



Artificial Intelligence

Laboratory activity

Name: Fodor, Zsófia and Katona, Áron

Group: 30433

Email: fodorzsofi@yahoo.com katonaaron01@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro



Contents

1	A1: Search	5
1.1	Introduction	5
1.2	Bidirectional Search	5
1.2.1	Implementation	5
1.3	Iterative Deepening Depth-First Search	6
1.3.1	Depth-Limited Search	6
1.3.2	Iterative Deepening Search	6
1.3.3	Iterative Deepening Depth-First Search	7
1.4	Iterative Deepening A* Search	7
1.4.1	Implementation	7
1.4.2	Remarks	7
1.5	Recursive Best-First Search	8
1.6	Comparison	8
1.6.1	Bidirectional Search vs Breadth-First Search	8
1.6.2	Iterative Deepening Depth-First Search	8
1.6.3	Iterative Deepening A* vs A*	9
1.6.4	Recursive Best-First Search	9
2	A2: Logics	11
2.1	Introduction	11
2.2	Logic via algebra	11
2.3	Solving lady and tigers with algebra	12
2.3.1	The first trial	14
2.3.2	The Second Trial	15
2.3.3	The Third Trial	16
2.3.4	The Fourth Trial	17
2.3.5	The Fifth Trial	18
2.3.6	The Sixth Trial	19
2.3.7	The Seventh Trial	21
2.3.8	The Eighth Trial	22
2.3.9	The Ninth Trial	23
2.3.10	The Tenth Trial	25
2.3.11	Representation for empty rooms	27
2.3.12	First, Second, and Third Choice	27
2.4	Solving logic puzzles programatically	30
2.5	Translating propositional logic to modular arithmetic programatically	31
2.5.1	Data representation	31
2.5.2	Parsing the input file	32
2.5.3	Defining the rules	32
2.5.4	Creating expression trees from propositions	33

2.5.5	Creating and transformation of expressions	34
2.5.6	Printing the result	35
2.5.7	The main program	35
2.6	Usage of the translator program	35
2.6.1	Input and output	36
2.6.2	Refining the output	36

A Source Code: Search 42

A.1	Nodes	42
A.2	Bidirectional Search	43
A.3	Depth Limited Search	44
A.4	Iterative Deepening Depth-First Search	45
A.5	Iterative Deepening A*	45
A.6	Recursive Best-First Search	46

B Source Code: Logic Puzzles 49

B.1	The First Trial	49
B.1.1	Propositional logic	49
B.1.2	Modular arithmetic	50
B.2	The Second Trial	50
B.2.1	Propositional logic	50
B.2.2	Modular arithmetic	51
B.3	The Third Trial	51
B.3.1	Propositional logic	51
B.3.2	Modular arithmetic	52
B.4	The Fourth Trial	52
B.4.1	Propositional logic	52
B.4.2	Modular arithmetic	53
B.5	The Fifth Trial	54
B.5.1	Propositional logic	54
B.5.2	Modular arithmetic	55
B.6	The Sixth Trial	55
B.6.1	Propositional logic	55
B.6.2	Modular arithmetic	56
B.7	The Seventh Trial	56
B.7.1	Propositional logic	56
B.7.2	Modular arithmetic	57
B.8	The Eighth Trial	58
B.8.1	Propositional logic	58
B.8.2	Modular arithmetic	59
B.9	The Ninth Trial	59
B.9.1	Propositional logic	59
B.9.2	Modular arithmetic	60
B.10	The Tenth Trial	61
B.10.1	Propositional logic	61
B.10.2	Modular arithmetic	62
B.11	First, Second, and Third Choice	63
B.11.1	Propositional logic	63
B.11.2	Modular arithmetic	64

C	Source Code: Translator	66
C.1	Input data representation	66
C.2	Parsing the Mace4 input file	67
C.3	Creating/transforming Boolean ring expression trees	69
C.4	Printing the result	75
C.5	Main program	76

Chapter 1

A1: Search

1.1 Introduction

Our aim is to implement multiple search algorithms, and compare them with themselves and with the ones that were already discussed: A*, DFS, BFS, UCS.

We chose the following searching strategies:

Uninformed

- Bidirectional Search
- Depth Limited Search
- Iterative Deepening Depth-First Search

Informed

- Iterative Deepening A*
- Recursive Best-First Search

1.2 Bidirectional Search

Bidirectional search simultaneously searches from the start state to the goal state (**forward searching**) and from the goal state to the start state (**backward searching**) hoping that these two searches will meet. The time and space complexity was reduced from $O(b^d)$ to $O(2 * b^{d/2}) = O(b^{d/2}) \ll O(b^d)$, where b is the branching factor and d is the depth of the shallowest solution.

1.2.1 Implementation

The algorithm uses the **Breadth-First Search policy**, meaning that it takes the shallowest node first. It stops when the forward and backward search intersect. If no solutions is found, than it returns *None*.

We used two sets one representing the explored nodes that are still in the queue and another one, that contains the visited nodes that were popped out of the queue. Entering the while loop, we firstly analyze the forward search. Take the next node from the corresponding queue and if it is not yet visited we put it in the visited set. After we have iterated through the nodes

of the backward search, if there is a meeting point then we return the current path and the path accumulated by the backward search in reverse order.

Next we analyze the backward search the same way as we did for the forward search, the only difference being that we replace each action with its opposite action, we reverse the list containing them and add it to the accumulated path.

The motivation behind using sets is that they are more efficient than lists in verifying whether they contain an arbitrary element.

Source code: Appendix A.2.

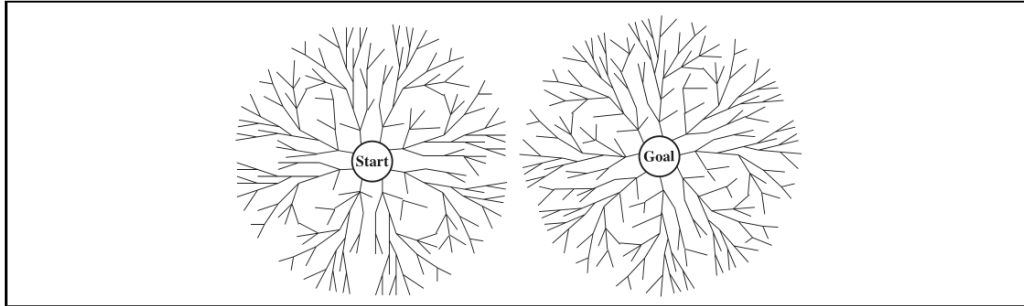


Figure 1.1: Visualization of the bidirectional search algorithm. [4]

1.3 Iterative Deepening Depth-First Search

1.3.1 Depth-Limited Search

The depth-limited search is similar to the depth-first search, with the only difference, that a **depth bound** is added. Thus the algorithm searches for the goal state until the search tree's depth reaches the bound. If the bound is reached, the search backtracks to the parent node and continues searching in the next successor of the parent node. Therefore if there is a path to the goal, whose length is smaller than or equal to the bound, it will be found.

Implementation

To represent the nodes we used the `Node` class, which is a simple data class whose `state`, `action` and `cost` fields are used. More information about the class can be found in Appendix A.1.

The algorithm is a modified version of the recursive **depth-first search**. At each level of recursion, the function calls itself for all of the successors. The function returns not only the found goal node on success and `None` on error, but also an additional boolean parameter which specifies whether the search terminated because of a cutoff, or not. In each level of recursion, the limit is decremented. When it reaches 0, the returned cutoff value is true.

Source code: Appendix A.3.

1.3.2 Iterative Deepening Search

The iterative deepening search strategies apply a search algorithm multiple times, with increasing bounds until the goal is reached. The calculation of the bound depends on the applied algorithm. We used the strategy of iterative deepening for the following search algorithms: depth-first search (1.3.3) and A* (1.4).

1.3.3 Iterative Deepening Depth-First Search

Starting from an initial bound of 0, we call the depth-limited search and verify the returned result. In case a cutoff occurred, we increase the bound by one, and start again the search. If the goal node was found, the path from the start node which ends in the goal node is reconstructed and returned.

The algorithm is complete, because depth-first search, and in particular depth-limited search is guaranteed to find a solution, if the distance to the goal node is inside the bound. The bound is incremented from 0 indefinitely, thus if the solution exists, it will be found.

The algorithm is optimal for unit action costs, because if no solution was found for bound b , the length of the shortest path from the start node to the goal node is at least $b + 1$, and if there is any solution with length $b + 1$, all of them are optimal solutions, and one of them will be found by the depth-limited search.

Source code: Appendix A.4.

1.4 Iterative Deepening A* Search

Iterative deepening is a preferred algorithm when we have a larger search state space than the one that can fit in memory and the depth of our solution is not known.

Iterative Deepening A* is similar to A* the difference being that we do not keep all reached states in memory, at a cost of visiting some states multiple times. It is a very frequently used algorithm for problems that do not fit in memory, as stated above.

1.4.1 Implementation

The idea behind the algorithm was that we search the node with the lowest combined cost and heuristic first ($f = g + h$). The algorithm is limiting the size of the frontier by using this calculated f value as a bound.

We used a set in which we stored the visited nodes. Searching in sets is faster than searching in lists, that is why we chose set. Besides this set, we used a list called path, that stored for each node its state, the action that needs to be performed to get there from the previous node and the cost.

We have a utility function that we call repeatedly until the returned value is either a very large number (∞) or the *None* state and we change the value of the bound to the returned value if none of these conditions is satisfied.

The utility function begins with taking the last element from the path, computing the f value of this element and comparing it to the bound. If it is larger than the bound, we return the f value. If the state of the node is the goal state, it means we reached our goal, so we return the *None* state. If none of these conditions are fulfilled, then we iterate through each child of the current node, add it to the visited set and to the path list, then we call the utility function on it. We make the adjustments according to the returned value of this call, remove the node from the visited list and the path. Lastly we return the minimum value which represents the minimum cost of all values that exceeded the current bound.

1.4.2 Remarks

This algorithm was actually easy to implement after we understood the idea behind it.

Comparing the number of expanded nodes we can see that it is much larger than for the A* algorithm, because of the fact that we do not keep all reached states in memory risking the fact that we might visit some nodes more than once.

1.5 Recursive Best-First Search

The recursive best-first search, similarly to the depth-limited search (1.3.1), searches for the deepest node first, but stops after surpassing a bound. But in this case the bound is placed on the f-value of a node, instead of its depth.

The f-value of a node is the maximum between the f-value of its parent and the cost of reaching the node + the heuristic, i.e. $f(n) = \max\{g(n) + h(n), f(n.parent)\}$. When a branch of the search is cut off because the limit on the f-value, while backtracking, the f-value of a parent node will be updated with the f-value of its child. This way, the f-value of a node will become a more and more accurate estimation of the true cost of reaching the goal node from it.

This algorithm uses the principle of best-first searching. It only expands the node with the smallest f-value. If the node is on the current searching branch, than it will be expanded, otherwise the search tree backtracks to the level of the node with the smallest f-value. Therefore at each level of recursion we take the first two successors with the lowest f-value, call the search for the best option, and supply the bound as the f-limit of the alternative successor. This is repeated until the solution is found, or until the f-value of the best node will become greater than the f-limit. Thus it always considers a best and an alternative path, and switches between them, if the alternative path becomes the best one, in terms of lowest f-value.

The advantage of this algorithm is that it uses linear space: used for storing the nodes along the search path, and also the sibling of each node. The disadvantage is that it regenerates already visited nodes, which could happen very frequently. Thus it trades speed for storage.

For the implementation the `Node` class (A.1) was used for storing the nodes, and a priority queue was used for obtaining the best successor of a node. The recursive function returns the result and `None`, if the goal node was found. If the goal was not found, it returns `None` and the f-value of the deepest node reached before surpassing the limit.

Source code: Appendix A.6.

1.6 Comparison

Consider b the branching factor and d the depth of the shallowest solution.

1.6.1 Bidirectional Search vs Breadth-First Search

As stated in chapter 1.2, the bidirectional search reduces the size of the frontier and the running time from $O(b^d)$ to $O(b^{d/2})$. To find a solution with length d , the two frontiers contain in the worst case only the nodes with the depth $d/2$ from either nodes. Thus both the space and the running time is reduced.

However the implementation of the bidirectional search is difficult if the actions cannot be reversed fast. E.g. North \leftrightarrow South.

1.6.2 Iterative Deepening Depth-First Search

vs Depth-First Search: The depth-first search does not find always the optimal solution, and may not find any solution even if it exists in the graph. However the iterative deepening version is optimal and complete, because each path is considered in increasing order of depth.

vs Breadth-First Search: The running time has the same $O(b^d)$ complexity. For the iterative deepening depth-first search the space complexity is reduced from $O(b^d)$ to $O(bd)$, because

the deepest node is taken first. However because of the iterative deepening, the number of expanded nodes is higher.

1.6.3 Iterative Deepening A* vs A*

Iterative Deepening A* is similar to A* the difference being that we do not keep all reached states in memory, at a cost of visiting some states multiple times. It is a very frequently used algorithm for problems that do not fit in memory.

1.6.4 Recursive Best-First Search

vs A*: RBFS uses only linear space, but suffers from frequently regenerating the nodes. Given enough time it could solve those problems that could not be solved by A* because of running out of memory.

vs Iterative Deepening A* The two algorithms are solutions for the same problem of reducing the size of the frontier of A*. Both suffer from revisiting and regenerating nodes. However the RBFS is slightly more efficient, because of storing more information than the other one, thus increasing the speed.

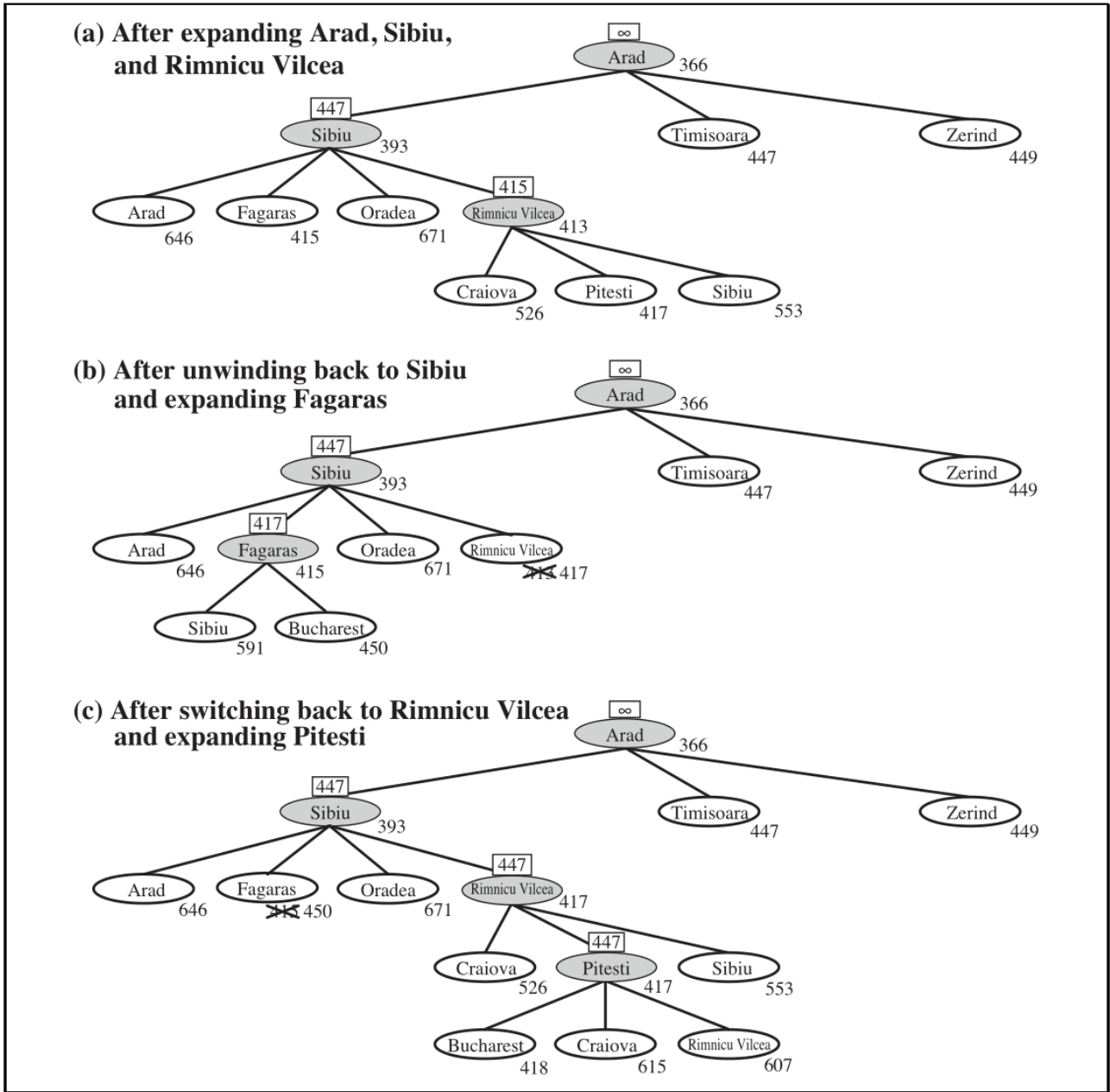


Figure 1.2: Finding a path from Arad to Bucharest by using recursive best-first search. The number above a node is its f-limit, and the number on the right of it is its f-value. The searching branch was switched two times. [4]

Chapter 2

A2: Logics

2.1 Introduction

The purpose of this chapter is to discuss the usage of modular arithmetics (Boolean rings) for solving logic puzzles, as represented in [3].

