ability values for the routing module in the first stage. Note that we do not use the original maximin approach but rather the capped maximin approach [20]. The capped maximin approach does not require to set a maximally allowed deviation, it combines maximizing the minimal TIF with minimizing the maximal difference between the TIFs. For the modules at the second stage (test specification (2)) the capped maximin approach is also used. To combine these constraints, knowing that the obtained TIF values in the first stage and the obtained TIF values at the second stage do not need to be in the same range, we can set a weight for the TIF values. In this case, the minimal TIF in both stages can be considered equally important. Hence, the weights are set to 1 (which is the default). Because the other test specifications in Figure 9 correspond to test specifications illustrated earlier, further explanations are omitted. The complete code for the multi-stage assessment use case, both for the original two-stage as well as the new combined assembly, with the solver call and the solution inspection, can be found in Supplement S4.

```
# Objective function (TIF stage 1)
maximinTIF1 <- combineConstraints(lapply(theta_values,
                                         function(theta_value)
{
  cappedMaximinObjective(
    nForms = 4,
    itemValues = items_diao2[, paste0("IIF_", theta_value)],
    weight = 1,
    whichForms = 1,
    itemIDs = items_diao2$item)
}))

# Objective function (TIF stage 2)
maximinTIF2 <- combineConstraints(lapply(theta_values,
                                         function(theta_value)
{
  cappedMaximinObjective(
    nForms = 4,
    itemValues = items_diao2[, paste0("IIF_", theta_value)],
    weight = 1,
    whichForms = which(theta_values == theta_value) + 1,
    itemIDs = items_diao2$item)
}))

# Content categories stage 1 and 2
contentConstraints1 <- itemCategoryConstraint(
  nForms = 4,
  itemCategories = items_diao2$Category,
  operator = ">=",
  targetValues = c(4, 4, 3, 4, 4, 5),
  whichForms = 1,
  itemIDs = items_diao2$item)
contentConstraints2 <- itemCategoryConstraint(
  nForms = 4,
  itemCategories = items_diao2$Category,
  operator = ">=",
  targetValues = c(3, 3, 2, 3, 3, 4),
  whichForms = 2:4,
  itemIDs = items_diao2$item)

# No item overlap
noOverlap2 <- itemUsageConstraint(
  nForms = 4,
  itemIDs = items_diao2$item)

# Test length stage 1 and 2
testLength1 <- itemsPerFormConstraint(
  nForms = 4,
  operator = "=",
  targetValue = 30,
  whichForms = 1,
  itemIDs = items_diao2$item)
testLength2 <- itemsPerFormConstraint(
  nForms = 4,
  operator = "=",
  targetValue = 20,
  whichForms = 2:4,
  itemIDs = items_diao2$item)
```

Figure 9. Constraint definitions for multi-stage assessment module assembly.

4. Discussion

In 2005, van der Linden published his seminal book on automated test assembly. Since then, additional tutorial papers and illustrations have been written to make ATA methods more accessible to practitioners e.g., [9,19]. However, we believe that hurdles are still quite substantial for practitioners who want to utilize ATA methods: Besides the conceptual challenges of formulating test specifications as concrete constraints, one has to become familiar with formulating mathematical constraints and the intricacies of specific solver APIs. The `eatATA` package and this tutorial paper have been written to promote ATA methods and make them more accessible to practitioners and researchers. We have provided a short overview of the basic ideas of ATA and an illustration of the typical `eatATA` work flow. Using a small illustrative example and four different, more realistic use cases, we demonstrated how the package can be used to implement ATA in `R`. By choosing a wide range of different ATA applications with diverse test specifications we hope to spark interest in ATA methods in a broad audience.

4.1. Limitations

There are currently a few limitations when using `eatATA` to solve ATA problems. As mentioned in use case (2), hierarchical item stimulus structures can not be implemented as flexibly as suggested by van der Linden [1] with optional item selection. However in practice, item sets are often treated as fixed units anyway, as altering the item set that is presented alongside a stimulus might have undesirable effects on the psychometric properties of the individual items. Furthermore, item overlap specifications between test forms cannot be specified directly in `eatATA`. Generally, a direct implementation of item overlap constraints drastically increases the complexity of the mathematical programming problem, resulting in high computing times. Moreover, often item overlap specifications can be met indirectly, for instance by first selecting a set of items that can serve as overlap items, and then constraining the number of overlap items per test form, as well as how many times the overlap items can appear across the test forms. Therefore, direct overlap constraints are deliberately not included in `eatATA`. Finally, solver selection in `eatATA` is limited to the solvers mentioned in the introduction. For example, `CPLEX` and `XPRESS` are potent commercial alternatives to `Gurobi`. Another potentially promising open source solver, unfortunately currently without an `R` API is `SCIP` [20]. However, we do believe that for many ATA contexts the available selection of solvers is more than sufficient.

