Abduction and consequence generation in a support system for the design of logical multiple-choice questions

Marta Cialdea Mayer

Dipartimento di Informatica e Automazione
Università di Roma Tre
Italy

Abstract

This paper presents Logitest, a system that can be used to assist the designer of multiple-choice questions aiming at verifying logical reasoning abilities (provided they can be represented in propositional logic). Beyond allowing the designer to check the status of an option with respect to the item stem (the problem presentation), i.e. whether it is derivable or consistent with the given information, the system can accomplish generative tasks (abduction and consequence generation) that can be useful both to complete the item stem, and to generate plausible distractors (incorrect options). The provably correct and complete algorithm used to perform abduction and consequence generation finds out minimal solutions with no need to compare each of them with all the others.

1 Introduction

Many educational institutions require students to take admission tests, either to check whether they have a suitable initial preparation and attitude, or to select (a limited number of) the most promising students among the applicants. Admission tests often include multiple-choice questions aiming at assessing logical reasoning abilities. The same kind of questions can be presented to students in scientific disciplines, who are generally required to be proficient in logical operations. Many logical tests have the following form: "given the description of a state of facts, which of the assertions listed below is derivable/underivable/consistent/inconsistent with the information you have?". Often, the situation described in such questions concerns a fixed number of individuals, so that it can be represented in propositional logic.

A multiple-choice item is made up by (1) an *item stem* that presents the problem, (2) a sequence of *options*, containing (2a) the *correct option*, and (2b) several *distractor* options, i.e. incorrect alternative answers. A well designed multiple-choice item should meet some important basic requirements (see, for

instance, [2, 4]). Some of them concern the linguistic presentation of the question, others are of a more "logical" nature. In particular, it is required that (1) exactly only one of the options is correct, and (2) the options should be equally plausible. The first of such requirements seems quite obvious, and yet, in the case of logical test items, checking its fulfillment can often be tricky. As a matter of fact, invalid test items are not so infrequent. The second requirement is even subtler, especially for logical tests, and very often unmet.

The author of this paper experienced the difficulty in designing good logical questions when she was charged by her Faculty to edit the logical section of students' admission tests. In fact, some of the questions proposed by colleagues appeared to be incorrect, as well as others the author found in published test collections. Moreover, the designer can easily "run out of ideas". This motivated the implementation of Logitest, a tool to assist a test designer in both checking and devising multiple-choice logical test items.

2 The main functionalities of the system

Logitest is a simple support system for the design and check of test items of the form "given the information T, which of the following assertions is derivable/underivable/consistent/inconsistent with T?". The system is written in Objective Caml [6] and is available at Logitest web page [7].

The system takes a file in input, specifying the components of the item and the tasks to be executed. The input file contains a description of the language used in the test (predicates, object types, constants, etc.) and the item stem (the information T), represented by a set of logical formulae. The syntax used in the input file is first-order, but formulae are translated into a propositional language. In fact, the program can only deal with questions involving a finite and fixed set of objects (which are assumed to be pairwise different).

Moreover, the input file contains directives to perform the tasks described in the sequel.

Provability and consistency check. The input file can specify a set of formulae to be checked for provability from the stem T or consistency with T. Then the system checks them one by one. This is the basic task, useful to verify whether one and only one of the options represents the correct answer.

Completion of the information given in the item stem. This task is useful when the designer has run out of ideas; in this case the item stem specified in the input file contains general information and possibly some facts, that however are not sufficient to derive some given conclusion C, representing the correct option. The system generates all the minimal conjunctions of literals E_i, that are consistent with T and such that $T \cup \{E_i\} \models C$ (excluding trivial ones, i.e. such that $E_i \models C$). In other terms, the system solves the abduction problem given by T and C. A set of *abducible predicates* can also be specified, in order to obtain solutions containing only such predicates.

Generation of correct answers. The system can be asked to generate formulae which are derivable from the item stem; the logical consequences that are generated are minimal disjunctions of literals, excluding trivial ones (that are already explicitly present in the item stem). The system can similarly be asked to generate literals that are consistent/inconsistent with the stem (excluding its

logical consequences).