The theory is applied in solving the “The Lady or the Tiger” puzzles [5].

Also a program was written in python for translating a Mace4 input file from propositional logic to modular arithmetics, producing algebraic expressions on the Boolean ring. The output can be supplied as input for Mace4 [1].

2.2 Logic via algebra

The article [3] presents the theory about using algebra for solving logic puzzles. Below are summarized the parts that are used in this chapter.

0 and 1 represent the truth values “false” and respectively “true”. $\llbracket P \rrbracket$ represents the truth value of the proposition P .

- $\llbracket P \rrbracket = 1 \iff P \text{ is true}$
- $\llbracket P \rrbracket = 0 \iff P \text{ is false}$

The set $\{0, 1\}$ is:

- the smallest possible Boolean algebra
- a distributive bounded lattice in which every element has a complement
- a Boolean ring: $x^2 = x \forall x$
- a field of characteristic 2: $x + x = 0 \forall x$

Therefore any Boolean connective can be expressed as a polynomial. Table 2.1 presents the connectives both in lattice and ring form. This table is used for translating the statements of the logic puzzles into equations. It is also used for the program which converts a Mace4 input file from propositional logic to modular arithmetic.

A logical puzzle is composed of statements in which an information about A is equivalent with a predicate P . Can be written as:

$$a = p$$

CONNECTIVE	LATTICE form	RING form
Conjunction (AND)	$p \wedge q$	pq
Exclusive disjunction (XOR)	$(p \vee q) \wedge \neg(p \wedge q)$	$p + q$
Inclusive disjunction (OR)	$p \vee q$	$pq + p + q$
Negation	$\neg p$	$p + 1$
Implication (\rightarrow)	$q \vee \neg p$	$pq + p + 1$
Biconditional (\leftrightarrow)	$(p \wedge q) \vee (\neg p \wedge \neg q)$	$p + q + 1$
Sheffer stroke (NAND)	$\neg(p \wedge q)$	$pq + 1$
Pierce's arrow (NOR)	$\neg(p \vee q)$	$pq + p + q + 1$

Table 2.1: Boolean connectives in the language of Boolean rings [3]

Where $p = \llbracket P \rrbracket$ and a has the truth value of that information about A, e.g. $a = \llbracket "A \text{ is a knight}" \rrbracket$. Thus the puzzle can be written as a system of equations:

$$\begin{cases} a_1 = p_1 \\ \dots \\ a_n = p_n \end{cases}$$

And can be reduced to the equation:

$$\prod_{i=1}^n (p_i + a_i + 1) = 1 \quad (2.1)$$

2.3 Solving lady and tigers with algebra

In this section “The Lady or Tiger” puzzles are solved using both propositional logic and modular arithmetics in order to compare the two methods.

Mace4

For verifying the correctness of the formulas written to solve the puzzle, we used mace4. Mace4 can solve the puzzles for both types of notations.

For the modular arithmetic input files we specified the following commands at the beginning, in order to configure arithmetic operations on the Boolean ring:

```
set(arithmetic).
assign(domain_size, 2).
```

However, mace4 has a flaw when using it for expressions on the Boolean ring. There are some cases in which mace4 does not give any result. To solve this, one must wrap those expressions with “mod 2”. An example can be seen below:

Input file that can be solved by Mace4

```
set(arithmetic).
assign(domain_size, 2).
assign(max_models, -1).

formulas(assumptions).
    m1 + m2 = 1.
end_of_list.
```

Mace4 output

```
interpretation( 2, [number = 1,
    seconds = 0], [
    function(m1, [0]),
    function(m2, [1])]).
interpretation( 2, [number = 2,
    seconds = 0], [
    function(m1, [1]),
    function(m2, [0])]).
```

Input file that cannot be solved by Mace4

```
set(arithmetic).
assign(domain_size, 2).
assign(max_models, -1).

formulas(assumptions).
    m1 + m2 + 1 = 0.
end_of_list.
```

Mace4 output

```
=== Mace4 starting on domain size 2.
===

----- process 62347 exit (exhausted)
-----
```

Input file made to be solvable by Mace4

```
set(arithmetic).
assign(domain_size, 2).
assign(max_models, -1).

formulas(assumptions).
    (m1 + m2 + 1) mod 2 = 0.
end_of_list.
```

Mace4 output

```
=== Mace4 starting on domain size 2.
===

----- process 62531 exit (
    all_models) -----
interpretation( 2, [number = 1,
    seconds = 0], [
    function(m1, [0]),
    function(m2, [1])] ).
interpretation( 2, [number = 2,
    seconds = 0], [
    function(m1, [1]),
    function(m2, [0])] ).
```

For simplicity, we will omit the wrapping by “mod 2” in this chapter.
The source files in the Mace4 input format can be found in annex B.

Representation

In **propositional logic** we can represent the state of the i -th room by the following predicates:

- li : There is a lady in room i .
- ti : There is a tiger in room i .
- mi : Message on the door of room i

Then one must specify, that if the room contains a lady, it does not contain a tiger, and viceversa.

```
11 -> -t1.
12 -> -t2.
```

For representing the rooms in **modular arithmetics** we can define:

- $ri = [[\text{“There is a lady in room } i \text{”}]] = 1 + [[\text{“There is a tiger in room } i \text{”}]]$
- $mi = [[\text{“Message on the door of room } i \text{”}]]$

In this representation the appearance of a lady and a tiger in the same room is implicitly exclusive.

These representations will be used in the following puzzles, unless it is specified otherwise.

2.3.1 The first trial

Knowledge base

The following statements are given:

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** In this room there is a lady, and in the other room there is a tiger
3. **Message on door 2:** In one of these rooms there is a lady, and in one of these rooms there is a tiger
4. One of the messages is true, but the other one is false

The first statement enumerates all four possibilities. It does not give more information than the one included in the representation.

The second statement represents the truth value of the first message. It can be written the following forms:

Propositional logic

$$m1 \leftrightarrow l1 \& t2.$$

Modular arithmetics

$$m1 = r1 * (r2 + 1).$$

One can observe that logical equivalence was replaced with equality, conjunction with multiplication, and negation with an addition by one.

The second message states that at least one tiger and at least one lady exists. Because there are only two rooms, this means that one room contains a lady, and the other one contains the tiger. Thus we can use the XOR operator between $r1$ and $r2$ to force exclusivity, which in the case of the modulo 2 algebra it is represented by the "+" symbol.

Propositional logic

$$m2 \leftrightarrow (l1 \mid l2) \& (t1 \mid t2).$$

Modular arithmetics

$$m2 = r1 + r2.$$

As for the fourth statement, we simply need to specify that the two messages are not equal, i.e. the first is equal with the negation of the second. Or we can say that the two does not take the same value simultaneously (XOR).

Propositional logic

$$m1 \leftrightarrow \neg m2.$$

Modular arithmetics

$$m1 + m2 = 1.$$

Resolution

The advantage of the modular arithmetics is that it can be resolved by algebraic methods, taking into consideration the rules of the boolean ring.

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * (r2 + 1) & (2.2a) \\ m2 = r1 + r2 & (2.2b) \\ m1 + m2 = 1 & (2.2c) \end{cases}$$

By replacing $m1$ and $m2$ in 2.2c we obtain:

$$r1 * (r2 + 1) + r1 + r2 = 1$$

Which can be rewritten as

$$r2 * (r1 + 1) + r1 + r1 = 1$$

Knowing that $x + x = 0$ and $x + 0 = x$, we can omit the $r1 + r1$ term from the equation and obtain:

$$r2 * (r1 + 1) = 1$$

In order to satisfy the equation, both operands of the multiplication must be 1. Otherwise the product would be 0. Therefore the result will be:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

2.3.2 The Second Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** At least one of these rooms contains a lady.
3. **Message on door 2:** A tiger is in the other room.
4. The messages are either both true or both false.

The first message tells us, that either one of the rooms contains a lady, or both of the rooms contain a lady.

Propositional logic

$$m1 \leftrightarrow (l1 \mid l2).$$

Modular arithmetics

$$m1 = r1 * r2 + r1 + r2.$$

One can observe that the logical OR operator is replaced by multiplying the two terms, then adding each one of them.

The second message tells us, that in the other room (i.e.: the first room) there is a tiger. Here the negation will be replaced with an addition by one.

Propositional logic

$$m2 \leftrightarrow t2.$$

Modular arithmetics

$$m2 = r1 + 1.$$

For the last statement we simply have to specify that the truth value of the two messages is the same.

Propositional logic

$$m1 \leftrightarrow m2.$$

Modular arithmetics

$$m1 = m2.$$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * r2 + r1 + r2 & (2.3a) \\ m2 = r1 + 1 & (2.3b) \\ m1 = m2 & (2.3c) \end{cases}$$

By replacing $m1$ and $m2$ in 2.3c we obtain:

$$r1 * r2 + r1 + r2 = r1 + 1$$

Which can be rewritten as:

$$r2 * (r1 + 1) = 1$$

In order to satisfy the equation, both operands of the multiplication must be 1. Otherwise the product would be 0. Therefore the result will be:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

2.3.3 The Third Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** Either a tiger is in this room or a lady is in the other room.
3. **Message on door 2:** A lady is in the other room.
4. The messages are either both true or both false.

The first message tells us, that either a tiger is in the first room, or a lady is in the second room. By double negation and De Morgan's Law we obtain: "It's not true that a lady is in the first room and a tiger in the second".

Propositional logic

$m1 \leftrightarrow t1 \mid l2.$

Modular arithmetics

$m1 = r1 * (r2 + 1) + 1.$

The second message tells us, that in the other room (i.e.: the first room) there is a lady.

Propositional logic

$m2 \leftrightarrow l1.$

Modular arithmetics

$m2 = r1.$

For the last statement we simply have to specify that the truth value of the two messages is the same.

Propositional logic

$m1 \leftrightarrow m2.$

Modular arithmetics

$m1 = m2.$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * (r2 + 1) + 1 & (2.4a) \\ m2 = r1 & (2.4b) \\ m1 = m2 & (2.4c) \end{cases}$$

By replacing $m1$ and $m2$ in 2.4c we obtain:

$$r1 * (r2 + 1) + 1 = r1$$

Which can be rewritten as:

$$r2 * r1 = 1$$

In order to satisfy the equation, both operands of the multiplication must be 1. Otherwise the product would be 0. Therefore the result will be:

$$\begin{cases} r1 = 1 \\ r2 = 1 \end{cases}$$

2.3.4 The Fourth Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** Both rooms contain ladies.
3. **Message on door 2:** Both rooms contain ladies.
4. If a lady is in the first room, then the message is true, but if a tiger is in it, then the message is false. For the second room, the rules are reversed, i.e. if a lady is in the second room, then the message is false otherwise the message is true.

The first and the second messages both tell us, that both rooms contain a lady.

Propositional logic

$m1 \leftrightarrow l1 \ \& \ l2.$

Modular arithmetics

$m1 = r1 * r2.$

Propositional logic

$m2 \leftrightarrow m1.$

Modular arithmetics

$m2 = m1.$

For the last statement we simply have to specify that, if a lady is in the first room, the first message is true, if a lady is in the second room, the second message is false.

Propositional logic

```
l1 -> m1. t1 -> -m1.
l2 -> -m2. t2 -> m2.
```

Modular arithmetics

```
r1 = m1.
r2 = m2 + 1.
```

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * r2 & (2.5a) \\ m2 = m1 & (2.5b) \\ r1 + m1 = 0 & (2.5c) \\ r2 + m2 = 1 & (2.5d) \end{cases}$$

By replacing $m1$ in 2.5c and $m2$ in 2.5d we obtain:

$$\begin{cases} r1 + r1 * r2 = 0 \\ r2 + r1 * r2 = 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r1 * (r2 + 1) = 0 \\ r2 * (r1 + 1) = 1 \end{cases}$$

In the second equation, in order to satisfy the it, both operands of the multiplication must be 1. Otherwise the product would be 0. Therefore the result will be:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

Which satisfies the first equation as well.

2.3.5 The Fifth Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** At least one of the rooms contains a lady.
3. **Message on door 2:** The other room contains a lady.
4. If a lady is in the first, then the message is true, but if a tiger is in it, then the message is false. For the second room, the rules are reversed, i.e. if a lady is in the room, then the message is false otherwise the message is true.

The first message tells us, that either only one room contains a lady, or both rooms contain ladies.

Propositional logic

$m1 \leftrightarrow l1 \mid l2.$

Modular arithmetics

$m1 = r1 * r2 + r1 + r2.$

The second message tells us, that in the other room (i.e.: the first room) there is a lady.

Propositional logic

$m2 \leftrightarrow l1.$

Modular arithmetics

$m2 = r1.$

For the last statement we simply have to specify that, if a lady is in the first room, the first message is true, if a lady is in the second room, the second message is false.

Propositional logic

$l1 \rightarrow m1. \quad t1 \rightarrow \neg m1.$
 $l2 \rightarrow \neg m2. \quad t2 \rightarrow m2.$

Modular arithmetics

$r1 = m1.$
 $r2 = m2 + 1.$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 * r2 + r1 + r2 & (2.6a) \\ m2 = r1 & (2.6b) \\ r1 + m1 = 0 & (2.6c) \\ r2 + m2 = 1 & (2.6d) \end{cases}$$

By replacing $m1$ in 2.6c and $m2$ in 2.6d we obtain:

$$\begin{cases} r1 + r1 * r2 + r1 + r2 = 0 \\ r2 + r1 = 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r2 * (r1 + 1) = 0 \\ r2 = r1 + 1 \end{cases}$$

Starting from the second equation, the two option would be $r1 = 0$ and $r2 = 1$ or $r1 = 1$ and $r2 = 0$. The first option does not satisfy the first equation, which means the result will be:

$$\begin{cases} r1 = 1 \\ r2 = 0 \end{cases}$$

2.3.6 The Sixth Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** It makes no difference which room you pick.

3. **Message on door 2:** There is a lady in the other room.