4.2. Alternatives

It is noteworthy that while for most data handling procedures or statistical methods a wide variety of `R` packages exists, this is not the case for ATA methods. More precisely, we are only aware of four other `R` packages on CRAN that have some ATA functionality implemented, of which only two (`TestDesign`, `RSCAT`) seem to be under active development. Indeed, `TestDesign` [28] provides access to the same selection of solvers as `eatATA` but has a strong focus on adaptive testing. This is illustrated by the fact that `TestDesign` is not suited for the assembly of multiple parallel test forms. In a similar vein, `RSCAT` provides functionality specific to the shadow test approach in computerized adaptive testing [29]. However, other testing approaches, such as multi-stage testing or linear testing, are not supported in that package. Finally, `Rata` [30] and `xxIRT` [31] also implement ATA methods in `R`. Yet both packages only provide access to the `Rglpk` and `lpSolve` solvers. In addition, although both packages in general have a similar work flow compared to `eatATA`, their functionality is more limited compared to `eatATA` (e.g., no specific categorical item constraint functions, no automatic calculation of target values, no item inclusions constraints).

Furthermore, alternative approaches to MIP have been proposed in the past, e.g., heuristic algorithms, which are also capable of solving automated test assembly problems. Examples include genetic algorithms [32–34] or simulated annealing [35]. For a short but comprehensive overview see van der Linden [1]. As these algorithms do not search the entire solution space they are not guaranteed to find the optimal solution to the optimization problem but may be computationally faster than the classic MIP solvers that are used by `eatATA`. Another potential benefit of heuristic algorithms is that some allow the introduction of soft constraints, which might be helpful for dealing with feasibility issues. However, to our knowledge, only limited ATA applications of these algorithms exist. For example, we are not aware of a single ATA application of heuristic algorithms using `R`. Furthermore, it can be argued that for most practical ATA applications MIP solvers perform sufficiently well from a computational stand point [36].

4.3. Conclusions

We believe that `eatATA` can be a helpful tool for researchers and practitioners that want to assemble test forms. It is applicable in a wide range of scenarios and its user interface should be rather intuitive for `R` users. By providing this tool we hope to promote automated test assembly methods, which are almost always superior to manual test assembly approaches.

Supplementary Materials: The following are available online at https://www.mdpi.com/article/10.3390/psych3020010/s1. R-Syntaxes S0-S4.

Author Contributions: Conceptualization, B.B. and D.D.; software, B.B. and D.D.; formal analysis, B.B., D.D., K.A.S. and S.W.; writing—original draft preparation, B.B., D.D., K.A.S. and S.W.; writing—review and editing, B.B., D.D., K.A.S. and S.W. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: All simulated data sets are available within the `eatATA` package.

Acknowledgments: We thank Pauline Kohrt for encouraging us to program `eatATA` and early testing of the package. We also thank Simone Dubiel for further ideas for and testing of `eatATA`.

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ATA	Automated Test Assembly
MIP	Mixed-integer Programming
CAT	Computerized Adaptive Testing
MST	Multi-stage Testing
LSA	Large-scale Assessment
HST	High-stakes Assessment
IIF	Item Information Function
TIF	Test Information Function

Appendix A

In the minimal example, the combination of constraints and objective results in an MIP model that can be mathematically formulated as follows. Let the items in the item pool have a unique index $i = 1, \ldots, I$, in this example $I = 30$. Let F be the total number of test forms to be assembled, here $F = 1$. The MIP model has two parts: (1) the objective function,

$$max\ \mathbf{c}^T\mathbf{x}; \tag{A1}$$

and (2) a set of constraints,

$$\mathbf{Ax} \leq \mathbf{d}, \tag{A2}$$

where $\mathbf{x}$ is the vector of variables that MIP needs to solve for. $\mathbf{x}$ contains binary decision variables x_{if}, $i = 1, 2, \ldots, I$, and $f = 1, 2, \ldots, F$ for every item $\times$ test from combination, and one real-valued variable z. Hence, $\mathbf{x}$ is a vector of length $I \times F + 1$. The binary decision variables are defined as:

$$x_{if} = \begin{cases} 1 & \text{if item } i \text{ is assigned to form } f, \\ 0 & \text{otherwise.} \end{cases} \tag{A3}$$

Further, in the objective function (Equation (A1)), **c** is a numeric vector of $I \times F + 1$ known coefficients for the objective function. In the set of constraints (Equation (A2)) **A** is a known coefficient matrix with $I \times F + 1$ columns and with one row for each constraint and **d** is a vector with the corresponding right-hand values of the constraints.

