CS5011 Artificial Intelligence Practice: Assessment 2
Logic

170004680

University of St. Andrews

March 2021

Contents

A Introduction **1**
A.1 Checklist . . . 1
A.2 Compilation & Execution Instructions . . . 1

B Design, Implementation and Evaluation (Word Count: ∼1550) **2**
B.1 Design and Implementation . . . 2
B.1.1 Part 1 - Procedural Constraint Checking . . . 2
B.1.2 Part 2 - Procedural Deductions . . . 2
B.1.3 Part 3 - Declarative Solver . . . 3
B.1.4 Part 4 - Hint System . . . 4
B.2 Evaluation . . . 4
B.2.1 Procedural and Declarative Logic Implementation Differences/Insights . . . 4
B.2.2 Hint System Evaluation . . . 4

C Test Summary **5**
C.1 Part 1 Testing . . . 5
C.2 Part 2 Testing . . . 6
C.3 Part 3 Testing . . . 7
C.4 Part 4 Testing . . . 8

D Bibliography **8**

A Introduction

In this assessment, logical (procedural and declarative) techniques have been used to implement a solver and hint system for the '*Easy As ABC*' puzzle.

The solver and hint system can be executed by following the instructions in section A.2. The design and implementation of the system is reported in section B.1. An evaluation is carried out in section B.2 to compare the use of procedural and declarative methods in AI logic. Robust correctness testing of the system is evidenced in section C.

A.1 Checklist

Table 1: Table indicating the completeness of this submission with respect to the specification requirements.

Part	Section	Status
1 (Procedural Constraint Checking)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
2 (Procedural Deductions)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
3 (Declarative Solver)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.
4 (Hint System)	Code	Attempted, Tested, Fully Working.
	Report	Attempted, Fully Reported.

A.2 Compilation & Execution Instructions

From within the `A2src/` directory, the `A2main` program can be compiled using (see figure 1):

export CLASSPATH=.:antlr-runtime-4.8.jar:logicng-2.0.2.jar:$CLASSPATH

javac A2main.java */*.java

After compiling the program in the `A2src/` directory, `A2main` can be executed using (see figure 2):

java A2main <TEST|SOLVE|HINT> <test-set|problem-to-solve|problem-get-hints-for>

, where the program parameters are as described in the practical specification.

Figure 1: The following screenshot shows the program can be compiled within the St. Andrews School of Computer Science lab environment without error. All of the `.java` files have been compiled into `.class` files usable for program execution.

```
/A2/A2src $ export CLASSPATH=.:antlr-runtime-4.8.jar:logicng-2.0.2.jar:$CLASSPATH
/A2/A2src $ javac A2main.java */*.java
```

Figure 2: The following screenshot shows the program can be executed within the St. Andrews School of Computer Science lab environment without error. As an example, providing '*TEST*' and '*Staff1*' as arguments runs the staff tests for part 1.

```
np57@pc3-010-l:~/Documents/CS5011/A2-local/A2/A2src $ java A2main TEST Staff1
passed Test Part 1-1 type isConsistent
passed Test Part 1-2 type isFullGrid
```

B Design, Implementation and Evaluation (Word Count: ∼1550)

B.1 Design and Implementation

B.1.1 Part 1 - Procedural Constraint Checking

The `isFullGrid` implementation iterates over every square and verifies the square contains a valid symbol. It could be verified that the unfilled character is not in the grid (more efficient), but the current implementation better captures the puzzle logic and is better suited for future work (e.g., adapting to variants like 'Not As Easy As ABC').

A grid is consistent iff:

1. Every assigned letter is unique within its corresponding row and column.
   If any pair of characters in a row/column are equal, then not all letters are unique to the row/column. Searching through pairs of letters has improved efficiency by not double-checking pairs.