4. If a lady is in the first, then the message is true, but if a tiger is in it, then the message is false. For the second room, the rules are reversed, i.e. if a lady is in the room, then the message is false otherwise the message is true.

The first message tells us, that either both rooms contain tigers or both rooms contain ladies.

Propositional logic

$$m1 \leftrightarrow (l1 \ \& \ l2) \mid (t1 \ \& \ t2).$$

Modular arithmetics

$$m1 = r1 + r2 + 1.$$

The second message tells us, that in the other room (i.e.: the first room) there is a lady.

Propositional logic

$$m2 \leftrightarrow l1.$$

Modular arithmetics

$$m2 = r1.$$

For the last statement we simply have to specify that, if a lady is in the first room, the first message is true, if a lady is in the second room, the second message is false.

Propositional logic

$$\begin{aligned} l1 &\rightarrow m1. \quad t1 \rightarrow \neg m1. \\ l2 &\rightarrow \neg m2. \quad t2 \rightarrow m2. \end{aligned}$$

Modular arithmetics

$$\begin{aligned} r1 &= m1. \\ r2 &= m2 + 1. \end{aligned}$$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 + r2 + 1 & (2.7a) \\ m2 = r1 & (2.7b) \\ r1 = m1 & (2.7c) \\ r2 = m2 + 1 & (2.7d) \end{cases}$$

By replacing $m1$ in 2.7c and $m2$ in 2.7d we obtain:

$$\begin{cases} r1 = r1 + r2 + 1 \\ r2 = r1 + 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r2 = 1 \\ r1 = r2 + 1 \end{cases}$$

Which gives us the following result:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

2.3.7 The Seventh Trial

Knowledge base

1. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
2. **Message on door 1:** It makes a difference which room you pick.
3. **Message on door 2:** There is a lady in the other room.
4. If a lady is in the first, then the message is true, but if a tiger is in it, then the message is false. For the second room, the rules are reversed, i.e. if a lady is in the room, then the message is false otherwise the message is true.

The first message tells us, that one room contains a lady and the other a tiger.

Propositional logic

$$m1 \leftrightarrow (l1 \& t2) \mid (t1 \& l2).$$

Modular arithmetics

$$m1 = r1 + r2.$$

The second message tells us, that in the other room (i.e.: the first room) there is a lady.

Propositional logic

$$m2 \leftrightarrow l1.$$

Modular arithmetics

$$m2 = r1.$$

For the last statement we simply have to specify that, if a lady is in the first room, the first message is true, if a lady is in the second room, the second message is false.

Propositional logic

$$\begin{aligned} l1 &\rightarrow m1. \quad t1 \rightarrow \neg m1. \\ l2 &\rightarrow \neg m2. \quad t2 \rightarrow m2. \end{aligned}$$

Modular arithmetics

$$\begin{aligned} r1 &= m1. \\ r2 &= m2 + 1. \end{aligned}$$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 + r2 & (2.8a) \\ m2 = r1 & (2.8b) \\ r1 = m1 & (2.8c) \\ r2 = m2 + 1 & (2.8d) \end{cases}$$

By replacing $m1$ in 2.8c and $m2$ in 2.8d we obtain:

$$\begin{cases} r1 = r1 + r2 \\ r2 = r1 + 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r2 = 0 \\ r2 = r1 + 1 \end{cases}$$

Which gives us the following result:

$$\begin{cases} r1 = 1 \\ r2 = 0 \end{cases}$$

2.3.8 The Eighth Trial

Representation

For this puzzle, for simplicity we took four messages, which represent the following:

m_{ij} : Message i is on the door of room j and it is true

Knowledge base

1. In this puzzle we do not know which message corresponds to which door.
2. Each of the two rooms contained either a lady or a tiger, but it could be that there were tigers in both rooms, or ladies in both rooms.
3. **Message 1:** This room contains a tiger.
4. **Message on door 2:** Both rooms contain tigers.
5. If a lady is in the first, then the message is true, but if a tiger is in it, then the message is false. For the second room, the rules are reversed, i.e. if a lady is in the room, then the message is false otherwise the message is true.

The first message tells us, that the room contains a tiger. If the message is on the first door:

Propositional logic

$m_{11} \leftrightarrow t_1.$

Modular arithmetics

$m_{11} = r_1 + 1$

If the message is on the second door:

Propositional logic

$m_{12} \leftrightarrow t_2.$

Modular arithmetics

$m_{12} = r_2 + 1$

The second message tells us, that both rooms contain tigers. In this case, for the representation, it does not matter on which door the message is.

Propositional logic

$m_{21} \leftrightarrow t_1 \ \& \ t_2.$
 $m_{22} \leftrightarrow m_{21}.$

Modular arithmetics

$m_{21} = (r_1 + 1) * (r_2 + 1).$
 $m_{22} = m_{21}.$

For the last statement we simply have to specify that, if a lady is in the first room, the message on the first door is true, if a lady is in the second room, the message on the second door is false. For the tigers it is the other way around, if the tiger is in the first room, the message on that door is false. If the tiger is in the second room, the message on that door is true.

Propositional logic

$l_1 \rightarrow m_{11} \mid m_{21}. \ t_1 \rightarrow \neg m_{11} \mid \neg m_{21}.$
 $l_2 \rightarrow \neg m_{12} \mid \neg m_{22}. \ t_2 \rightarrow m_{12} \mid m_{22}.$

Modular arithmetics

$(r_2 + 1) * (m_{12} * m_{22} + m_{12} + m_{22}) +$
 $r_2 = 1.$
 $r_2 * ((m_{12} + 1) * (m_{22} + 1) + m_{12} +$

$$m22) + r2 + 1 = 1.$$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m11 = r1 + 1 & (2.9a) \\ m12 = r2 + 1 & (2.9b) \\ m21 = (r1 + 1) * (r2 + 1) & (2.9c) \\ m22 = m21 & (2.9d) \\ r1 * (m11 * m21 + m11 + m21) + r1 + 1 = 1 & (2.9e) \\ (r1 + 1) * ((m11 + 1) * (m21 + 1) + m11 + m21) + r1 = 1 & (2.9f) \\ (r2 + 1) * (m12 * m22 + m12 + m22) + r2 = 1 & (2.9g) \\ r2 * ((m12 + 1) * (m22 + 1) + m12 + m22) + r2 + 1 = 1 & (2.9h) \end{cases}$$

By replacing $m11$, $m12$, $m21$ and $m22$ in 2.9e, 2.9f, 2.9g, 2.9h we obtain:

$$\begin{cases} r1 * ((r1 + 1) * ((r1 + 1) * (r2 + 1)) + r1 + 1 + (r1 + 1) * (r2 + 1)) + r1 + 1 = 1 \\ (r1 + 1) * (r1 * ((r1 + 1) * (r2 + 1) + 1) + r1 + (r1 + 1) * (r2 + 1)) + r1 = 1 \\ (r2 + 1) * ((r2 + 1) * ((r1 + 1) * (r2 + 1)) + r2 + 1 + (r1 + 1) * (r2 + 1)) + r2 = 1 \\ r2 * (r2 * ((r1 + 1) * (r2 + 1) + 1) + r2 + 1 + (r1 + 1) * (r2 + 1)) + r2 + 1 = 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r1 * (r1 * r2 + r1 + r1 * r2 + r1 + r1 * r2 + r1 + r2 + 1 + r1 + 1 + r1 * r2 + r1 + r2 + 1) + r1 + 1 = 1 \\ (r1 + 1) * (r1 * r2 + r1 + r1 * r2 + r1 + r1 * r2 + r1 + r2) + r1 = 1 \\ (r2 + 1) * (r1 * r2 + r1 * r2 + r2 + r2 + r1 * r2 + r1 + r2 + 1 + r2 + 1 + r1 * r2 + r1 + r2 + 1) + r2 = 1 \end{cases}$$

Knowing that $x + x = 0$ and $x * x = x$, we get the following equations:

$$\begin{cases} r1 + 1 = 1 \\ r1 * r2 + r1 + r2 = 1 \end{cases}$$

We get the following result:

$$\begin{cases} r1 = 0 \\ r2 = 1 \end{cases}$$

2.3.9 The Ninth Trial

Knowledge base

1. Now we have three rooms: one contains a lady, and the other two contain a tiger
2. **Message on door 1:** A tiger is in this room.
3. **Message on door 2:** A lady is in this room.
4. **Message on door 3:** A tiger is in room 2.
5. At most one of the three signs is true.

The first statement tells us, that one room contains a lady and the other two contain tigers.

Propositional logic

$11 \ \& \ t2 \ \& \ t3 \mid 12 \ \& \ t1 \ \& \ t3 \mid 13 \ \& \ t1 \ \& \ t2.$

Modular arithmetics

$r1 \ * \ r2 \ * \ r3 \ = \ 0.$
 $r1 \ + \ r2 \ + \ r3 \ = \ 1.$

The second statement (first message) tells us, that the room contains a tiger.

Propositional logic

$m1 \ \<-> \ t1.$

Modular arithmetic

$m1 \ = \ r1 \ + \ 1.$

The third statement (second message) tells us, that the room contains a lady.

Propositional logic

$m2 \ \<-> \ 12.$

Modular arithmetic

$m2 \ = \ r2.$

The fourth statement (third message) tells us, that the second room contains a tiger.

Propositional logic

$m3 \ \<-> \ t2.$

Modular arithmetic

$m3 \ = \ r2 \ + \ 1.$

For the last statement we simply have to specify that, at most one message is true.

Propositional logic

$m1 \ \rightarrow \ \neg m2 \ \& \ \neg m3.$
 $m2 \ \rightarrow \ \neg m3.$

Modular arithmetical

$m1 \ * \ m2 \ + \ m1 \ * \ m3 \ + \ m2 \ * \ m3 \ + \ 1 \ = \ 1.$

Resolution

The system of equations representing the knowledge base:

$$\begin{cases} m1 = r1 + 1 & (2.10a) \\ m2 = r2 & (2.10b) \\ m3 = r2 + 1 & (2.10c) \\ m1 * m2 + m1 * m3 + m2 * m3 + 1 = 1 & (2.10d) \\ r1 * r2 * r3 = 0 & (2.10e) \\ r1 + r2 + r3 = 1 & (2.10f) \end{cases}$$

By replacing $m1$, $m2$ and $m3$ in 2.10d we obtain:

$$\begin{cases} (r1 + 1) * r2 + (r1 + 1) * (r2 + 1) + r2 * (r2 + 1) = 1 \\ r1 * r2 * r3 = 0 \\ r1 + r2 + r3 = 1 \end{cases}$$

Which can be rewritten as:

$$\begin{cases} r1 * r2 + r2 + r1 * r2 + r1 + r2 + 1 + r2 * r2 + r2 + 1 = 1 \\ r1 * r2 * r3 = 0 \\ r1 + r2 + r3 = 1 \end{cases}$$

Knowing that $x + x = 0$ and $x * x = x$, we get:

$$\begin{cases} r1 = 1 \\ r2 * r3 = 0 \\ r2 + r3 = 0 \end{cases}$$

Which gives us the following result:

$$\begin{cases} r1 = 1 \\ r2 = 0 \\ r3 = 0 \end{cases}$$

2.3.10 The Tenth Trial

Knowledge base

1. Now we have three rooms: one contains a lady, and the other two contain a tiger
2. **Message on door 1:** A tiger is in this room.
3. **Message on door 2:** A lady is in this room.
4. **Message on door 3:** A tiger is in room 2.
5. The message on the room containing the lady is true, and from the other two, at least one is false.

The first statement tells us, that one room contains a lady and the other two contain tigers.

Propositional logic

```
l1 & t2 & t3 | l2 & t1 & t3 | l3 &
t1 & t2.
```

Modular arithmetics

```
r1 * r2 * r3 = 0.
r1 + r2 + r3 = 1.
```

The second statement (first message) tells us, that a tiger is in room 2.

Propositional logic

```
m1 <-> t2.
```

Modular arithmetic

```
m1 = r2 + 1.
```

The third statement (second message) tells us, that the room contains a tiger.

Propositional logic

```
m2 <-> t2.
```

Modular arithmetic

```
m2 = r2 + 1.
```

The fourth statement (third message) tells us, that the first room contains a tiger.

Propositional logic

```
m3 <-> t1.
```

Modular arithmetic

```
m3 = r1 + 1.
```

For the last statement we simply have to specify that, the message on the room containing the lady is always true, and from the other two messages at least one is false.

Propositional logic

```

l1 -> m1.
l2 -> m2.
l3 -> m3.
m1 -> -m2 | -m3.
m2 -> -m1 | -m3.
m3 -> -m1 | -m2.

```

Modular arithmetical

```

r1 * (m1 + 1) = 0.
r2 * (m2 + 1) = 0.
r3 * (m3 + 1) = 0.
m1 * m2 * m3 = 0.
(m1 + 1) * (m2 + 1) * (m3 + 1) = 0.

```

Resolution

The system of equations representing the knowledge base:

$$\begin{cases}
 m1 = r2 + 1 & (2.11a) \\
 m2 = r2 + 1 & (2.11b) \\
 m3 = r1 + 1 & (2.11c) \\
 r1 * (m1 + 1) = 0 & (2.11d) \\
 r2 * (m2 + 1) = 0 & (2.11e) \\
 r3 * (m3 + 1) = 0 & (2.11f) \\
 m1 * m2 * m3 = 0 & (2.11g) \\
 (m1 + 1) * (m2 + 1) * (m3 + 1) = 0 & (2.11h) \\
 r1 * r2 * r3 = 0 & (2.11i) \\
 r1 + r2 + r3 = 1 & (2.11j)
 \end{cases}$$

By replacing $m1$ and $m2$ in 2.11d, 2.11e, 2.11f, 2.11g and 2.11h we obtain:

$$\begin{cases}
 r1 * r2 = 0 \\
 r2 = 0 \\
 r3 * r1 = 0 \\
 r1 * r2 + r1 + r2 + 1 = 0 \\
 r1 * r2 = 0 \\
 r1 * r2 * r3 = 0 \\
 r1 + r2 + r3 = 1
 \end{cases}$$

Which can be rewritten as:

$$\begin{cases}
 r3 * r1 = 0 \\
 r1 + 1 = 0 \\
 r1 + r3 = 1
 \end{cases}$$

Which gives us the following result:

$$\begin{cases}
 r1 = 1 \\
 r2 = 0 \\
 r3 = 0
 \end{cases}$$

2.3.11 Representation for empty rooms

From now on, besides lady and tiger we will also have empty rooms. The notations will be similar:

in **propositional logic** we will represent the state of the i -th room in the following way:

- li : There is a lady in room i .
- ti : There is a tiger in room i .
- ei : Room i is empty.
- mi : Message on the door of room i

Then we must specify that if a lady is in a room, then there can not be a tiger and neither can it be empty. Same rules apply if a tiger is in the room, there can not be a lady in that room and that room can not be empty:

```
l1 -> -t1 & -e1 .
l2 -> -t2 & -e2 .
l3 -> -t3 & -e3 .
t1 -> -e1 .
t2 -> -e2 .
t3 -> -e3 .
```

For **modular arithmetic** we define:

- $li = [["\text{There is a lady in room } i"]]$
- $ti = [["\text{There is a tiger in room } i"]]$
- $ei = [["\text{Room } i \text{ is empty}"]]$
- $mi = [["\text{Message on the door of room } i"]]$

This representation will no longer exclude the possibility of a tiger and lady in the same room, so we have to specify it explicitly.