The code in Figure 2 simultaneously creates coefficients for **c** as well as coefficients for one row in **A** and the corresponding value in **d**. More specifically, the coefficients in **c** for all binary decision variables are set to zero, whereas the coefficient for the real-valued variable z is set to one 1. Thus, Equation (A1) simplifies to:

$$max\, z, \tag{A4}$$

In addition, the following constraint is added as one row in **A** and the corresponding value in **d**:

$$\sum_{i=1}^{I} s_i \times x_{if} - z \geq 0, \quad \text{for } f = 1, \ldots, F. \tag{A5}$$

In Equation (A5) s_i denotes the IIF value of item i at a medium ability level. Hence, the coefficients in **A** for the binary decision variables x_{if} are set to s_i, whereas the coefficient for z is set to 1. Finally, the value in **d** corresponding to the row in **A** is set to 0. The combination of Equations (A4) and (A5) makes sure that the TIF of the test forms is maximized.

In addition, the following constraints are also enforced:

$$\sum_{i=1}^{I} x_{if} = 10, \quad \text{for } f = 1, \ldots, F, \tag{A6}$$

and

$$\begin{aligned} &\sum_{i=1}^{I} t_i \times x_{if} \leq (8 \times 60) + 5, \quad \text{and} \\ &\sum_{i=1}^{I} t_i \times x_{if} \geq (8 \times 60) - 5, \quad \text{for } f = 1, \ldots, F. \end{aligned} \tag{A7}$$

In Equation (A7) t_i denotes the average response time for item i in seconds. Hence, Equation (A6) constrains the length of the test forms to be equal to ten and Equation (A7) constrains the sum of the expected response times to be within five seconds of eight minutes. The left-hand sides and the right-hand sides of Equations (A5) to (A7) again correspond to the rows in **A** and **d**, respectively. Note that the coefficient for z in these rows of **A** are set to zero.

Appendix B

Table A1. First five items of the simulated pilot study item pool.

Item	diffCategory	Format	Domain	Time	Exclusions
1	2	cmc	listening	44.54	
2	4	cmc	listening	44.81	
3	4	mc	writing	32.36	76
4	2	mc	listening	48.03	
5	2	mc	writing	42.06	9

Appendix C

Table A2. First 10 items of the simulated LSA assessment item pool.

Testlet	Item	Level	Format	Frequency	Infit	Time	Anchor
TRA5308	TRA5308a	IV	multiple choice	0.19	1.22	54.00	0
TRA5308	TRA5308b	IV	multiple choice	0.24	1.01	66.00	0
TRA5308	TRA5308c	II	multiple choice	0.42	1.22	89.00	0
TRA5308	TRA5308d	III	multiple choice	0.41	1.21	92.00	0
TRB6832	TRB6832a	III	open answer	0.51	1.21	85.00	0
TRB6832	TRB6832b	III	open answer	0.20	1.08	61.00	0
TRB6832	TRB6832c	IV	open answer	0.33	1.25	84.00	0
TRB6832	TRB6832d	II	open answer	0.49	1.05	109.00	0
TRC9792	TRC9792a	I	cmc	0.70	1.10	94.00	0
TRC9792	TRC9792b	I	cmc	0.61	1.02	110.00	0

Appendix D

Table A3. First five items of the simulated high-stakes assessment item pool.

Item	a	b	c	Category
1	0.54	−0.09	0.17	6
2	0.71	−1.07	0.24	1
3	0.84	−1.11	0.17	2
4	1.38	−0.71	0.21	3
5	1.26	−0.44	0.12	4

Appendix E

Table A4. Item category distribution in the item pool and test specification.