2. Every full row/column contains every puzzle letter.
   All rows and columns are checked together, taking advantage of the square grid dimensions. Column retrieval via an index was added. Thus, operations performed on both rows and columns could use a single method; row and column constraints are symmetric. Checking whether a row/column is full and has no missing letters is trivial by considering the row/column as a string and using standard, optimised manipulation methods.

3. All clues are adhered to or can be adhered to with further assignments.
   The first 'visible' character to every clue must be the clue letter (adjacent to the clue or separated only by blanks) or the unfilled character (may become filled with the clue letter).

B.1.2 Part 2 - Procedural Deductions

The following deductions have been implemented:

1. If any corner has different adjacent clues, there must be a blank in that corner square.
   For all four corner squares, the clues adjacent to each square are checked for letters. If they are letters and differ, then the corresponding corner of the grid is set to blank. If any clue adjacent to a corner is unfilled, then the deduction is not applied to that corner.

2. If any square is the only unfilled square in a line, and one letter does not appear in that line, then the letter must go in that square.
   Every line is iterated over and considered as a string. The set of unique characters within the string is retrieved. If this set contains the unfilled character, then there is a single unfilled square in the line. The set of puzzle letters is iterated over to determine if there is a single letter missing in the line. If so, the unfilled square is set to the missing letter.

3. If every letter appears in a line (once), then the remaining squares in the line must be blank.
   For each line, the set of unique characters in each line is retrieved as with deduction 2. If this set has no letters missing, then all unfilled squares within the line are set to blank.

4. If a clue set is full and contains only one occurrence of any letter, that letter must appear in the first adjacent square to the clue.
   The sets of clues are considered as strings. If a clue string contains an unfilled character (the clue set is not full), then this deduction is not applied. Otherwise, the unique letters within the clue set are retrieved. Each letter in this set appears once in the clue set, so the adjacent square to the clue is set to the clue letter.

The deductions are symmetric for both rows and columns. This allows for methods to be reused irrespective of whether the given line represents a row or column. As a result, the deduction implementations are simpler.

B.1.3 Part 3 - Declarative Solver

A propositional variable is created for every combination of square and character (letter, blank, or unfilled). Variable names have the form rcs: r is the row index, c is the column index, and s is the symbol. If the variable '$00a$' is true, then the square at $(0, 0)$ contains the letter 'a'.

Data structures are used to manage the propositional variables: `rowsToLiterals`, `colsToLiterals`, and `symbolsToLiterals`. These data structures map rows, columns, and symbols to the set of propositional variables corresponding to each row, column and symbol respectively. This simplified the implementation of the constraints considerably.

The following definitions are used: r is the current row, c is the current column, v is the current character index, n is the number of rows (also column as grid is square), and k is the number of characters (symbols and the unfilled character). Also, `rowsToLiterals[i]` refers to the set of variables corresponding to row i, taken from the mapping. This notation is identical for `colsToLiterals` and `symbolsToLiterals`.

The following constraints are formed as a set of `clauses` to provide a CNF formula to the MiniSAT solver. Puzzle constraints are represented as clauses as this results in the simplest implementation, since the structures of the constraints become similar.