2.3.12 First, Second, and Third Choice

Knowledge base

1. Now we have three rooms: one contains a lady, another one contains a tiger, and the third is empty
2. **Message on door 1:** Room three is empty.
3. **Message on door 2:** The tiger is in room one.
4. **Message on door 3:** This room is empty.
5. The message on door of the room where the lady is is true, the message where the tiger is is false, and the message on the room that is empty can be either true or false.

The first statement tells us, that only one room contains a lady, one room contains a tiger and one room is empty.

Propositional logic

```
l1 | l2 | l3.
l1 -> -l2 & -l3.
l2 -> -l1 & -l3.
l3 -> -l1 & -l2.
```

```
t1 | t2 | t1.
t1 -> -t2 & -t3.
t2 -> -t1 & -t3.
t3 -> -t2 & -t1.
```

```
e1 | e2 | e3.
e1 -> -e2 & -e3.
e2 -> -e1 & -e3.
e3 -> -e2 & -e1.
```

Modular arithmetics

```
l1 * l2 * l3 = 0.
l1 + l2 + l3 = 1.
```

```
t1 * t2 * t3 = 0.
t1 + t2 + t3 = 1.
```

```
e1 * e2 * e3 = 0.
e1 + e2 + e3 = 1.
```

```
l1 * ((t1 + 1) * (e1 + 1)) + l1 + 1
= 1.
l2 * ((t2 + 1) * (e2 + 1)) + l2 + 1
= 1.
l3 * ((t3 + 1) * (e3 + 1)) + l3 + 1
= 1.
```

```
t1 * ((e1 + 1) * (l1 + 1)) + t1 + 1
= 1.
t2 * ((e2 + 1) * (l2 + 1)) + t2 + 1
= 1.
t3 * ((e3 + 1) * (l3 + 1)) + t3 + 1
= 1.
```

The second statement tells us, that room three is empty.

Propositional logic

```
m1 <-> e3.
```

Modular arithmetics

```
m1 = e3.
```

The third statement tells us, that room one contains a tiger.

Propositional logic

```
m2 <-> t1.
```

Modular arithmetics

```
m2 = t1.
```

The fourth statement tells us, that the room is empty.

Propositional logic

```
m3 <-> e3.
```

Modular arithmetics

```
m3 = e3.
```

For the last statement we simply have to specify that, if a lady is the room, the message is true, if a tiger is in the room, the message is false For empty rooms, we do not have to write any conditions, because they can be either true or false.

Propositional logic

```
l1 -> m1.
l2 -> m2.
l3 -> m3.
```

```
t1 -> -m1.
t2 -> -m2.
t3 -> -m3.
```

Modular arithmetic

```
l1 * m1 + l1 + 1 = 1.
l2 * m2 + l2 + 1 = 1.
l3 * m3 + l3 + 1 = 1.
```

```
t1 * (m1 + 1) + t1 + 1 = 1.
t2 * (m2 + 1) + t2 + 1 = 1.
t3 * (m3 + 1) + t3 + 1 = 1.
```

Resolution

The system of equations representing the knowledge base:

$$\begin{cases}
 l1 * l2 * l3 = 0 & (2.12a) \\
 l1 + l2 + l3 = 1 & (2.12b) \\
 t1 * t2 * t3 = 0 & (2.12c) \\
 t1 + t2 + t3 = 1 & (2.12d) \\
 e1 * e2 * e3 = 0 & (2.12e) \\
 e1 + e2 + e3 = 1 & (2.12f) \\
 l1 * ((t1 + 1) * (e1 + 1)) + l1 + 1 = 1 & (2.12g) \\
 l2 * ((t2 + 1) * (e2 + 1)) + l2 + 1 = 1 & (2.12h) \\
 l3 * ((t3 + 1) * (e3 + 1)) + l3 + 1 = 1 & (2.12i) \\
 t1 * ((e1 + 1) * (l1 + 1)) + t1 + 1 = 1 & (2.12j) \\
 t2 * ((e2 + 1) * (l2 + 1)) + t2 + 1 = 1 & (2.12k) \\
 t3 * ((e3 + 1) * (l3 + 1)) + t3 + 1 = 1 & (2.12l) \\
 m1 = e3 & (2.12m) \\
 m2 = t1 & (2.12n) \\
 m3 = e3 & (2.12o) \\
 l1 * m1 + l1 + 1 = 1 & (2.12p) \\
 l2 * m2 + l2 + 1 = 1 & (2.12q) \\
 l3 * m3 + l3 + 1 = 1 & (2.12r) \\
 t1 * (m1 + 1) + t1 + 1 = 1 & (2.12s) \\
 t2 * (m2 + 1) + t2 + 1 = 1 & (2.12t) \\
 t3 * (m3 + 1) + t3 + 1 = 1 & (2.12u)
 \end{cases}$$

By replacing $m1$, $m2$ and $m3$ in 2.12p, 2.12q, 2.12r, 2.12s, 2.12t, 2.12u, we obtain:

$$\left\{ \begin{array}{l} l1 * l2 * l3 = 0. \\ l1 + l2 + l3 = 1. \\ t1 * t2 * t3 = 0. \\ t1 + t2 + t3 = 1. \\ e1 * e2 * e3 = 0. \\ e1 + e2 + e3 = 1. \\ l1 * ((t1 + 1) * (e1 + 1)) + l1 + 1 = 1. \\ l2 * ((t2 + 1) * (e2 + 1)) + l2 + 1 = 1. \\ l3 * ((t3 + 1) * (e3 + 1)) + l3 + 1 = 1. \\ t1 * ((e1 + 1) * (l1 + 1)) + t1 + 1 = 1. \\ t2 * ((e2 + 1) * (l2 + 1)) + t2 + 1 = 1. \\ t3 * ((e3 + 1) * (l3 + 1)) + t3 + 1 = 1. \\ l1 * e3 + l1 + 1 = 1. \\ l2 * t1 + l2 + 1 = 1. \\ l3 * e3 + l3 + 1 = 1. \\ t1 * (e3 + 1) + t1 + 1 = 1. \\ t2 * (t1 + 1) + t2 + 1 = 1. \\ t3 * (e3 + 1) + t3 + 1 = 1. \end{array} \right.$$

Which, after the right calculations, gives us the following result:

$$\left\{ \begin{array}{l} l1 = 1 \\ l2 = 0 \\ l3 = 0 \\ t1 = 0 \\ t2 = 1 \\ t3 = 0 \\ e1 = 0 \\ e2 = 0 \\ e3 = 1 \end{array} \right.$$

2.4 Solving logic puzzles programatically

The advantage of translating propositions into a system of equations in the ring form is that it can be solved by using only algebra. Thus a range of mathematical tools can be used for finding a solution, having the following characteristics:

- Can do symbolic computation
- Can solve multi-variable equations
- Can work with values from the Boolean ring
 - Can simplify x^2 to x and $x + x$ to 0
 - Can solve equations with variables from this domain

Such tool could be Mathematica. Mace4 is also a good tool for solving this kind of equations, however it has a flaw, presented in 2.3, which requires in some cases to wrap the expressions in “mod 2”. We used Mace4 for verifying the correctness of our solutions.

Practically if we multiply all the expressions together, according to the equation 2.1, we could have a single multi-variable equation, which can be solved by these tools, thus solving the logic puzzle via algebra.

2.5 Translating propositional logic to modular arithmetic programatically

Our purpose was not to write a solver for the equations in the ring form. Instead, a translator program was written in Python which converts Mace4 input files in the propositional logic form to the modular arithmetic equivalent.

For this purpose, the **SymPy** library was used which provides symbolic computation capabilities in Python. However it has some limitations:

- Does not simplify x^2 to x automatically. Solved by modifying the expression tree.
- We could not find any possibility for achieving all the capabilities from 2.4 at the same time, i.e. solving multi-variable equation on the Boolean ring.

The source code of the program can be found in annex C.

2.5.1 Data representation

Input data representation

A Mace4 input file contains a set of commands, formulas, clauses and lists. Currently the program supports only those input files that have only commands and formulas.

In `commands.py` two classes were defined to represent the commands and formula lists of the input file. These classes were created only for holding data. The correspondence between the input files and the classes can be seen below:

Mace4 command

```
1 assign(domain_size, 2).
```

Command object creation

```
1 Command(  
2     name = "assign",  
3     args = ["domain_size", "2"])
```

Mace4 formulas

```
1 formulas(assumptions).  
2     l1 -> -t1.  
3     l2 -> -t2.  
4 end_of_list.
```

Formulas object creation

```
1 Formulas(  
2     name = "assumptions",  
3     formula_list = [  
4         "l1 -> -t1",  
5         "l2 -> -t2"  
6     ])
```

Additionally, the Formulas class hold two class variables which represent its starting and ending keywords.

```
class Formulas:
    ...
    commandName = "formulas"
    endCommand = "end_of_list"
    ...
```

Expression tree representation

In order to represent expressions, which are expression trees, we used the already implemented classes of **SymPy**. The **Expr** class is the base class for algebraic expressions. The **Mul** and **Add** classes inherit from this class and represent the multiplication and addition respectively. The operands of these operators can be a number a symbol or another expression.

The **Symbol** class represents a symbol, which is considered as an atomic expression, thus its also a descendant of **Expr** and can be used as an operand for other expressions.

Each expression tree can be considered as a standalone expression. Thus the composite pattern was applied in order to create expression trees.

2.5.2 Parsing the input file

`input.py` contains the functions used for parsing the input file.

The `processLines(lines)` function takes a list of lines (strings), processes them, and returns another list of lines as a result. The comments, whitespaces and empty lines are removed. Also, the lines of a Mace4 input file are logically separated by a “.” (dot). Therefore the lines are merged and split again, in order to change the separator to the dot.

The `linesToCommandsAndFormulas(lines)` function takes a list of lines (processed by the previous function) and transforms the data into a list of **Command** and **Formulas** objects. The correspondence between the lines and the classes was presented in 2.5.1. A command takes a single line, and its data can be obtained by splitting the line by “(”, “)” and “,”. A formulas section takes multiple lines. Thus the lines between the starting and ending keywords are read before creating the **Formulas** object.

The parsing method presented above is not the most efficient solution, however it was implemented easier, thanks to the string and list manipulation functions in Python.

2.5.3 Defining the rules

The first part of `expression.py` contains the definitions regarding the rules of the propositional logic in Mace4, and the rules of conversion to the ring form.

Operator precedence

Based on [2], the precedence of propositional logic operators was constructed. It can be observed in table 2.2. Additionally, the operators of the same type or the same priority are right associative. Therefore by constructing the expression tree from left to right, the associativity rules are satisfied.

Symbols

In order to make difference between operators and operands while verifying the lines character-by-character, the function `is_symbol_char(char)` was created which determines whether that character can be part of a symbol (operand) or not.

Operator	String form	Priority
Left parenthesis	(0
Right parenthesis)	0
Implication (\rightarrow)	\rightarrow	1
Biconditional (\leftrightarrow)	\leftrightarrow	1
Inclusive disjunction (\vee)	\vee	2
Conjunction (\wedge)	\wedge	3
Negation (\neg)	\neg	4

Table 2.2: The precedence of operators in propositional logic. The higher the priority number, the earlier is the operator evaluated.

It was assumed that a symbol is composed by english letters and digits. It was further assumed, that no operator contain characters that can be part of a symbol.

Conversion to ring form

The conversion of propositions to expression trees of the ring form was based on table 2.1. This table was implemented in the `operatorStringToExpression(operator, *operands)` function, which takes as parameter a string, representing an operator, and a list of operands. Using the `Mul` and `Add` classes of **SymPy** it returns an expression in which the operands are applied to the corresponding operators, according to the table.

2.5.4 Creating expression trees from propositions

The next part of `expression.py` contains the functionalities for transforming propositions into SymPy expression trees.

The `splitProposition(proposition)` function takes as argument a proposition as a string. It separates all the terms (operators + operands) of the string and returns them as a list of strings. The separation is done by verifying each character if it could be part of a symbol or not. If between two consecutive characters this verification returns a different result, they belong to different terms. Moreover if space or parenthesis is occurred, it is sure that the previous term terminated.

The `buildExpressionTree(proposition, substitutions=None)` function handles the main logic for the expression tree building. It obtains as argument a string proposition and extracts its terms using the previous function. Then each term is verified from left to right.

- In case a left parenthesis occurred, it is pushed to the term stack.
- In case the term is an operand, a `Symbol` is created from it, and pushed to the node stack.
- In case an operator is occurred, the term stack is popped until the TOS has a lower priority than the current operator. A new node is formed from the popped operator and the corresponding one (unary) or two (binary) operands from the node stack. The `operatorStringToExpression` is used for the creation of the nodes. Finally the current term is pushed to the term stack.
- In case of a right parenthesis, the node creation occurs until a left parenthesis is reached in the term stack.
- After each term is considered, additional node creations are done until the term stack becomes empty.

- Finally the node stack should contain only one element, the resulting expression tree.

The `substitutions` parameter is a dictionary which defines which symbols should be substituted with which expression. If the parameter is given, whenever a term appears in the dictionary, the corresponding expression will be pushed to the node stack.

The expression tree was created directly in the algebraic form. Another possibility would be to create a propositional logic expression tree, then convert the tree to the algebraic form, using the `operatorStringToExpression` function.

The resulting expression tree is in the algebraic form, the operators were translated just as they were defined in the table. However the result is unreadable and can be further simplified. Also, the automated simplification that occurred during the node creations followed only the rules of general algebra, therefore coefficients could appear with values greater than one, and also powers could appear.

The `simplifyExpression(expr, symbols)` function takes as argument an expression and the list of symbols appearing in it, and simplifies it according to the rules of the Boolean ring. A SymPy polynomial is created from the expression with the modulus parameter set to 2. This will result in the automated simplification of the expression, based on the parity of the coefficients.

However the powers still remain. We know that $x^2 = x$. Therefore x on any power will result in x . Thus, to further simplify the equation, it was needed to manually replace a power operator (Pow class) with its first argument, the base. The `recurse_replace(expr, func)` and the `rewrite(expr, new_args)` functions do the DFS traversal and the replacing. Finally the modulus is set again to 2, in order to trigger further simplifications. The resulting expression is extracted from the polynomial.

Listing 2.1: Simplification on the Boolean ring using SymPy

```
def simplifyExpression(expr, symbols):
    ...
    p = Poly(expr, symbols, modulus=2)
    p = recurse_replace(p, rewrite).set_modulus(2)

    return p.as_expr()
```

2.5.5 Creating and transformation of expressions

The last part of `expression.py` contains functions for further transforming the expressions, and also a wrapper function which puts together all the functionalities of this file.

The `formulasToExpressionTree(formulas, merge, simplify, collect, substitutions)` function takes as parameter an object of type `Formulas` and changes its `formula_list` field from a list of strings to a SymPy expression (tree). Returns the resulted object.

The `simplify` flag indicates whether or not to simplify the resulted expressions using the `simplifyExpression` function described above

If `collect` flag is set, the terms of the expressions are grouped together, based on the frequency of the symbols. An expression is first grouped by the most frequent term, then by the second most frequent, and so on. This heuristic was used in order to minimize the length of the resulting expression. The `getSymbolsByDecreasingFrequency(expression)` returns the symbols of an expression in decreasing order of frequency, and the `collectExpression(expression)` function does the collecting using the previous function and the `sympy.collect` function.

If the `merge` flag is set, all the expressions of a `Formulas`, is multiplied together to form a single expression according to the equation 2.1.

The `substitutions` argument is the same as discussed before, and is forwarded directly to the `buildExpressionTree` function.