Generation of distractors. The specification file may include a belief set, that is intended to represent possible students' *misconceptions* or an incorrect interpretation of the stem (such as, for instance, reading a logical equivalence in place of an implication). The beliefs are used to generate incorrect though "plausible" options. The system generates minimal disjunctions C of literals such that $B \models C$ and $T \not\models C$, where B is the belief set and T the item stem.

3 Techniques and algorithms

Logical consequence and satisfiability tests are carried out by means of Ordered Binary Decision Diagrams [1]. The generation of consistent and inconsistent literals simply iterates over the set of literals and tests them one by one.

Abduction is a computationally hard problem: deciding whether a given abduction problem has a solution at all is a Σ_2^P-complete problem [5]. However, due to the rather small size of problems to be solved in the present context, it is a feasible task.

Logic-based abduction essentially consists in the generation of (non-trivial) explanations E such that $T \cup \{E\} \models F$ (where F is the "observation"), i.e. such that $T \cup \{\neg F\} \models \neg E$. Abduction and consequence generation can therefore be approached in the same way. And in fact Logitest performs the two tasks following the same methodology. From the syntactical point of view, abduction usually requires E to be a conjunction of literals. Using the same approach in consequence generation means that "interesting" consequences of the stem or the belief set are disjunctions of literals. The main difference between consequence generation and logic-based abduction is that the latter requires E to be consistent with T and such that $E \not\models F$, while interesting logical consequences of T should exclude those that are already explicitly contained in T itself. Such tests can however be performed after having generated E. Similarly, when generating distractors, i.e. consequences of the beliefs that are not implied by T, the test $T \not\models C$ is performed after the generation of C.

In the following, the symbol T^* stands for $T \cup \{\neg F\}$ in the case of an abductive task, T itself (or the belief set) in the consequence generation task. Candidates for abductive explanations are usually subjected to minimality criteria, such as subset-minimality. It is reasonable to ask the consequence generation task to fulfill the same requirement. In general, therefore, we want to find out *subset-minimal* sets S of literals such that $T^* \cup S$ is inconsistent. The algorithm used for performing both tasks takes T^* in input and returns a set S of literals. In the abduction case, the solution is the conjunction of such literals; in the consequence generation case, it is the disjunction of the complements of the literals in S.

The algorithm is inspired by [3], where a proof-theoretical abduction method based on semantic tableaux is defined. Explanations are identified on the basis of the set of open branches in a complete tableau for T^* (possibly simplified using containment), and such a characterization is the declarative counterpart of a non-deterministic algorithm that generates a single (minimal) abductive explanation, with no need to test each candidate solution against the others. However, in order to be complete, the algorithm suggested in [3] has to consider every permutation of the set of open tableaux branches. Logitest uses an

improved algorithm, that iterates over a single sequence of tableau branches.

Like in [3], first of all a (simplified) set $\{F_1, ..., F_n\}$ of sets of literals representing the open branches in a complete tableaux for T^* is built (if we identify T^* with the conjunction of its elements, $\{F_1, ..., F_n\}$ represents a DNF of T^*). Then the branches are *complemented*, building the set $\gamma(T^*) = \{C_1, ..., C_n\}$ where $C_i = \{\overline{\ell} \mid \ell \in F_i\}$ ($\overline{\ell}$ being the complement of ℓ).

In order to present the algorithm, some preliminary definitions are needed. A set $S = \{\ell_1, ..., \ell_k\}$ of literals is said to *cover* a set $\Gamma = \{C_1, ..., C_n\}$, where each C_i is a set of literals, if for all $C_i \in \Gamma$, $S \cap C_i \neq \emptyset$ (i.e. S contains at least one element from each C_i).

Obviously, $T^* \cup S$ is inconsistent iff S covers $\gamma(T^*)$. Therefore, the search space of candidate solutions is constituted by the sets covering $\gamma(T^*)$. Candidate solutions S are generated by a backtracking algorithm which builds S in an incremental manner, scanning the list of elements of $\gamma(T^*)$.