1. Every square contains at least one symbol.
   For every square (r, c) in the grid, the variables associated with the square are retrieved by taking the intersection ($\cap$) of `rowsToLiterals[r]` and `colsToLiterals[c]`. The disjunction of these intersection variables is added to `clauses`.
   $$C_1 = \bigwedge_{1 \leq r,c \leq n} (rc1 \vee rc2 \vee ... \vee rck)$$
2. Every square contains at most one symbol.
   The variables associated with each square (r, c) are `rowsToLiterals[r]` $\cap$ `colsToLiterals[c]`. For every pair of variables (rcv, rcv'), the clause $(\neg rcv \vee \neg rcv')$ is added to `clauses`.
   $$C_2 = \bigwedge_{1 \leq r,c \leq n, 1 \leq v < v' \leq k} (\neg rcv \vee \neg rcv')$$
3. Every row/column has every letter at least once.
   For rows, we take `rowsToLiterals[r]` $\cap$ `symbolsToLiterals[v]`, for all rows (r) and all symbols (v). The disjunction of the variables in the intersection is added to `clauses`. This is similar for columns, where `colsToLiterals[c]` $\cap$ `symbolsToLiterals[v]` is used.
   $$C_{3rows} = \bigwedge_{1 \leq r \leq n, 1 \leq v \leq k} (r1v \vee r2v \vee ... \vee rnv)$$
   $$C_{3cols} = \bigwedge_{1 \leq c \leq n, 1 \leq v \leq k} (1cv \vee 2cv \vee ... \vee ncv)$$
4. Every row/column has every letter at most once.
   For rows, we take `rowsToLiterals[r]` $\cap$ `symbolsToLiterals[v]`, for all rows (r) and all symbols (v). For every pair of intersection variables $(rcv, rc'v)$, the clause $(\neg rcv \vee \neg rc'v)$ is added to `clauses`. This is similar for columns, where `colsToLiterals[c]` $\cap$ `symbolsToLiterals[v]` is used.
   $$C_{4rows} = \bigwedge_{1 \leq r \leq n, 1 \leq v \leq k, 1 \leq c < c' \leq n} (\neg rcv \vee \neg rc'v)$$
   $$C_{4cols} = \bigwedge_{1 \leq c \leq n, 1 \leq v \leq k, 1 \leq r < r' \leq n} (\neg rcv \vee \neg r'cv)$$
5. When a clue is given for a row/column, the first 'visible' symbol in that row/column is the clue.
   For each row/column associated with each clue, a constraint is formed dictating that the clue letter must appear in the first square, or following squares along the row/column as long as only blanks are between the clue and the clue letter. Efficiency is improved by taking advantage of the fact that there is a maximum number of blanks permitted in these constraints.

B.1.4 Part 4 - Hint System

A procedural-based hint system has been implemented which repeatedly applies a series of deductions until either a solution is found or a dead-end is reached. The deductions are applied in an order designed to closely mimic how a person would solve the puzzle. For example, by applying letters to squares with certainty via the clues before moving to more complex techniques. As a result, produced sets of hints are more understandable; the ordered hints have a comprehensible chain of reasoning using deductive techniques a person would consider.

Additionally, hints are provided with informative explanations. The explanations closely resemble the deductive reasoning used to form each clue within the context of the current grid state. Thus, clues are even more understandable as the reasoning behind any hint assignments is explained.

B.2 Evaluation

B.2.1 Procedural and Declarative Logic Implementation Differences/Insights

One key insight when implementing procedural and declarative approaches is that there is no well-defined boundary. The declarative approach required a specialist program to express the puzzle as logical statements. The procedural approach enforced logical constraints for the puzzle, except by manipulating data structures as opposed to expressing a propositional logic sentence.

A procedural approach can be made more efficient than a deductive approach; significant search for a solution can be averted by applying puzzle-specific deductions, as opposed to the general declarative approach which exhaustively explores a SAT search tree.

The declarative approach is guaranteed to find solutions when they exist. The procedural approach makes no such guarantee. The set of deductions implemented procedurally may not be complete, so a solution to every solvable puzzle may not be found. This is the responsibility of the programmer to investigate, whilst the programmer is only responsible for correctly expressing the puzzle constraints in a declarative approach.

Procedural methods embed the rules and logic of the puzzle within code. If any changes to the puzzle rules are required, then significant alterations may be needed. Thus, the complexity of making changes to the implementation can be greater for a procedural approach. However, the opposite may also be true. It is possible for changes to the puzzle to be trivially implemented with procedural methods, whilst the changes may be significantly difficult to add to a declarative approach due to the model used for representing the puzzle.