In order to create the substitutions dictionary, the `createSubstitutions(predicates)` function can be used which takes a list of strings, each string having the form $a \leftrightarrow f(b)$, where a is a symbol, and $f(b)$ is a predicate. The resulting dictionary is created by associating to each a symbol (string) an expression, obtained by building an algebraic expression tree from $f(b)$ using the `buildExpressionTree` function.

2.5.6 Printing the result

The `output.py` file contains the functions which prints the `Command` and `Formulas` classes in a format that can be given as input for Mace4.

The commands that have as first argument “arithmetic” or “domain_size” are ignored, because the program sets these values automatically in order to ensure a correct input for Mace4.

Listing 2.2: The commands that are automatically inserted by the program

```
assign(domain_size, 2).
assign(max_models, -1).
```

2.5.7 The main program

The `translator.py` file contains the main function, which reads the command line arguments and executes the operations using the previously defined functions, finally printing the result.

For the list of substitutions the program consider only those predicates that are placed inside a `formulas(substitutions).list`. These predicates are removed from the output.

2.6 Usage of the translator program

By calling the program with the `-h` or `--help` parameters the following list of help message is shown. It describes all the command line arguments used by the program.

Listing 2.3: Help message of the program

```
$ python translator.py -h

Usage: translator.py [options] [-f <inputfile>]

A converter from propositional logic to modular arithmetic in the mace4
format

Options:
  -c, --collect          Collects the terms of each expression by the
                        decreasing frequency of their symbols.
  -h, --help             Prints help message
  -f, --file string      Path to the input file. If the option is missing,
                        the program reads from STDIN
  -m, --merge            Merges all the expressions of a "formulas" section
                        into a single expression.
  --mod                  Wraps all expressions in "mod 2"
  -s, --subs             Substitutes symbols with their corresponding
                        expression. The substitutions must be given in the "a <-> f(b)." format
                        in "formulas(substitutions)."
```

2.6.1 Input and output

If the `-f` or `--file` parameter is not given, the program reads the input file from the STDIN. This can be used for supplying a piped input. If the flag is given, the program opens the file and reads the lines from it.

Listing 2.4: The possible ways of supplying the input to the program

```
$ python translator.py -f input.txt
$ python translator.py --file=input.txt
$ cat input.txt | python translator.py
$ python translator.py < input.txt
```

The output of the program is written to the STDOUT. Therefore it can be used both for writing into a file and for piping into another program.

Listing 2.5: The possible ways of using the output of the program

```
$ cat input.txt | python translator.py --mod | mace4 | interpformat
$ python translator.py < input.txt > output.txt
```

2.6.2 Refining the output

There are various options to transform the output of the program.

No refining

Consider the following input file:

Listing 2.6: Sample input file

```
1 assign(domain_size, 2).
2 assign(max_models, -1).
3
4 % l1: There is a lady in room 1
5 % l2: There is a lady in room 2
6 % t1: There is a tiger in room 1
7 % t2: There is a tiger in room 2
8 % m1: Message on the door of room 1
9 % m2: Message on the door of room 2
10
11 formulas(assumptions).
12
13     % Each of the two rooms contained either a lady or a tiger,
14     % but it could be that there were tigers in both rooms, or ladies in
15     % both rooms.
16     % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant
17
18     % No tiger in the room where the lady stays
19     l1 -> -t1.
20     l2 -> -t2.
21
22     % Message on door #1: In this room there is a lady, and in the other
23     % room there is a tiger.
24     m1 <-> l1 & t2.
25
26     % Message on door #2: In one of these rooms there is a lady, and in one
27     % of these rooms there is a tiger.
28     m2 <-> (l1 | l2) & (t1 | t2).
```

```

27 % One of the messages is true, but the other one is false.
28 m1 <-> -m2.
29 end_of_list.

```

The input file consists of a list of five formulas, two command, a several comments. If the program is called without setting any flags, the following output would be observed:

Listing 2.7: Program output for the sample file

```

1 $ python translator.py -f input.txt
2 assign(max_models, -1).
3 set(arithmetic).
4 assign(domain_size, 2).
5
6 formulas(assumptions).
7     l1*t1 + 1 = 1.
8     l2*t2 + 1 = 1.
9     l1*t2 + m1 + 1 = 1.
10    l1*l2*t1*t2 + l1*l2*t1 + l1*l2*t2 + l1*t1*t2 + l1*t1 + l1*t2 + l2*t1
    *t2 + l2*t1 + l2*t2 + m2 + 1 = 1.
11    m1 + m2 = 1.
12 end_of_list.

```

One can observe, that the all the commands that not contain “arithmetic” or “domain_size” as first arguments, are directly copied to the output. Then these two values is automatically set by the program: to use arithmetic resolution on a domain size of 2. This is done in order to ensure that the output is a valid mace4 input.

Substitution

One can also observe that in the previous output the formulas are at the same place, but they are converted to the ring form and simplified. However, each predicate is translated and simplified separately. Therefore, in this case, the program does not know that it could substitute $t1$ with $(l1 + 1)$, thus further simplifying the equations.

To specify substitutions, one must write formulas of form $a \leftrightarrow f(b)$, where a is the symbol to be substituted and $f(b)$ will be the expression that substitutes it. These substitution rules must be written in the body of `formulas(substitutions)`. . Below can be seen the same input file with added substitutions.

Listing 2.8: Sample input file with substitutions

```

1 %set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % l1: There is a lady in room 1
6 % l2: There is a lady in room 2
7 % t1: There is a tiger in room 1
8 % t2: There is a tiger in room 2
9 % m1: Message on the door of room 1
10 % m2: Message on the door of room 2
11
12 formulas(substitutions).
13     t1 <-> -l1.
14     t2 <-> -l2.
15 end_of_list.
16
17 formulas(assumptions).
18

```

```

19 % Each of the two rooms contained either a lady or a tiger,
20 % but it could be that there were tigers in both rooms, or ladies in
    both rooms.
21 % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant
22
23 % No tiger in the room where the lady stays
24 l1 -> -t1.
25 l2 -> -t2.
26
27 % Message on door #1: In this room there is a lady, and in the other
    room there is a tiger.
28 m1 <-> l1 & t2.
29
30 % Message on door #2: In one of these rooms there is a lady, and in one
    of these rooms there is a tiger.
31 m2 <-> (l1 | l2) & (t1 | t2).
32
33 % One of the messages is true, but the other one is false.
34 m1 <-> -m2.
35 end_of_list.

```

Calling the program with the `-s` or `--subs` flags, it will give the following output:

Listing 2.9: Program output for the sample file with substitutions

```

1 $ python translator.py -s -f input.txt
2 assign(max_models, -1).
3 set(arithmetic).
4 assign(domain_size, 2).
5
6 formulas(assumptions).
7     1 = 1.
8     1 = 1.
9     l1*l2 + l1 + m1 + 1 = 1.
10    l1 + l2 + m2 + 1 = 1.
11    m1 + m2 = 1.
12 end_of_list.

```

One can observe the following:

- The substitution formulas were removed from the output.
- Because of the substitutions, the first two formulas became tautologies. However they were not removed from the output, in order to preserve the number and the order of the formulas.
- The result became easily readable, close to the one that we wrote by hand in B.1.2.

Collecting (grouping) terms

The lengths of the expressions can further be reduced by grouping the terms according to the distributivity rule of the Boolean ring. The terms are grouped by the decreasing order of the apparitions of the symbols.

Calling the program with the `-c` or `--collect` flags for the same input, it will give the following output:

Listing 2.10: Program output for the sample file with substitutions and collecting

```

1 $ python translator.py -s -c -f input.txt
2 assign(max_models, -1).

```

```

3 set(arithmetic).
4 assign(domain_size, 2).
5
6 formulas(assumptions).
7     1 = 1.
8     1 = 1.
9     l1*(l2 + 1) + m1 + 1 = 1.
10    l1 + l2 + m2 + 1 = 1.
11    m1 + m2 = 1.
12 end_of_list.

```

Besides the third formula, no expression contains a symbol more than once. In the third formula `m1` appears once, `l2` appears once, and `l1` appears twice. The list containing the symbols in the decreasing order of frequency will be `[r1, r2, m1]`. Therefore the terms of the expression are grouped by this order which results in the same output that was printed to the console.

Merging expressions

According to the equation 2.1, by multiplying each expression, we obtain a single multi-variable equation whose solutions will be the solutions of the logic puzzle.

Calling the program with the `-m` or `--merge` flags for the same input, it will multiply all equations of a “formulas” list into a single one, and will give the following output:

Listing 2.11: Program output for the sample file with substitutions, collecting and merging

```

1 $ python translator.py -s -c -m -f input.txt
2 assign(max_models, -1).
3 set(arithmetic).
4 assign(domain_size, 2).
5
6 formulas(assumptions).
7     m2*(l1*l2*m1 + l1*l2 + l2*m1 + l2) = 1.
8 end_of_list.

```

Providing that single equation to any mathematical tool that satisfies the capabilities mentioned in 2.4 will give the solutions for each symbol, therefore solving the logic puzzle. Thus this program exemplifies the idea mentioned in [3]: “Solving Knights-and-Knaves with One Equation”

Wrapping equations in “mod 2”

Because of the flaw of Mace4 mentioned in 2.3, some expressions must be wrapped in “mod 2” in order to ensure that mace4 can solve it.

Therefore the program provides the `--mod` flag. When it is set, all the expressions are wrapped in “mod2”. Calling the sample program with this flag for the same input will result in the following output:

Listing 2.12: Program output for the sample input with substitutions, collecting, merging and wrapping in mod 2

```

1 $ python translator.py -s -c -m -f input.txt
2 assign(max_models, -1).
3 set(arithmetic).
4 assign(domain_size, 2).
5
6 formulas(assumptions).
7     m2*(l1*l2*m1 + l1*l2 + l2*m1 + l2) = 1.

```

```
8 end_of_list.
```

Listing 2.13: The output of mace4 for the previous program output given as input

```
1 $python translator.py -s -c -m --mod -f input.txt | mace4 | interpformat
2
3 === Mace4 starting on domain size 2. ===
4
5 ----- process 55484 exit (all_models) -----
6 interpretation( 2, [number = 1,seconds = 0], [
7     function(l1, [0]),
8     function(l2, [1]),
9     function(m1, [0]),
10    function(m2, [1])]).
```


Bibliography

- [1] Mace4 (Models And CounterExamples). In Prover9 Manual 2009-11A. <https://www.cs.unm.edu/~mccune/prover9/manual/2009-11A/>. Accessed: 2020-11-30.
- [2] Operator Precedence. In Introduction to Logic. http://intrologic.stanford.edu/glossary/operator_precedence.html. Accessed: 2020-11-30.
- [3] Francesco Ciraulo and Samuele Maschio. Solving knights-and-knaves with one equation. volume 52, pages 82–89. 2020.
- [4] Stuart Russell and Peter Norvig. Solving problems by searching. In *Artificial Intelligence: A Modern Approach*, chapter 3, pages 63–105. Pearson, 4 edition, 2020.
- [5] Raymond M. Smullyan. Lady or the tiger? and other logic puzzles including a mathematical novel that features godel’s great discovery. Random House Puzzles & Games, 1992.

Appendix A

Source Code: Search

The code snippets below are from `search.py`.

A.1 Nodes

To represent the search nodes, we used different strategies across the various search algorithms. In some cases we used tuples containing the state, the action and the cumulative cost of a node, while in other cases we used the `Node` class defined below.

In the latter case, we stored in a node a reference to its parent. This information is used for reconstructing the path from the start node to the goal node, in the `reconstructPath(node)` method.

Listing A.1: The Node class

```
67 class Node:
68     """
69     Node used for search algorithms.
70     """
71
72     def __init__(self, state, action, cost, parent, f=0):
73         """
74         state: (x,y) coordinates
75         action: direction from Directions of game.py
76         cost: cost of reaching the node
77         parent: parent node
78         f: the backed-up f-value
79         """
80         self.state = state
81         self.action = action
82         self.cost = cost
83         self.parent = parent
84         self.f = f
```

Listing A.2: Method for reconstructing the path from the start to the goal node.

```
87 def reconstructPath(node):
88     """
89     Reconstructs a path, whose end node is given as a parameter, by
90     iterating through the parent references.
91
92     node: search node, instance of Node
93     """
94     if not node:
```

```

95         return None
96
97     path = []
98     while node.parent is not None:
99         path = path + [node.action]
100         node = node.parent
101     path.reverse()
102     return path

```

A.2 Bidirectional Search

```

228 def bidirectionalSearch(problem):
229     """
230     Simultaneously searches from both the start and the goal positions.
231     Stops when the two frontiers intersect.
232     The shallowest node is taken first (BFS).
233     """
234
235     forward = util.Queue()
236     start = problem.getStartState()
237     exploredForward = {start}
238     forward.push((start, []))
239
240     backward = util.Queue()
241     goal = problem.goal
242     exploredBackward = {goal}
243     backward.push((goal, []))
244
245     visitedForward = set()
246     visitedBackwards = set()
247
248     while not forward.isEmpty() and not backward.isEmpty():
249
250         # Forward searching
251         currentNode, currentActions = forward.pop()
252
253         if currentNode not in visitedForward:
254             visitedForward.add(currentNode)
255             if currentNode in exploredBackward:
256                 while not backward.isEmpty():
257                     node, actions = backward.pop()
258                     if node == currentNode:
259                         solution = currentActions + actions.reverse()
260                         return solution
261
262                 for (childState, childAction, childCost) in problem.
263                     getSuccessors(currentNode):
264                         forward.push((childState, currentActions + [childAction]))
265                         exploredForward.add(childState)
266
267         # Backward searching
268         currentNode, currentActions = backward.pop()
269
270         if currentNode not in visitedBackwards:
271             visitedBackwards.add(currentNode)
272             if currentNode in exploredForward:
273

```

```

274         while not forward.isEmpty():
275             node, actions = forward.pop()
276             if node == currentNode:
277                 backwardActions = [reverseAction(action) for action
in currentActions]
278                 backwardActions.reverse()
279                 solution = actions + backwardActions
280                 return solution
281
282         for (childState, childAction, childCost) in problem.
getSuccessors(currentNode):
283             backward.push((childState, currentActions + [childAction]))
284             exploredBackward.add(childState)
285         return None
286
287 def reverseAction(action):
288     """Changes the action to its inverse."""
289
290     if action == 'North':
291         return 'South'
292     elif action == 'South':
293         return 'North'
294     elif action == 'West':
295         return 'East'
296     else:
297         return 'West'
298

```

A.3 Depth Limited Search

```

366 def depthLimitedSearch(problem, limit):
367     """
368     Search the deepest nodes in the search tree first. The search depth is
limited by the given parameter.
369
370     returns a (node, cutoff) tuple:
371     - node is the goal node containing reference to its parent.
372     - cutoff is True if no solution was found in the given limit, False
otherwise.
373     """
374
375     start_node = Node(problem.getStartState(), None, 0, None)
376     visited = {start_node.state}
377     return recursiveDLS(problem, visited, start_node, limit)
378
379 def recursiveDLS(problem, visited, node, limit):
380     """
381     Helper function for depthLimitedSearch(problem, limit).
382
383     returns a (node, cutoff) tuple:
384     - node: the goal node containing reference to its parent, or False if no
solution was found.
385     - cutoff: True if no solution was found in the given limit, False
otherwise.
386     """
387
388     if problem.isGoalState(node.state):
389

```

```

390         return node, False
391     if limit == 0:
392         return None, True
393
394     cutoff_occurred = False
395     for child_state, child_action, child_cost in problem.getSuccessors(node.
state):
396         if child_state not in visited:
397
398             child = Node(child_state, child_action, child_cost, node)
399
400             visited.add(child_state)
401             (result, cutoff) = recursiveDLS(problem, visited, child, limit -
1)
402             visited.remove(child_state)
403
404             if cutoff:
405                 cutoff_occurred = True
406             elif result:
407                 return result, False
408
409     if cutoff_occurred:
410         return None, True
411     else:
412         return None, False