	Cat. 1	Cat. 2	Cat. 3	Cat. 4	Cat. 5	Cat. 6
Item Pool	23	26	22	29	29	36
HST	9	9	7	9	9	11
MST: Stage 1	4	4	3	4	4	5
MST: Stage 2	3	3	2	3	3	4

Appendix F

```
maximinTIF1 <- combineConstraints(lapply(1:3, function(index)
  {
    maximinObjective(
      nForms = 1,
      itemValues = IIFs[,index],
      allowedDeviation = 0.5,
      itemIDs = items_diao2$item
    )
  })
)

contentConstraints1 <- itemCategoryConstraint(
  nForms = 1,
  itemCategories = items_diao2$category,
  operator = ">=",
  targetValues = c(4, 4, 3, 4, 4, 5),
  itemIDs = items_diao2$item)

noOverlap1 <- itemUsageConstraint(
  nForms = 1,
  itemIDs = items_diao2$item)

testLength1 <- itemsPerFormConstraint(
  nForms = 1,
  operator = "=",
  targetValue = 30,
  itemIDs = items_diao2$item)
```

Figure A1. Test assembly constraints for multi-stage test stage 1.

```
maximinTIF2 <- combineConstraints(lapply(1:3, function(index)
  {
    maximinObjective(
      nForms = 3,
      itemValues = IIFs_stage2[, index],
      allowedDeviation = 0.2,
      whichForms = index,
      itemIDs = items_diao2_stage2$item)
  })
)

contentConstraints2 <- itemCategoryConstraint(
  nForms = 3,
  itemCategories = items_diao2_stage2$Category,
  operator = ">=",
  targetValues = c(3, 3, 2, 3, 3, 4),
  itemIDs = items_diao2_stage2$item)

noOverlap2 <- itemUsageConstraint(
  nForms = 3,
  itemIDs = items_diao2_stage2$item)

testLength2 <- itemsPerFormConstraint(
  nForms = 3,
  operator = "=",
  targetValue = 20,
  itemIDs = items_diao2_stage2$item)
```

Figure A2. Test assembly constraints for multi-stage test stage 2.

References

1. Van der Linden, W.J. *Linear Models for Optimal Test Assembly*; Springer: New York, NY, USA, 2005.
2. Luecht, R.M.; Sireci, S.G. *A Review of Models for Computer-Based Testing*; *Research Report 2011-12*; College Board: New York, NY, USA, 2011.
3. Kuhn, J.T.; Kiefer, T. Optimal test assembly in practice. *Z. Für Psychol.* **2015**, *221*, 190–200. [CrossRef]
4. OECD. *PISA 2018 Technical Report*; Technical Report; OECD Publishing: Paris, France, 2019.
5. Yan, D.; Von Davier, A.A.; Lewis, C. *Computerized Multistage Testing: Theory and Applications*; CRC Press: Boca Raton, FL, USA, 2016.
6. Van der Linden, W.J.; Glas, C.A. *Computerized Adaptive Testing: Theory and Practice*; Kluwer Academic Publishers: New York, NY, USA, 2000.
7. R Core Team. *R: A Language and Environment for Statistical Computing*; R Foundation for Statistical Computing: Vienna, Austria, 2020.
8. Konis, K.; Schwendinger, F. *lpSolveAPI: R Interface to 'lp_solve' Version 5.5.2.0*; R Package Version 5.5.2.0-17.7. 2020. Available online: https://CRAN.R-project.org/package=lpSolveAPI (accessed on 20 May 2021).
9. Diao, Q.; van der Linden, W.J. Automated test assembly using lp_solve version 5.5 in R. *Appl. Psychol. Meas.* **2011**, *35*, 398–409. [CrossRef]
10. Becker, B.; Debeer, D. *eatATA: Create Constraints for Small Test Assembly Problems*; R Package Version 0.11.2. 2021. Available online: https://CRAN.R-project.org/package=eatATA (accessed on 20 May 2021).
11. Makhorin, A. GLPK (GNU Linear Programming Kit). 2018. Available online: https://www.gnu.org/software/glpk/ (accessed on 20 May 2021).
12. Berkelaar, M.; Eikland, K.; Notebaert, P. lp_solve 5.5.2.5. 2016. Available online: http://lpsolve.sourceforge.net/5.5/ (accessed on 20 May 2021).
13. Ladanyi, L.; Ralphs, T.; Menal, G.; Mahajan, A. coin-or/SYMPHONY: Version 5.6.17. 2019. Available online: https://projects.coin-or.org/SYMPHONY (accessed on 20 May 2020).
14. Gurobi Optimization, LLC. *Gurobi Optimizer Reference Manual*; Gurobi Optimization, LLC: Houston, TX, USA, 2021.
15. Theussl, S.; Hornik, K. *Rglpk: R/GNU Linear Programming Kit Interface*; R Package Version 0.6-4. 2019. Available online: https://CRAN.R-project.org/package=Rglpk (accessed on 20 May 2020).
16. Berkelaar, M.; Csárdi, G. *lpSolve: Interface to 'Lp_solve' v. 5.5 to Solve Linear/Integer Programs*; R Package Version 5.6.15. 2020. Available online: https://CRAN.R-project.org/package=lpSolve (accessed on 20 May 2021).
17. Harter, R.; Hornik, K.; Theussl, S. *Rsymphony: SYMPHONY in R*; R Package Version 0.1-29. 2020. Available online: https://CRAN.R-project.org/package=Rsymphony (accessed on 20 May 2021).
18. Gurobi Optimization, LLC. *Gurobi: Gurobi Optimizer 9.1 interface*, R package version 9.1-1; Gurobi Optimization, LLC: Houston, TX, USA, 2021.
19. Donoghue, J.R. *Comparison of Integer Programming (IP) Solvers for Automated Test Assembly (ATA)*; Research Report 15-05; Educational Testing Service: Princeton, NJ, USA, 2015.
20. Luo, X. Automated Test Assembly with Mixed-Integer Programming: The Effects of Modeling Approaches and Solvers. *J. Educ. Meas.* **2020**, *57*, 547–565. [CrossRef]
21. Rasch, G. *Studies in Mathematical Psychology: I. Probabilistic Models for Some Intelligence and Attainment Tests*; Nielsen & Lydiche: Copenhagen, Denmark, 1960.