B.2.2 Hint System Evaluation

The hint system provides a good level of comprehensibility for produced hints (figure 3). Use of procedural deductions allowed for the ordering of hints and hint explanations to be tailored, maximising comprehension by mimicking a human solver. However, the implemented system is a basic foundation for future work given its clear limitations.

Produced hints are highly understandable; however, the hint system is limited by the relatively small amount of puzzles that can be solved without (declarative) search. This is a significant drawback of the system (figure 4). Future work should improve the current system by implementing search when no further deductions can be applied. The search can be prioritised by squares that have the fewest candidates, which is how a human would solve the puzzle. Alternatively, further deductions can be added with robust explanations until a complete set of deductions is implemented. Note that whilst it would have been trivial to implement declarative search to find a solution when there is a dead-end, doing so while ensuring the quality of the provided hints is not trivial.

Furthermore, whilst the ordering of deductions has been implemented to mimic a human for increased comprehension, this ordering is fixed. Further work should consider making the ordering of techniques dynamic by using heuristics about the current state of the grid. This sophistication can result in shorter, more understandable hint chains as better successive hints can be selected.

Figure 3: The following screenshots illustrate the sets of hints given by the hint system. The provided hints are given using an ordering of techniques that a human would prioritise. Also, detailed explanations of the techniques forming each clue within the context of the current grid state are provided. The top screenshot shows the produced hints for the provided '*VeryEasy*' instance. The bottom screenshot shows the produced hints for the '*StillEasy*' instance. Both clue sets are correct.

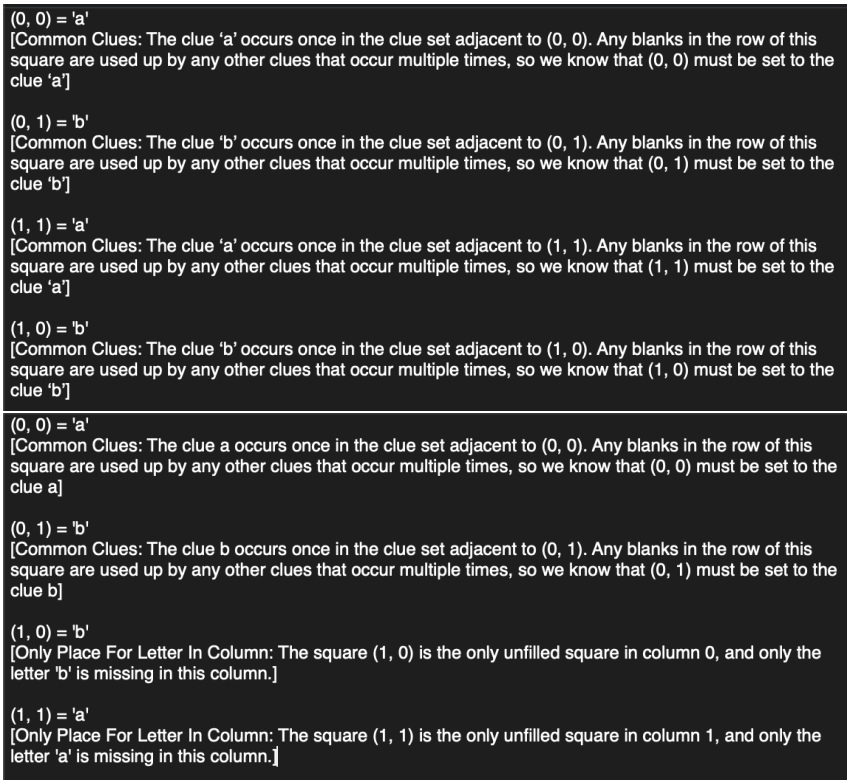


Figure 4: The following screenshot shows the result for the provided '*ABC1*' puzzle. This puzzle is solvable and not very difficult but still could not be solved by the hint system. The provided hint system is very basic and future work should add more deductions and/or implement some form of solution search.

C Test Summary

This section outlines core testing carried out on the implementations of all parts of this assessment to ensure correctness and conformity with the specification.