Minimality is dealt with by means of the following notion. A set of literals S is said to be *admissible* with respect to a set Γ of sets of literals if and only if for all $\ell \in S$ there exists $C \in \Gamma$ such that $S \cap C = \{\ell\}$. I.e. S is not admissible wrt Γ iff there exists a literal $\ell \in S$ such that for all $C \in \Gamma$ containing ℓ, C contains also some other literal $\ell' \in S$. This means that the literal ℓ is *redundant*: if S covers Γ, ℓ can be eliminated from S obtaining a set that still covers Γ.

Finally, an *admissible covering* of Γ is a set that covers Γ and is admissible wrt Γ, and a *non-admissible covering* of Γ is a set that covers Γ and is not admissible wrt Γ.

It can easily be proved that:

P1. If S covers Γ, then it is subset-minimal (i.e. no $S' \subset S$ covers Γ) if and only if it is admissible wrt Γ.

Therefore, S is a minimal consistent set of literals such that $T^* \cup S$ is inconsistent iff S is a consistent and admissible covering of $\gamma(T^*)$.

The algorithm used by Logitest builds a consistent solution S incrementally, in such a way that it is always admissible. Initially, $S = \emptyset$. Then the elements of $\gamma(T^*)$ are considered one by one (in any fixed order). Each $C_i \in \gamma(T^*)$ such that $S \cap C_i \neq \emptyset$, is ignored. If $S \cap C_i = \emptyset$, a literal ℓ from C_i consistent with S is chosen; if either no literal in C_i is consistent with S of $S \cup \{\ell\}$ is not admissible wrt $\{C_1, ..., C_i\}$, the algorithm fails (and possibly backtracks), otherwise goes on to C_{i+1}, with $S \cup \{\ell\}$. Backtracking on every choice point generates all the minimal solutions of the problem.

Here follows the pseudo-code of the backtracking algorithm. In the description of the algorithm, **choose** identifies a backtrackable choice point.[1]

Input: T^*, a set of formulae
Output: a set of literals
1) Build a simplified DNF $\{F_1, ..., F_n\}$ of T^*
2) Build the set $\gamma(T^*) = \{C_1, ..., C_n\}$, where $C_i = \{\overline{\ell} \mid \ell \in F_i\}$
3) Initialize $S_0 \leftarrow \emptyset$
4) **for** $i = 1, ..., n$
5) **do if** $S_{i-1} \cap C_i = \emptyset$ **then**
6) **choose** $\ell \in C_i$ such that $\overline{\ell} \notin S_{i-1}$

[1]The algorithm suggested by [3] builds candidate solutions in a similar way, but for the fact that the admissibility test is replaced by a restriction on literal choices: a literal ℓ cannot be added to S_i at stage i if $\ell \in C_1 \cup ... \cup C_{i-1}$. If such an algorithm is run only on a fixed permutation of $\{C_1, ..., C_n\}$, it is incomplete.

7) **if** there is no such ℓ **then fail**
8) **else** $S_i \leftarrow \{\ell\} \cup S_{i-1}$
9) **if** S_i is not admissible wrt $\{C_1, ..., C_i\}$, **then fail**
10) **done**
11) **return** S_n

The algorithm is obviously correct: S_n does not contain any pair of complementary literals (test at line 7), it covers $\gamma(T^*)$ and is admissible wrt $\gamma(T^*)$ (otherwise S_n is rejected by the test at line 9 in the last iteration). Hence, by P1, it outputs only minimal consistent sets S such that $T^* \cup S$ is inconsistent.

With respect to completeness, let's note that the test at line 7 rejects inconsistent solutions and the test at line 5 rules out solutions that would trivially be non minimal. However, in principle, the algorithm could be incomplete, if a set S_i is discarded because it is not admissible wrt $\{C_1, ..., C_i\}$, but it is admissible wrt $\{C_1, ..., C_{i+1}\}$ (or, in general, it can be extended to a consistent admissible covering of $\{C_1, ..., C_n\}$). For instance, consider the sets $C_1 = \{p, q\}, C_2 = \{q, r\}, C_3 = \{p, s\}$, and let's assume that $S_1 = \{p\}$ (p is chosen at the first iteration) and $S_2 = \{p, q\}$. At this stage, the algorithm fails because S_2 is not admissible wrt $\{C_1, C_2\}$. Backtracking on the choice point of stage 2 yields $S'_2 = \{p, r\}$; at the third iteration nothing is added to S'_2 and it is output as a solution. However, if S_2 had not been rejected, at the next iteration it would be recognized as admissible wrt $\{C_1, C_2, C_3\}$. It seems that a correct solution has been lost. But the algorithm can still backtrack on the choice at the first stage: q can be chosen from C_1, i.e. $S'_1 = \{q\}$; nothing is added at the second iteration ($S'_1 \cap C_2 \neq \emptyset$), and finally p is added to S'_1 giving a second admissible solution $\{p, q\}$, thus recovering the apparently lost one.