22. Spaccapanico Proietti, G.; Matteucci, M.; Mignani, S. Automated Test Assembly for Large-Scale Standardized Assessments: Practical Issues and Possible Solutions. *Psych* **2020**, *2*, 315–337. [CrossRef]
23. Gonzalez, E.; Rutkowski, L. Principles of multiple matrix booklet design and parameter recovery in large-scale assessments. In *IERI Monograph Series: Issues and Methodologies in Large-Scalse Assessments: Volume 3*; von Davier, M., Hastedt, D., Eds.; IEA-ETS Research Institute: Hamburg, Germany, 2010; pp. 125–156.
24. Frey, A.; Hartig, J.; Rupp, A.A. An NCME instructional module on booklet designs in large-scale assessments of student achievement: Theory and practice. *Educ. Meas. Issues Pract.* **2009**, *28*, 39–53. [CrossRef]
25. Pokropek, A. Missing by design: Planned missing-data designs in social science. *Res. Methods* **2011**, *20*, 81–105.
26. Kolen, M.J.; Brennan, R.L. *Test Equating, Scaling, and Linking: Methods and Practices*; Springer Science & Business Media: New York, NY, USA, 2014.
27. OECD. *PISA 2012 Technical Report*; Technical Report; OECD Publishing: Paris, France, 2014.
28. Choi, S.W.; Lim, S. *TestDesign: Optimal Test Design Approach to Fixed and Adaptive Test Construction*; R Package Version 1.2.2. 2021. Available online: https://CRAN.R-project.org/package=TestDesign (accessed on 20 May 2021).
29. Jiang, B. *RSCAT: Shadow-Test Approach to Computerized Adaptive Testing*; R Package Version 1.1.0. 2021. Available online: https://CRAN.R-project.org/package=RSCAT (accessed on 20 May 2021).
30. Luo, X. *Rata: Automated Test Assembly*; R Package Version 0.0.2. 2019. Available online: https://CRAN.R-project.org/package=Rata (accessed on 20 May 2021).
31. Luo, X. *xxIRT: Item Response Theory and Computer-Based Testing in R*; R Package Version 2.1.2. 2019. Available online: https://CRAN.R-project.org/package=xxIRT (accessed on 20 May 2021).
32. Chang, T.Y.; Shiu, Y.F. Simultaneously construct IRT-based parallel tests based on an adapted CLONALG algorithm. *Appl. Intell.* **2012**, *36*, 979–994. [CrossRef]
33. Sun, K.T.; Chen, Y.J.; Tsai, S.Y.; Cheng, C.F. Creating IRT-based parallel test forms using the genetic algorithm method. *Appl. Meas. Educ.* **2008**, *21*, 141–161. [CrossRef]
34. Verschoor, A.J. Genetic Algorithms for Automated Test Assembly. Ph.D. Thesis, Twente University, Enschede, The Netherlands, 2007.
35. Veldkamp, B.P. Multiple objective test assembly problems. *J. Educ. Meas.* **1999**, *36*, 253–266. [CrossRef]
36. Van der Linden, W.J.; Li, J. Comment on three-element item selection procedures for multiple forms assembly: An item matching approach. *Appl. Psychol. Meas.* **2016**, *40*, 641–649. [CrossRef] [PubMed]