All tests provided with the starter code (`TestsStaff`) are passed by the implementations. Additional test sets have been provided (`TestsStudent`) that robustly test the implementations on a method-by-method basis. These tests better ensure the correctness of the system as behaviour is checked via chosen normal, edge, and extreme cases across the input space. The following sections outline specific testing carried out for each part of the assessment.

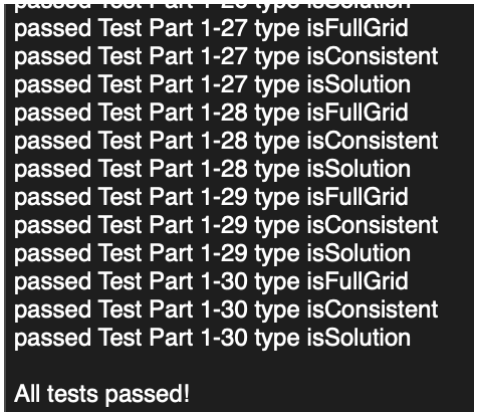
C.1 Part 1 Testing

Table outlining the correctness testing carried out for Part 1 (figure 5). All of these tests have been implemented as a test set that can be checked by specifying '*Student1*' as a program argument with '*TEST*'.

Test Description	**Passed?**
isFullGrid(): True when grid is full of puzzle letters.	Yes
isFullGrid(): True when grid is full of blank letters.	Yes
isFullGrid(): True when grid is full of puzzle and blank letters.	Yes

Test Description	Passed?
isFullGrid(): False when grid contains non-puzzle letters.	Yes
isFullGrid(): False when grid contains unfilled squares.	Yes
isConsistent(): False when multiple of same letter in a row.	Yes
isConsistent(): False when multiple of same letter in a column.	Yes
isConsistent(): True when an unfilled square does not break the clue constraints.	Yes
isConsistent(): True when the first 'visible' letter is the clue (and all other consistency conditions met).	Yes
isConsistent(): True when the first 'visible' letter is the clue, even when there are blanks (and all other consistency conditions met).	Yes
isConsistent(): False when the first 'visible' letter is not the clue letter.	Yes
isConsistent(): True when full rows contains all letters once (and all other consistency conditions met).	Yes
isConsistent(): True when full columns contains all letters once (and all other consistency conditions met).	Yes
isSolution(): False when grid is consistent but not full.	Yes
isSolution(): False when grid is full but not consistent.	Yes
isSolution(): False when grid is not full and not consistent.	Yes
isSolution(): True when grid is full and consistent.	Yes

Figure 5: The following screenshots shows that all tests implemented for part 1 correctly passed.



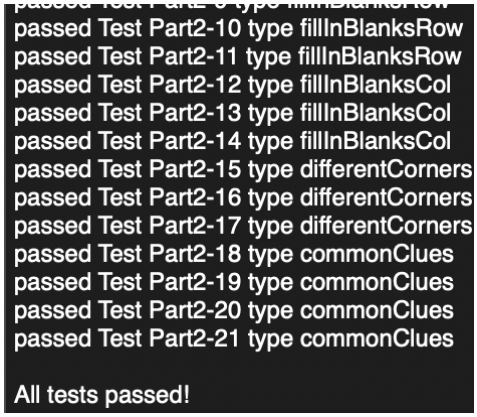
C.2 Part 2 Testing

Table outlining the correctness testing carried out for Part 2 (figure 6). All of these tests have been implemented as a test set that can be checked by specifying '*Student2*' as a program argument with '*TEST*'.