```

A.4 Iterative Deepening Depth-First Search

```

416 def iterativeDeepeningSearch(problem):
417     """
418     Search the deepest nodes in the search tree first. The search depth is
limited, but the limit is increased in each
419     iteration.
420
421     returns:
422     - node: the goal node containing reference to its parent, or False if no
solution was found.
423     """
424
425     depth = 0
426     while True:
427         (result, cutoff) = depthLimitedSearch(problem, depth)
428         if not cutoff:
429             return reconstructPath(result)
430         depth += 1

```

A.5 Iterative Deepening A*

```

433 def iterativeDeepeningAStarSearch(problem, heuristic=nullHeuristic):
434     """
435     Search the node that has the lowest combined cost and heuristic first.
It differs from A* by limiting the size of
436     the frontier, using a bound on the f value.
437     """

```

```

438     start = problem.getStartState()
439     bound = heuristic(start, problem)
440     path = [(start, None, 0)]
441     visited = {start}
442
443
444     while True:
445         t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path,
446         visited, bound)
447
448         if t is None:
449             return [action for (state, action, cost) in path if action is
450             not None]
451         if t == sys.maxint:
452             return None
453
454         bound = t
455
456 def iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited,
457 bound):
458     """
459     Utility function for iterativeDeepeningAStarSearch.
460     Returns the minimum cost of all values that exceeded the current bound.
461     """
462     state, action, cost = path[-1]
463     f = cost + heuristic(state, problem)
464
465     if f > bound:
466         return f
467     if problem.isGoalState(state):
468         return None
469
470     min = sys.maxint
471
472     for child_state, child_action, child_cost in problem.getSuccessors(state
473 ):
474         if child_state not in visited:
475             visited.add(child_state)
476             path.append((child_state, child_action, child_cost + cost))
477
478             t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path,
479             visited, bound)
480
481             if t is None:
482                 return None
483             if t < min:
484                 min = t
485
486             path.pop()
487             visited.remove(child_state)
488
489     return min

```

A.6 Recursive Best-First Search

```

301 def recursiveBestFirstSearch(problem, heuristic=nullHeuristic):
302     """

```

```

303     Similarly to the DFS, search the deepest nodes in the search tree first,
304     but uses the f_limit variable to keep
305     track of the f values. The f value is the largest reached  $g(n) + h(n)$ 
306     value upon one path after the search is
307     stopped because of the f_limit. This f values is backed up to the parent
308     of a node upon backtracking. The f_limit of
309     the best child path is the f value of the best alternative path.
310
311     returns:
312     - node: the goal node containing reference to its parent, or False if no
313       solution was found.
314     """
315     start_node = Node(problem.getStartState(), None, 0, None)
316     start_node.f = 0
317     visited = {start_node.state}
318     result, _ = RBFS(problem, heuristic, visited, start_node, float('inf'))
319
320     return reconstructPath(result)
321
322 def RBFS(problem, heuristic, visited, node, f_limit):
323     """
324     Helper function of recursiveBestFirstSearch(problem, heuristic).
325
326     returns:
327     - node: the goal node containing reference to its parent, or None if no
328       solution was found.
329     - f-cost: the f value obtained on the path.
330     """
331
332     if problem.isGoalState(node.state):
333         return node, None
334
335     successors = util.PriorityQueueWithFunction(lambda n: n.f)
336
337     for child_state, child_action, child_cost in problem.getSuccessors(node.
338 state):
339         if child_state not in visited:
340             path_cost = child_cost + node.cost
341             child_f = max(path_cost + heuristic(child_state, problem), node.
342 f)
343             child = Node(child_state, child_action, path_cost, node, child_f
344 )
345             successors.push(child)
346
347     if successors.isEmpty():
348         return None, float('inf')
349
350     while True:
351         best = successors.pop()
352
353         if best.f > f_limit:
354             return None, best.f
355
356         if successors.isEmpty():
357             alternative_f = float('inf')
358         else:
359             alternative = successors.pop()
360             alternative_f = alternative.f
361             successors.push(alternative)

```

```
355
356     visited.add(best.state)
357     result, best.f = RBFS(problem, heuristic, visited, best, min(f_limit
, alternative_f))
358     visited.remove(best.state)
359
360     if result:
361         return result, best.f
362
363     successors.push(best)
```


Appendix B

Source Code: Logic Puzzles

The following sections contain the Mace4 input files for solving the corresponding logic puzzles. They can be found in the `logic` directory. The solutions are discussed in details in section 2.3.

B.1 The First Trial

B.1.1 Propositional logic

Listing B.1: `propositional/1.in`

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % l1: There is a lady in room 1
6 % l2: There is a lady in room 2
7 % t1: There is a tiger in room 1
8 % t2: There is a tiger in room 2
9 % m1: Message on the door of room 1
10 % m2: Message on the door of room 2
11
12 formulas(assumptions).
13
14     % Each of the two rooms contained either a lady or a tiger,
15     % but it could be that there were tigers in both rooms, or ladies in
16     % both rooms.
17     % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant
18
19     % No tiger in the room where the lady stays
20     l1 -> -t1.
21     l2 -> -t2.
22
23     % Message on door #1: In this room there is a lady, and in the other
24     % room there is a tiger.
25     m1 <-> l1 & t2.
26
27     % Message on door #2: In one of these rooms there is a lady, and in one
28     % of these rooms there is a tiger.
29     m2 <-> (l1 | l2) & (t1 | t2).
30
31     % One of the messages is true, but the other one is false.
32     m1 <-> -m2.
33
34 end_of_list.
```

B.1.2 Modular arithmetic

Listing B.2: arithmetic/1.in

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % m1 = ["The message on the door of room 1 is true"]
6 % m2 = ["The message on the door of room 2 is true"]
7 % r1 = ["There is a lady in room 1"]
8 %     = 1 + ["There is a tiger in room 1"]
9 % r2 = ["There is a lady in room 2"]
10 %     = 1 + ["There is a tiger in room 2"]
11
12 formulas(assumptions).
13
14     % Message on door #1: In this room there is a lady, and in the other
15     room there is a tiger.
16     m1 = r1 * (r2 + 1).
17
18     % Message on door #2: In one of these rooms there is a lady, and in one
19     of these rooms there is a tiger.
20     m2 = r1 + r2.
21
22     % One of the messages is true, but the other one is false.
23     m1 + m2 = 1.
24
25 end_of_list.
```

B.2 The Second Trial

B.2.1 Propositional logic

Listing B.3: propositional/2.in

```
1
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13
14 % Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 % If lady in room, there is no tiger
18 l1 -> -t1.
19 l2 -> -t2.
20
21 % At least one of these room contains a lady
22 m1 <-> l1 | l2.
```

```

23
24 %A tiger is in the other room
25 m2 <-> t1.
26
27 %The messages are either both true or both false
28 m1 <-> m2.
29
30 end_of_list.

```

B.2.2 Modular arithmetic

Listing B.4: arithmetic/2.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % m1 = ["The message on the door of room 1 is true"]
6 % m2 = ["The message on the door of room 2 is true"]
7 % r1 = ["There is a lady in room 1"]
8 %    = 1 + ["There is a tiger in room 1"]
9 % r2 = ["There is a lady in room 2"]
10 %    = 1 + ["There is a tiger in room 2"]
11
12 formulas(assumptions).
13
14     % Message on door #1: At least one of these rooms contains a lady
15     m1 = r1 * r2 + r1 + r2.
16
17     % Message on door #2: A tiger is in the other room
18     m2 = r1 + 1.
19
20     % Either both are true or both are false
21     % (m1 + m2) mod 2 = 0.
22     m1 = m2.
23
24 end_of_list.

```

B.3 The Third Trial

B.3.1 Propositional logic

Listing B.5: propositional/3.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13

```

```

14 %Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 %If lady in room, there is no tiger
18 l1 -> -t1.
19 l2 -> -t2.
20
21 %Either a tiger is in this room, or a lady is in the other room
22 m1 <-> t1 | l2.
23
24 %A lady is in the other room
25 m2 <-> l1.
26
27 %The messages are either both true or both false
28 (-m1 & -m2) | (m1&m2).
29
30 end_of_list.

```

B.3.2 Modular arithmetic

Listing B.6: arithmetic/3.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % m1 = ["The message on the door of room 1 is true"]
6 % m2 = ["The message on the door of room 2 is true"]
7 % r1 = ["There is a lady in room 1"]
8 %     = 1 + ["There is a tiger in room 1"]
9 % r2 = ["There is a lady in room 2"]
10 %     = 1 + ["There is a tiger in room 2"]
11
12 formulas(assumptions).
13
14     % Either a tiger is in this room or a lady is in the other room
15     % m1 = ((r1 + 1) * r2 + (r1 + 1) + r2) mod 2.
16     % By double negation and De Morgan's Law we obtain:
17     % "It's not true that a lady is in the first room and a tiger in the
18     % second"
19     m1 = (r1 * (r2 + 1) + 1) mod 2.
20
21     % A lady is in the other room
22     m2 = r1.
23
24     % Either both are true or both are false
25     m1 = m2.
26 end_of_list.

```

B.4 The Fourth Trial

B.4.1 Propositional logic

Listing B.7: propositional/4.in

```

1 set(arithmetic).

```

```

2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13
14 %Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 %If lady in room, there is no tiger
18 l1 -> -t1.
19 l2 -> -t2.
20
21 %Both rooms contains ladies
22 m1 <-> l1 & l2.
23
24 %Both rooms contains ladies
25 m2 <-> m1.
26
27 %If lady in first room, message is true. If tiger in first room, message is
    false
28 l1 -> m1.
29 t1 -> -m1.
30
31 %If tiger in second room, message is true, if lady in second room, message
    is false
32 l2 -> -m2.
33 t2 -> m2.
34
35 end_of_list.

```

B.4.2 Modular arithmetic

Listing B.8: arithmetic/4.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % m1 = ["The message on the door of room 1 is true"]
8 % m2 = ["The message on the door of room 2 is true"]
9 % r1 = ["There is a lady in room 1"]
10 %    = 1 + ["There is a tiger in room 1"]
11 % r2 = ["There is a lady in room 2"]
12 %    = 1 + ["There is a tiger in room 2"]
13
14 %Both rooms contain ladies
15 m1 = r1 * r2.
16
17 %Both rooms contain ladies
18 m2 = m1.

```

```

19
20
21
22 %If lady in first room, message is true
23 r1 = m1.
24
25
26 %If lady in second room, message is false
27 r2 = m2 + 1.
28
29 end_of_list.

```

B.5 The Fifth Trial

B.5.1 Propositional logic

Listing B.9: propositional/5.in

```

1
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13
14 %Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 %If lady in room, there is no tiger
18 l1 -> -t1.
19 l2 -> -t2.
20
21 %At least one room contains a lady
22 m1 <-> l1 | l2.
23
24 %The other room contains a lady
25 m2 <-> l1.
26
27 %If lady in first room, message is true. If tiger in first room, message is
   false
28 l1 -> m1.
29 t1 -> -m1.
30
31 %If tiger in second room, message is true, if lady in second room, message
   is false
32 l2 -> -m2.
33 t2 -> m2.
34
35 end_of_list.

```

B.5.2 Modular arithmetic

Listing B.10: arithmetic/5.in

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % m1 = ["The message on the door of room 1 is true"]
8 % m2 = ["The message on the door of room 2 is true"]
9 % r1 = ["There is a lady in room 1"]
10 %    = 1 + ["There is a tiger in room 1"]
11 % r2 = ["There is a lady in room 2"]
12 %    = 1 + ["There is a tiger in room 2"]
13
14 %At least one room contains a lady
15 m1 = r1 * r2 + r1 + r2.
16
17 %The other room contains a lady
18 m2 = r1.
19
20
21
22 %If lady in first room, message is true
23 r1 = m1.
24
25
26 %If lady in second room, message is false
27 r2 = (m2 + 1) mod 2.
28
29 end_of_list.
```

B.6 The Sixth Trial

B.6.1 Propositional logic

Listing B.11: propositional/6.in

```
1
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13
14 %Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 %If lady in room, there is no tiger
18 l1 -> -t1.
```

```

19 l2 -> -t2.
20
21 %It makes no difference which room you pick
22 m1 <-> (l1 & l2) | (t1 & t2).
23
24 %There is a lady in the other room
25 m2 <-> l1.
26
27 %If lady in first room, message is true. If tiger in first room, message is
    false
28 l1 -> m1.
29 t1 -> -m1.
30
31 %If tiger in second room, message is true, if lady in second room, message
    is false
32 l2 -> -m2.
33 t2 -> m2.
34
35 end_of_list.

```

B.6.2 Modular arithmetic

Listing B.12: arithmetic/6.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6 % m1 = ["The message on the door of room 1 is true"]
7 % m2 = ["The message on the door of room 2 is true"]
8 % r1 = ["There is a lady in room 1"]
9 %     = 1 + ["There is a tiger in room 1"]
10 % r2 = ["There is a lady in room 2"]
11 %     = 1 + ["There is a tiger in room 2"]
12
13 %It makes no difference which room you pick
14 m1 = (r1 + r2 + 1) mod 2.
15
16 %The other room contains a lady
17 m2 = r1.
18
19
20 %If lady in first room, message is true
21 r1 = m1.
22
23
24 %If lady in second room, message is false
25 r2 = (m2 + 1) mod 2.
26
27 end_of_list.

```

B.7 The Seventh Trial

B.7.1 Propositional logic

Listing B.13: propositional/7.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % m1: Message on the door of room 1
12 % m2: Message on the door of room 2
13
14 %Either one lady and one tiger or two tigers or two ladies
15 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
16
17 %If lady in room, there is no tiger
18 l1 -> -t1.
19 l2 -> -t2.
20
21 %It does make a difference which room you pick
22 m1 <-> (l1 & t2) | (l2 & t1).
23
24 %You are better off choosing the other room
25 m2 <-> l1.
26
27 %If lady in first room, message is true. If tiger in first room, message is
    false
28 l1 -> m1.
29 t1 -> -m1.
30
31 %If tiger in second room, message is true, if lady in second room, message
    is false
32 l2 -> -m2.
33 t2 -> m2.
34
35 end_of_list.

```

B.7.2 Modular arithmetic

Listing B.14: arithmetic/7.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6 % m1 = ["The message on the door of room 1 is true"]
7 % m2 = ["The message on the door of room 2 is true"]
8 % r1 = ["There is a lady in room 1"]
9 %     = 1 + ["There is a tiger in room 1"]
10 % r2 = ["There is a lady in room 2"]
11 %     = 1 + ["There is a tiger in room 2"]
12
13
14
15 %It does make a difference which room you pick
16 m1 = (r1 + r2) mod 2.

```

```

17
18 %You are better off choosing the other room
19 m2 = r1.
20
21
22 %If lady in first room, message is true
23 r1 = m1.
24
25
26 %If lady in second room, message is false
27 r2 = (m2 + 1) mod 2.
28
29 end_of_list.