This holds in general: it can be shown that

P2. If a set S_0 of literals is a consistent non-admissible covering of $\{C_1, ..., C_k\}$ and there exists $S \supseteq S_0$ that is admissible wrt $\{C_1, ..., C_k, C_{k+1}\}$, then there exists a consistent admissible covering $S_1 \subseteq S$ of $\{C_1, ..., C_k\}$.

Using P1 and P2, it can be proved that the algorithm is complete. In fact, if $\Gamma(T^*) = \{C_1, ..., C_n\}$, an inductive reasoning shows that, for any $i = 1, ..., n$, if S_i is a consistent and minimal covering of $C_1, ..., C_i$, then there exist choices of literals at iterations $1, ..., i$ such that S_i is built at stage i. The completeness proof, as well as proofs of P1 and P2, can be found in the technical report available at [7].

It is worth discussing alternative algorithms that would still be correct and complete. The simplest variant one can think of consists in first computing a consistent covering of $\{C_1, ..., C_n\}$ and then make it minimal eliminating unnecessary literals. Or, similarly, the algorithm presented above can be modified at line 9, in such a way that, if S_i is not admissible, then all redundant literals are removed from S_i before going on (without failing). A second alternative consists in failing at line 9 only if S_i is not admissible wrt $\{C_1, ..., C_i, ..., C_n\}$. Showing completeness of such variants is certainly easier, but in both cases it is very likely, even in simple cases, to produce several times the same solution. In order to avoid repetitions, each solution should be compared to all the previously found ones before it is given in output. The algorithm used by Logitest has not been proved to avoid such redundancies; however, in the experiments carried out, it did not happen to output the same solution more than once (run on an abductive problem with almost 5,000 solutions, no duplicates were found).

4 Concluding remarks

The system described in this paper has been satisfactorily used by the author to design logical test items. Obviously, the user is assumed to have some basic logical notions in order to give adequate specifications in the input file.

Logitest can easily be improved by some routine work, in order to allow, for instance, a compact specification of some frequently used predicate properties (reflexivity, functionality, being an order relation, etc.), the possibility to define different belief sets, and the generation of literals that are consistent/inconsistent with the beliefs. A further step to make it more usable is the implementation of a graphical user interface.

Future work may finally include the study of common "logical misconceptions", so that belief sets, and consequently distractors, can be automatically generated.

Acknowledgements. The author thanks Serenella Cerrito for being willing to discuss the details of the abduction algorithm.

References

[1] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[2] J. Carneson, G. Delpierre, and K. Masters. Designing and managing multiple choice questions. `http://web.uct.ac.za/projects/cbe/mcqman/mcqman01.html`.

[3] M. Cialdea Mayer and F. Pirri. First order abduction via tableau and sequent calculi. *Bulletin of the Interest Group in Pure and Applied Logics (IGPL)*, 1:99–117, 1993.

[4] V. L. Clegg and W. E. Cashin. Improving multiple-choice tests. Idea paper no. 16. Kansas State University: Center for Faculty Evaluation and Development, 1986.

[5] Thomas Eiter, Georg Gottlob, and Technische Universitt Wien. The complexity of logic-based abduction. *Journal of the ACM*, 42:3–42, 1995.

[6] X. Leroy. The Objective Caml system, release 3.11. Documentation and user's manual. `http://caml.inria.fr/`, 2008.

[7] Logitest web page. `http://cialdea.dia.uniroma3.it/logitest/`.