Test Description	Passed?
When there is one unfilled square and one remaining letter (in a row/column), then fill the square with the letter.	Yes
When there are multiple unfilled squares and one remaining letter (in a row/column), then do not fill.	Yes
When there is one unfilled square but multiple remaining letters (in a row/column), then do not fill.	Yes
When there are multiple unfilled squares and letters (in a row/column), then do not fill.	Yes
When all puzzle letters are in a row/column, then remaining unfilled squares are set to blank.	Yes
When not all puzzle letters are in a row/column, then do not fill unfilled squares with blank.	Yes

Test Description	Passed?
When in any corner the adjacent clues differ, then fill the corner with blank.	Yes
When in any corner the adjacent clues match, then do not fill the corner with blank.	Yes
When a clue occurs once in a clue set, the neighbouring square in the grid is set to the letter.	Yes
When a clue occurs multiple times in a clue set, the neighbouring square is not set to the letter.	Yes
When a clue set contains an unfilled character, then do not set letters in the neighbouring squares if clues occur once.	Yes

Figure 6: The following screenshots shows that all tests implemented for part 2 correctly passed.

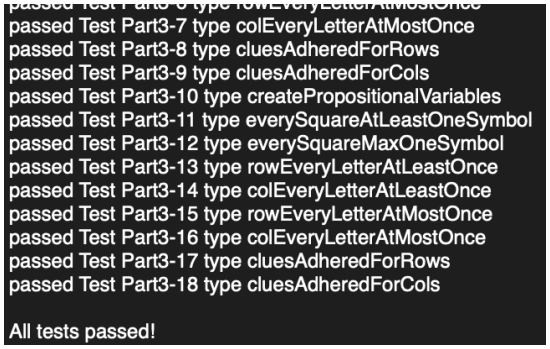


C.3 Part 3 Testing

Table outlining the correctness testing carried out for Part 3 (figure 7). All of these tests have been implemented as a test set that can be checked by specifying '*Student3*' as a program argument with '*TEST*'.

Test Description	Passed?
Check propositional variables are created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every square contains at least one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every square contains at most one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every row contains at least one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every row contains at most one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every column contains at least one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'every column contains at most one symbol' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'clues are adhered to on rows' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes
Check that 'clues are adhered to on columns' constraints created correctly for both a 1x1 and an NxN puzzle instance.	Yes

Figure 7: The following screenshots shows that all tests implemented for part 3 correctly passed.

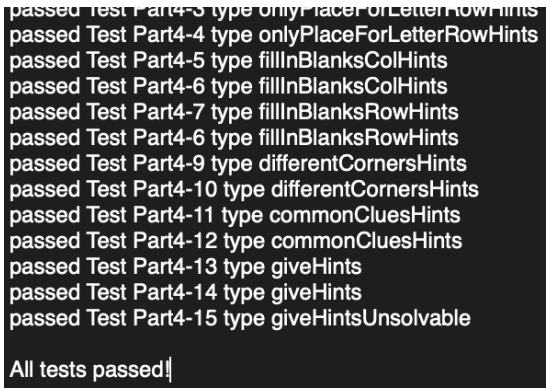


C.4 Part 4 Testing

Table outlining the correctness testing carried out for Part 4 (figure 8). All of these tests have been implemented as a test set that can be checked by specifying '*Student4*' as a program argument with '*TEST*'.

Test Description	**Passed?**
Check that `onlyPlaceForLetterCol` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that `onlyPlaceForLetterRow` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that `fillInBlanksCol` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that `fillInBlanksRow` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that `differentCorners` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that `commonClues` returns the correct amount of hints (instances where no hints are expected and where multiple hints are expected).	Yes
Check that explanations given by the hint system are correct and as expected.	Yes
Check that a correct and complete set of hints are provided for a trivial puzzle case.	Yes
Check that a correct and complete set of hints are provided for an easy puzzle case.	Yes
Check that a correct and complete set of hints are provided for a less easy puzzle case.	Yes
Check that no hints are provided for an unsolvable (by the method) puzzle case and that this is indicated to the user appropriately.	Yes

Figure 8: The following screenshots shows that all tests implemented for part 4 correctly passed.



D Bibliography

The required content for this assessment was acquired from the specification and module material.