```

B.8 The Eighth Trial

B.8.1 Propositional logic

Listing B.15: propositional/8.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % mij: Message i on the door of room j
12
13 %Either one lady and one tiger or two tigers or two ladies
14 l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.
15
16 %If lady in room, there is no tiger
17 l1 -> -t1.
18 l2 -> -t2.
19
20 %This rooms contains a tiger
21 m11 <-> t1.
22 m12 <-> t2.
23
24 %Both rooms contain tigers
25 m21 <-> t1 & t2.
26 m22 <->t1 & t2.
27
28
29 %If lady in first room, message is true. If tiger in first room, message is
    false
30 l1 -> m11 | m21.
31 t1 -> -m11 | -m21.
32
33 %If tiger in second room, message is true, if lady in second room, message
    is false
34 l2 -> -m12 | -m22.
35 t2 -> m12 | m22.
36

```

```
37 end_of_list.
```

B.8.2 Modular arithmetic

Listing B.16: arithmetic/8.in

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6 % mij = [["Message i is on the door of room j and it is true"]]
7 % r1 = [["There is a lady in room 1"]]
8 %     = 1 + [["There is a tiger in room 1"]]
9 % r2 = [["There is a lady in room 2"]]
10 %     = 1 + [["There is a tiger in room 2"]]
11
12 %This room contains a tiger
13 m11 = (r1 + 1) mod 2.
14 m12 = (r2 + 1) mod 2.
15
16 %Both rooms contain a tiger
17 m21 = ((r1 + 1) * (r2 + 1)) mod 2.
18 m22 = m21.
19
20
21 %If lady in first room, message is true
22 (r1 * ( m11 * m21 + m11 + m21) + r1 + 1) mod 2 = 1.
23
24 ((r1 + 1) * ((m11 + 1) * ( m21 + 1) + m11 + m21) + r1) mod 2 = 1.
25
26 %If lady in second room, message is false
27 ((r2 + 1)*(m12 * m22 + m12 + m22) + r2)mod 2= 1.
28
29 (r2 * ((m12 + 1) * (m22 + 1) + m12 + m22) + r2 + 1) mod 2 = 1.
30
31 end_of_list.
```

B.9 The Ninth Trial

B.9.1 Propositional logic

Listing B.17: propositional/9.in

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % l3: There is a lady in room 3
10 % t1: There is a tiger in room 1
11 % t2: There is a tiger in room 2
12 % t3: There is a tiger in room 3
13 % m1: Message on the door of room 1
```

```

14 % m2: Message on the door of room 2
15 % m3: Message on the door of room 3
16
17 %One lady and two tigers
18 l1 & t2 & t3 | l2 & t1 & t3 | l3 & t1 & t2.
19
20 %no lady in more than 1 room; the princess is unique
21 l1 -> -l2.
22 l1 -> -l3.
23 l2 -> -l1.
24 l2 -> -l3.
25 l3 -> -l1.
26 l3 -> -l2.
27
28
29 %no tiger in the room where the lady stays
30 l1 -> -t1.
31 l2 -> -t2.
32 l3 -> -t3.
33
34 %A tiger is in this room
35 m1 <-> t1.
36
37 %A lady is in this room
38 m2 <-> l2.
39
40 %A tiger is in room two
41 m3 <-> t2.
42
43 %At most one sign is true
44 m1 -> -m2 & -m3.
45 m2 -> -m3.
46
47 end_of_list.

```

B.9.2 Modular arithmetic

Listing B.18: arithmetic/9.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6 % m1 = ["The message on the door of room 1 is true"]
7 % m2 = ["The message on the door of room 2 is true"]
8 % m3 = ["The message on the door of room 3 is true"]
9 % r1 = ["There is a lady in room 1"]
10 %    = 1 + ["There is a tiger in room 1"]
11 % r2 = ["There is a lady in room 2"]
12 %    = 1 + ["There is a tiger in room 2"]
13 % r3 = ["There is a lady in room 3"]
14 %    = 1 + ["There is a tiger in room 3"]
15
16 %Lady in one room and tigers in the other two rooms
17
18 % One lady and two tigers
19 r1 * r2 * r3 = 0.
20 r1 + r2 + r3 = 1.

```

```

21
22 %Tiger in this room
23 m1 = (r1 + 1) mod 2.
24
25 %Lady in this room
26 m2 = r2.
27
28
29 %Tiger in room 2
30 m3 = (r2 + 1) mod 2.
31
32
33 %At most one of the three signs is true
34 m1 * m2 + m1 * m3 + m2 * m3 + 1 = 1.
35
36 end_of_list.

```

B.10 The Tenth Trial

B.10.1 Propositional logic

Listing B.19: propositional/10.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 % l1: There is a lady in room 1
8 % l2: There is a lady in room 2
9 % l3: There is a lady in room 3
10 % t1: There is a tiger in room 1
11 % t2: There is a tiger in room 2
12 % t3: There is a tiger in room 3
13 % m1: Message on the door of room 1
14 % m2: Message on the door of room 2
15 % m3: Message on the door of room 3
16
17 %One lady and two tigers
18 l1 & t2 & t3 | l2 & t1 & t3 | l3 & t1 & t2.
19
20 %no lady in more than 1 room; the princess is unique
21 l1 -> -l2.
22 l1 -> -l3.
23 l2 -> -l1.
24 l2 -> -l3.
25 l3 -> -l1.
26 l3 -> -l2.
27
28
29 %no tiger in the room where the lady stays
30 l1 -> -t1.
31 l2 -> -t2.
32 l3 -> -t3.
33
34 %A tiger is in room two
35 m1 <-> t2.

```

```

36
37 %A tiger is in this room
38 m2 <-> t2.
39
40 %A tiger is in room one
41 m3 <-> t1.
42
43 %The sign on the door, where the lady is, is true
44 l1 -> m1.
45 l2 -> m2.
46 l3 -> m3.
47
48 %From the other two signs at least one is false
49 m1 -> -m2 | -m3.
50 m2 -> -m1 | -m3.
51 m3 -> -m1|-m2.
52
53 end_of_list.

```

B.10.2 Modular arithmetic

Listing B.20: arithmetic/10.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 % m1 = ["The message on the door of room 1 is true"]
6 % m2 = ["The message on the door of room 2 is true"]
7 % r1 = ["There is a lady in room 1"]
8 %    = 1 + ["There is a tiger in room 1"]
9 % r2 = ["There is a lady in room 2"]
10 %    = 1 + ["There is a tiger in room 2"]
11
12 formulas(assumptions).
13     % One lady and two tigers
14     r1 * r2 * r3 = 0.
15     r1 + r2 + r3 = 1.
16
17     % The message on the door containing the lady is true
18     (r1 * (m1 + 1)) mod 2 = 0.
19     (r2 * (m2 + 1)) mod 2 = 0.
20     (r3 * (m3 + 1)) mod 2 = 0.
21
22     % At least one of the other two messages is false
23     m1 * m2 * m3 = 0.
24     ((m1 + 1) * (m2 + 1) * (m3 + 1)) mod 2 = 0.
25
26     % Message on door #1: There is a tiger in room #2
27     m1 = r2 + 1.
28
29     % Message on door #2: There is a tiger here
30     m2 = r2 + 1.
31
32     % Message on door #3: There is a tiger in room #1
33     m3 = (r1 + 1) mod 2 .
34 end_of_list.

```

B.11 First, Second, and Third Choice

B.11.1 Propositional logic

Listing B.21: propositional/11.in

```
1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6 % l1: There is a lady in room 1
7 % l2: There is a lady in room 2
8 % l3: There is a lady in room 3
9 % t1: There is a tiger in room 1
10 % t2: There is a tiger in room 2
11 % t3: There is a tiger in room 3
12 % e1: Room 1 is empty
13 % e2: Room 2 is empty
14 % e3: Room 3 is empty
15 % m1: Message on the door of room 1
16 % m2: Message on the door of room 2
17 % m3: Message on the door of room 3
18
19
20
21 %there is a lady in room 1, 2 or 3
22 l1 | l2 | l3.
23
24 %no lady in more than 1 room; the princess is unique
25 l1 -> -l2 & -l3.
26 l2 -> -l1 & -l3.
27 l3 -> -l1 & -l2.
28
29 %no tiger in the room where the lady stays, room not empty if lady is in it
30 l1 -> -t1 & -e1.
31 l2 -> -t2 & -e2.
32 l3 -> -t3 & -e3.
33
34
35 t1 | t2 | t3.
36
37
38 %no tiger in more than 1 room
39 t1 -> -t2 & -t3.
40 t2 -> -t1 & -t3.
41 t3 -> -t2 & -t1.
42
43 %if there is a tiger in the room it can not be empty
44 t1 -> -e1.
45 t2 -> -e2.
46 t3 -> -e3.
47
48 e1 | e2 | e3.
49
50 %no more than one empty room
51 e1 -> -e2 & -e3.
52 e2 -> -e1 & -e3.
53 e3 -> -e2 & -e1.
54
```

```

55
56 %Room three is empty
57 m1 <-> e3.
58
59 %There is a tiger in room one.
60 m2 <-> t1.
61
62 %This room is empty
63 m3 <-> e3.
64
65 %If lady in room, the message is true
66 l1 -> m1.
67 l2 -> m2.
68 l3 -> m3.
69
70 %If tiger in room, the message is false
71 t1 -> -m1.
72 t2 -> -m2.
73 t3 -> -m3.
74
75
76
77 end_of_list.

```

B.11.2 Modular arithmetic

Listing B.22: arithmetic/11.in

```

1 set(arithmetic).
2 assign(domain_size, 2).
3 assign(max_models, -1).
4
5 formulas(assumptions).
6
7 %mi = ["The message on the door of room i is true"]
8 %li = ["There is a lady in room i"]
9 %ei = ["Room i is empty"]
10 %ti = ["There is a tiger in room i"]
11
12
13 %One empty, one lady, one tiger
14 l1 * l2 * l3 = 0.
15 l1 + l2 + l3 = 1.
16
17
18 t1 * t2 * t3 = 0.
19 t1 + t2 + t3 = 1.
20
21
22 e1 * e2 * e3 = 0.
23 e1 + e2 + e3 = 1.
24
25 %One room can only contain a lady/tiger or it is empty
26 (l1 * ((t1 + 1) * (e1 + 1)) + l1 + 1) mod 2 = 1.
27 (l2 * ((t2 + 1) * (e2 + 1)) + l2 + 1) mod 2 = 1.
28 (l3 * ((t3 + 1) * (e3 + 1)) + l3 + 1) mod 2 = 1.
29
30
31 (t1 * ((e1 + 1) * (l1 + 1)) + t1 + 1) mod 2 = 1.

```



```

32 (t2 * ((e2 + 1) * (l2 + 1)) + t2 + 1) mod 2 = 1.
33 (t3 * ((e3 + 1) * (l3 + 1)) + t3 + 1) mod 2 = 1.
34
35 %Room three is empty
36 m1 = e3.
37
38 %There is a tiger in room one.
39 m2 = t1.
40
41 %This room is empty
42 m3 = e3.
43
44 %If lady in room, the message is true
45 (l1 * m1 + l1 + 1) mod 2 = 1.
46 (l2 * m2 + l2 + 1) mod 2 = 1.
47 (l3 * m3 + l3 + 1) mod 2 = 1.
48
49 %If tiger in room, the message is false
50 (t1 * (m1 + 1) + t1 + 1) mod 2 = 1.
51 (t2 * (m2 + 1) + t2 + 1) mod 2 = 1.
52 (t3 * (m3 + 1) + t3 + 1) mod 2 = 1.
53
54
55
56
57 end_of_list.

```

Appendix C

Source Code: Translator

The following sections contain the source code of the Python program which translates Mace4 input files in the propositional logic form to the modular arithmetic equivalent. They can be found in the `logic/translator` directory. The program is discussed in more details in section 2.5.

C.1 Input data representation

Listing C.1: `commands.py`

```
1 class Command:
2     """
3     Class for representing mace4 commands.
4     Commands are given as: command_name(argument1[,argument2,...]).
5     """
6     def __init__(self, name, args):
7         self.name = name
8         self.args = args
9
10    def __str__(self):
11        return "Command(%s, %s)" % (self.name, self.args)
12
13    def __repr__(self):
14        return self.__str__()
15
16
17 class Formulas:
18     """
19     Class for representing mace4 formulas.
20     Formulas are given as:
21         formulas(name).
22         predicate1.
23         [predicate2.
24         ...]
25         end_of_list.
26     """
27
28     commandName = "formulas"
29     endCommand = "end_of_list"
30
31    def __init__(self, name, formula_list):
32        self.name = name
33        self.formulaList = formula_list
34
```

```

35     def __str__(self):
36         return "Formulas(%s, %s)" % (self.name, self.formulaList)
37
38     def __repr__(self):
39         return self.__str__()

```

C.2 Parsing the Mace4 input file

Listing C.2: input.py

```

1  """
2  Functions for processing the input data.
3  """
4
5  from commands import Command, Formulas
6
7  commentSymbol = "%"
8  lineSeparator = "."
9
10
11 def processLine(line):
12     """
13     Processes a single line of the input. It removes the comments and the
14     leading and trailing whitespaces.
15
16     :param line: a line of the input data
17     :return: the result after processing, or None if the result is empty.
18     """
19
20     # Remove comments
21     line = line.split(commentSymbol, 1)[0]
22
23     # Remove leading and trailing whitespace
24     line = line.strip()
25
26     # Remove empty lines
27     if not line:
28         return None
29
30     return line
31
32 def changeLineSeparator(lines, separator):
33     """
34     Given a list of lines, changes the line separator to the given one.
35     Thus merges all the lines and separates them by the given separator.
36
37     :param lines: list containing the lines of the input data
38     :param separator: the desired line separator
39     :return: list of strings representing the lines of the input data
40     separated by the given parameter
41     """
42
43     lines = " ".join(lines).split(separator)
44
45     if lines[-1]:
46         raise ValueError("Terminating \"%s\" not found" % lineSeparator)

```

```

47     return map(lambda x: x.strip(), lines[:-1])
48
49
50 def processLines(lines):
51     """
52     Processes each line of the input data.
53
54     :param lines: list of lines of the input data
55     :return: list of non-empty lines of the input data, separated by '
lineSeparator'
56     """
57
58     # Process each line. Remove the empty lines.
59     lines = filter(lambda x: x, map(processLine, lines))
60
61     if not lines:
62         raise ValueError("Input file is empty")
63
64     # Change the line separator from the newline character to a different
one.
65     # Thus merging the lines not terminating in the new character.
66     lines = changeLineSeparator(lines, lineSeparator)
67
68     return lines
69
70
71 def linesToCommandsAndFormulas(lines):
72     """
73     Transforms the lines of data into lists of Command and Formulas objects.
74
75     :param lines: lines of the input data
76     :return: (commands, formulas) tuples obtained by parsing the lines
77     """
78     commands = []
79     formulas = []
80     formulaName = None
81     propositions = []
82
83     for line in lines:
84
85         if formulaName:
86             if line == Formulas.endCommand:
87                 formulas.append(Formulas(formulaName, propositions))
88                 formulaName = None
89             else:
90                 propositions.append(line)
91         else:
92             try:
93                 name, rest = line.split("(", 1)
94                 args = rest.split(")")[0].split(",")
95                 args = map(lambda x: x.strip(), args)
96             except ValueError:
97                 raise ValueError("Invalid command format: \"%s\" % line)
98
99             if name == Formulas.commandName:
100                 formulaName = args[0]
101                 propositions = []
102             else:
103                 command = Command(name, args)
104                 commands.append(command)

```

```

105
106     return commands, formulas

```

C.3 Creating/transforming Boolean ring expression trees

Listing C.3: expression.py

```

1  """
2  Functions for
3  - creating sympy expression trees in mod 2 arithmetics from proposition
4    logic.
5  - transforming sympy expression trees
6  """
7
8  import re
9
10 import sympy
11 from sympy import Mul, Add, Symbol, Poly
12
13 from commands import Formulas
14
15 # Priorities of the propositional logic operators
16 priority = {
17     "(": 0,
18     ")": 0,
19     "->": 1,
20     "<->": 1,
21     "|": 2,
22     "&": 3,
23     "-": 4
24 }
25
26 def is_symbol_char(char):
27     """
28     Verifies whether a character can be part of a symbol or not.
29
30     :param char: character to be verified
31     :return: True or False
32     """
33     return re.search("[a-zA-z0-9]+$", char) is not None
34
35 def operatorStringToExpression(operator, *operands):
36     """
37     Given a string representing a propositional logic operator, returns a
38     sympy expression which is the modular
39     arithmetic translation of the operator.
40
41     This function simplifies the expression according to the rules of
42     general algebra and does not consider the rules of
43     the mod 2 arithmetics.
44
45     :param operator: string representing a propositional logic operator
46     :param operands: list of sympy expressions or integers
47     :return: sympy expression which results from applying the operands to
48             the translated operator
49     """
50     if operator == "&":

```

```

48     return Mul(*operands)
49 elif operator == "-":
50     return Add(operands[0], 1)
51 elif operator == "|":
52     return Add(Add(*operands), Mul(*operands))
53 elif operator == "<->":
54     return Add(Add(*operands), 1)
55 elif operator == "->":
56     return Add(Mul(*operands), Add(operands[0], 1))
57 else:
58     raise ValueError("Unknown operator \"%s\" % operator)
59
60
61 def splitProposition(proposition):
62     """
63     Splits a proposition into terms of operands and operators.
64
65     :param proposition: proposition to be split
66     :return: list of strings representing the operands and operators of the
67             proposition in order
68     """
69     terms = []
70     is_symbol = is_symbol_char(proposition[0])
71     term = ""
72
73     def addTerm(t):
74         t = t.strip()
75         if t:
76             terms.append(t)
77
78     for char in proposition:
79         is_symbol2 = is_symbol_char(char)
80
81         if char in " ":
82             addTerm(term)
83             term = ""
84             is_symbol = False
85             continue
86
87         if char in "()":
88             addTerm(term)
89             term = ""
90             is_symbol = False
91             terms.append(char)
92             continue
93
94         if is_symbol == is_symbol2:
95             term = term + char
96         else:
97             addTerm(term)
98             term = char
99
100     is_symbol = is_symbol2
101
102     if term:
103         addTerm(term)
104
105     return terms
106

```

```

107 def buildExpressionTree(proposition, substitutions=None):
108     """
109     Builds a sympy expression tree from the given proposition in
110     propositional logic.
111
112     The result is an expression (tree) in modulo 2 arithmetics. However it
113     is simplified based only on the general rules
114     of algebra. E.g. x * x will be x**2 instead of x.
115
116     Substitutes the symbols to the sympy expressions associated with them in
117     the substitutions dictionary.
118
119     :param proposition: string, in propositional logic
120     :param substitutions: a dictionary whose elements are (string_symbol, (
121     expression, symbols)).
122
123     No substitution if None is given.
124     string_symbol: the string which will be substituted
125     expression: a sympy expression which substitutes the string
126     symbols: the list of sympy symbols that are present in the
127     expression
128     :return: a sympy expression (tree)
129     """
130     operators = set(priority.keys())
131
132     node_stack = []
133     term_stack = []
134     symbols = set()
135
136     terms = splitProposition(proposition)
137
138     def createAndPush():
139         """
140         Creates a new node and pushes to the node stack
141
142         :return: None
143         """
144         op = term_stack.pop()
145
146         if op == '-':
147             # Unary operator
148             node = operatorStringToExpression(op, node_stack.pop())
149         else:
150             # Binary operator
151             right = node_stack.pop()
152             left = node_stack.pop()
153             node = operatorStringToExpression(op, left, right)
154
155         node_stack.append(node)
156
157     for term in terms:
158         if term == '(':
159             term_stack.append(term)
160         elif term not in operators:
161             if not substitutions or term not in substitutions:
162                 node = Symbol(term)
163                 symbols.add(node)
164             else:
165                 node, node_symbols = substitutions[term]
166                 symbols.update(node_symbols)
167             node_stack.append(node)

```

```

162     elif priority[term] > 0:
163         while term_stack \
164             and term_stack[-1] != '(' \
165             and priority[term_stack[-1]] >= priority[term]:
166             createAndPush()
167
168         term_stack.append(term)
169
170     elif term == ')':
171         while term_stack and term_stack[-1] != '(':
172             createAndPush()
173         term_stack.pop()
174
175     while term_stack:
176         createAndPush()
177
178     return node_stack[-1], symbols
179
180
181 def simplifyExpression(expr, symbols):
182     """
183     Simplifies a sympy expression by applying also the rules of the boolean
184     ring.
185
186     1. Uses the sympy.simplify function for modulus 2 (implicitly, by
187     creating a polynomial).
188     2. Replaces all the powers with their bases. Because  $x * x = x$  for all  $x$ 
189     in  $\{0, 1\}$ .
190
191     :param expr: sympy expression to be simplified
192     :param symbols: symbols that are used in the expression
193     :return: the simplified sympy expression
194     """
195
196     def recurse_replace(expr, func):
197         """
198         Traverses the expression tree in a DFS order and replaces all nodes
199         with the result of applying them to the
200         given function.
201
202         :param expr: the expression tree
203         :param func: the transformation function
204         :return: the transformed expression tree
205         """
206         if len(expr.args) == 0:
207             return expr
208         else:
209             new_args = tuple(recurse_replace(a, func) for a in expr.args)
210             return func(expr, new_args)
211
212     def rewrite(expr, new_args):
213         """
214         Traverses the arguments of a sympy expression node. If an argument
215         is a power, it is replaced by the base of the
216         power.
217
218         :param expr: expression node or polynomial whose arguments are
219         verified
220         :param new_args: a list of nodes (sympy expression or integer) that
221         would be the new arguments of the expr node.

```



```

215         :return: an object having the same type as expr, and having as
arguments the transformed new_args list.
216         """
217         args = []
218
219         for arg in new_args:
220             if hasattr(arg, 'is_Pow') and arg.is_Pow:
221                 args.append(arg.args[0])
222             else:
223                 args.append(arg)
224
225         if hasattr(expr, 'is_Poly') and expr.is_Poly:
226             return Poly(args[0], expr.gens)
227         else:
228             new_node = type(expr)(*args)
229             return new_node
230
231     p = Poly(expr, symbols, modulus=2)
232     p = recurse_replace(p, rewrite).set_modulus(2)
233
234     return p.as_expr()
235
236
237 def getSymbolsByDecreasingFrequency(expression):
238     """
239     Returns all the symbols of the given sympy expression in decreasing
order of their frequency of appearance.
240
241     Example:
242     expression: m1 + r1*r2 + r1 + 1
243     frequency: {m1: 1, r2: 1, r1: 2}
244     result: [r1, m1, r2]
245
246     :param expression: sympy expression
247     :return: list of sympy symbols
248     """
249     freq = {}
250     for node in sympy.preorder_traversal(expression):
251         if hasattr(node, 'is_Symbol') and node.is_Symbol:
252             freq[node] = freq.get(node, 0) + 1
253     symbols = sorted(freq.keys(), key=lambda k: freq[k], reverse=True)
254     return symbols
255
256
257 def collectExpression(expression):
258     """
259     Groups the terms of the expressions, based on the frequency of its
symbols.
260
261     Example:
262     expression: m1 + r1*r2 + r1 + 1
263     frequency: {m1: 1, r2: 1, r1: 2}
264     result: m1 + r1*(r2 + 1) + 1
265
266     :param expression:
267     :return:
268     """
269     symbols = getSymbolsByDecreasingFrequency(expression)
270     return sympy.collect(expression, symbols)
271

```

```

272
273 def formulasToExpressionTree(formulas, merge=False, simplify=False, collect=
274     False, substitutions=None):
275     """
276     Transforms a Formulas object, by replacing the propositional logic
277     propositions with sympy algebraic expressions.
278
279     :param formulas: the Formulas object whose propositions are to be
280     transformed
281     :param merge: specifies whether to create a single expression from all
282     the expressions of the Formulas object by
283     multiplying them (logical AND)
284     :param simplify: specifies whether to simplify the expressions using the
285     rules of the boolean ring.
286     If False is given, the result can have coefficients and
287     powers that are not part of the ring.
288     :param collect: specifies whether to collect the terms of each
289     expression by the decreasing frequency of their
290     symbols.
291     :param substitutions: a dictionary whose elements are (string_symbol, (
292     expression, symbols)).
293     No substitution if None is given.
294     string_symbol: the string which will be substituted
295     expression: a sympy expression which substitutes the string
296     symbols: the list of sympy symbols that are present in the
297     expression
298     :return: Formula object with the same name, but the propositions
299     replaced with sympy expression(s)
300     """
301     symbols = set()
302
303     def transform(prop):
304         expr, sym = buildExpressionTree(prop, substitutions)
305         symbols.update(sym)
306         return expr
307
308     expressions = map(transform, formulas.formulaList)
309
310     if merge:
311         expressions = [reduce(lambda a, b: Mul(a, b), expressions, 1)]
312
313     if simplify:
314         expressions = map(lambda e: simplifyExpression(e, symbols),
315             expressions)
316
317     if collect:
318         expressions = map(collectExpression, expressions)
319
320     return Formulas(formulas.name, expressions)
321
322 def createSubstitutions(predicates):
323     """
324     Given a list of predicates in the form of 'a <-> f(b)', returns a
325     dictionary which associates
326     to each symbol 'a' the sympy expression representing 'f(b)' and the used
327     symbols.
328
329     :param predicates: list of strings, each having the form 'a <-> f(b)',
330     where 'a' is a string propositional logic

```

```

318         symbol, and 'f(b)' is the proposition which replaces
        'a'.
319     :return: a dictionary whose elements are (string_symbol, (expression,
symbols)).
320         string_symbol: the string which will be substituted
321         expression: a sympy expression which substitutes the string
322         symbols: the list of sympy symbols that are present in the
expression
323     """
324     substitutions = {}
325
326     for predicate in predicates:
327         terms = splitProposition(predicate)
328         if len(terms) < 2 or terms[1] != '<->':
329             raise ValueError('Substitutions of type 'a <-> f(b)' are
supported only')
330         substitutions[terms[0]] = buildExpressionTree(" ".join(terms[2:]))
331
332     return substitutions

```

C.4 Printing the result

Listing C.4: output.py

```

1  """
2  Functions for printing the program output to STDOUT.
3  """
4
5
6  def printCommand(command):
7      """
8      Prints an instance of the Command class.
9
10     :param command: command to be printed
11     :return: None
12     """
13     print command.name + '(' + ', '.join(command.args) + ') .'
14
15
16  def printCommands(commands):
17      """
18      Prints a list of Command objects.
19
20     :param commands: list of commands to be printed
21     :return: None
22     """
23     for command in commands:
24         if command.args and command.args[0] not in ['arithmetic', '
domain_size']:
25             printCommand(command)
26     print "set(arithmetic)."
27     print "assign(domain_size, 2)."
28
29
30  def printFormulas(formulas, mod20Output=False):
31      """
32      Prints an instance of the Formulas class.
33

```

```

34 :param formulas: object to be printed
35 :param mod2Output: if True: wraps each expression with "mod 2"
36 :return: None
37 """
38 print formulas.commandName + '(' + formulas.name + ') .'
39
40 for expression in formulas.formulaList:
41     if mod2Output:
42         print '\t(' + str(expression) + ') mod 2 = 1.'
43     else:
44         print '\t' + str(expression) + ' = 1.'
45
46 print formulas.endCommand + ' .'
47
48
49 def printFormulasList(formulasList, mod2Output=False):
50     """
51     Prints a list of Formulas objects.
52
53     :param formulasList: list of objects to be printed
54     :param mod2Output: if True: wraps each expression with "mod 2"
55     :return: None
56     """
57     for formulas in formulasList:
58         printFormulas(formulas, mod2Output)
59     print
60
61
62 def printCommandsAndFormulas(commands, formulas, mod2Output=False):
63     """
64     Prints a list of Command and a list of Formulas objects.
65
66     :param commands: list of Command objects to be printed
67     :param formulas: list of Formulas objects to be printed
68     :param mod2Output: if True: wraps each formula expression with "mod 2"
69     :return: None
70     """
71     printCommands(commands)
72     print
73     printFormulasList(formulas, mod2Output)

```

C.5 Main program

Listing C.5: translator.py

```

1  """
2  Main program
3  """
4
5  import getopt
6  import os
7  import sys
8
9  from expression import formulasToExpressionTree, createSubstitutions
10 from input import processLines, linesToCommandsAndFormulas
11 from output import printCommandsAndFormulas
12
13 # Name of the program

```

```

14 programName = os.path.basename(__file__)
15
16 # Program configuration
17 inputFile = sys.stdin
18 mod2Output = False
19 collect = False
20 substitute = False
21 merge = False
22
23 # Help message
24 usage = "%s [options] [-f <inputfile>]" % programName
25 description = "A converter from propositional logic to modular arithmetic in
    the mace4 format"
26 help_message = """
27 Usage: %s
28
29 %s
30
31 Options:
32     -c,\t--collect\tCollects the terms of each expression by the decreasing
    frequency of their symbols.
33     -h,\t--help\t\tPrints help message
34     -f,\t--file string\tPath to the input file. If the option is missing,
    the program reads from STDIN
35     -m,\t--merge\t\tMerges all the expressions of a "formulas" section into
    a single expression.
36     \t--mod\t\tWraps all expressions in "mod 2"
37     -s,\t--subs\t\tSubstitutes symbols with their corresponding expression.
    The substitutions must be given in the "a <-> f(b)." format in "formulas(
    substitutions)."
38 """ % (usage, description)
39
40
41 def processArguments(argv):
42     """
43     Reads the command line arguments of the program and updates respectively
    the global variables
44
45     :param argv: the command line arguments, list of strings
46     :return: None
47     """
48     global inputFile, mod2Output, collect, substitute, merge
49
50     try:
51         opts, args = getopt.getopt(argv, "hcsmf:", ["mod", "help", "collect"
52 , "subs", "merge", "file="])
53     except getopt.GetoptError:
54         print help_message
55         sys.exit(2)
56     for opt, arg in opts:
57         if opt in ['-h', '--help']:
58             print help_message
59             sys.exit()
60         elif opt in ['-f', '--file']:
61             inputFile = open(arg, 'r')
62         elif opt == '--mod':
63             mod2Output = True
64         elif opt in ['-c', '--collect']:
65             collect = True
66         elif opt in ['-s', '--subs']:

```

```

66         substitute = True
67         elif opt in ['-m', '--merge']:
68             merge = True
69
70
71 def main(argv):
72     """
73     Main program.
74
75     Reads the command line arguments.
76     Reads and processes the input data.
77     Prints the result to STDOUT.
78
79     :param argv: the command line arguments, list of strings
80     :return: None
81     """
82     processArguments(argv)
83     lines = processLines(inputFile.readlines())
84
85     commands, formulas = linesToCommandsAndFormulas(lines)
86
87     substitutions = None
88     if substitute:
89         try:
90             substitutionFormulas = next(f for f in formulas if f.name == "
substitutions")
91         except StopIteration:
92             raise ValueError('Cannot substitute. \'formulas(substitutions)
.\' does not exists')
93         substitutions = createSubstitutions(substitutionFormulas.formulaList
)
94         formulas.remove(substitutionFormulas)
95
96         formulas = map(lambda f: formulasToExpressionTree(f, simplify=True,
collect=collect, substitutions=substitutions, merge=merge),
97                        formulas)
98
99     printCommandsAndFormulas(commands, formulas, mod2Output=mod2Output)
100
101
102 if __name__ == '__main__':
103     try:
104         main(sys.argv[1:])
105     except ValueError as e:
106         sys.stderr.write('Error: ' + str(e))
107         sys.exit(1)

```

