Department of Computer Science
Technical University of Cluj-Napoca

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

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 $O(2 * b^{d/2}) = O(b^{d/2}) << O(b^d)$, 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.

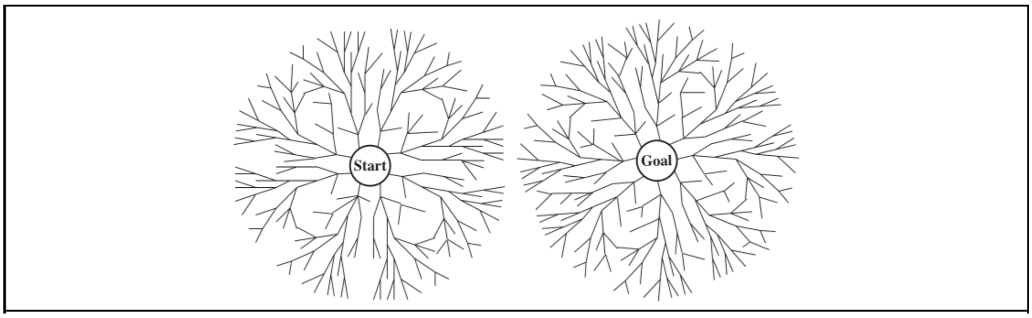

Figure 1.1: Visualization of the bidirectional search algorithm. [4]

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 that the other one, thus increasing the speed.

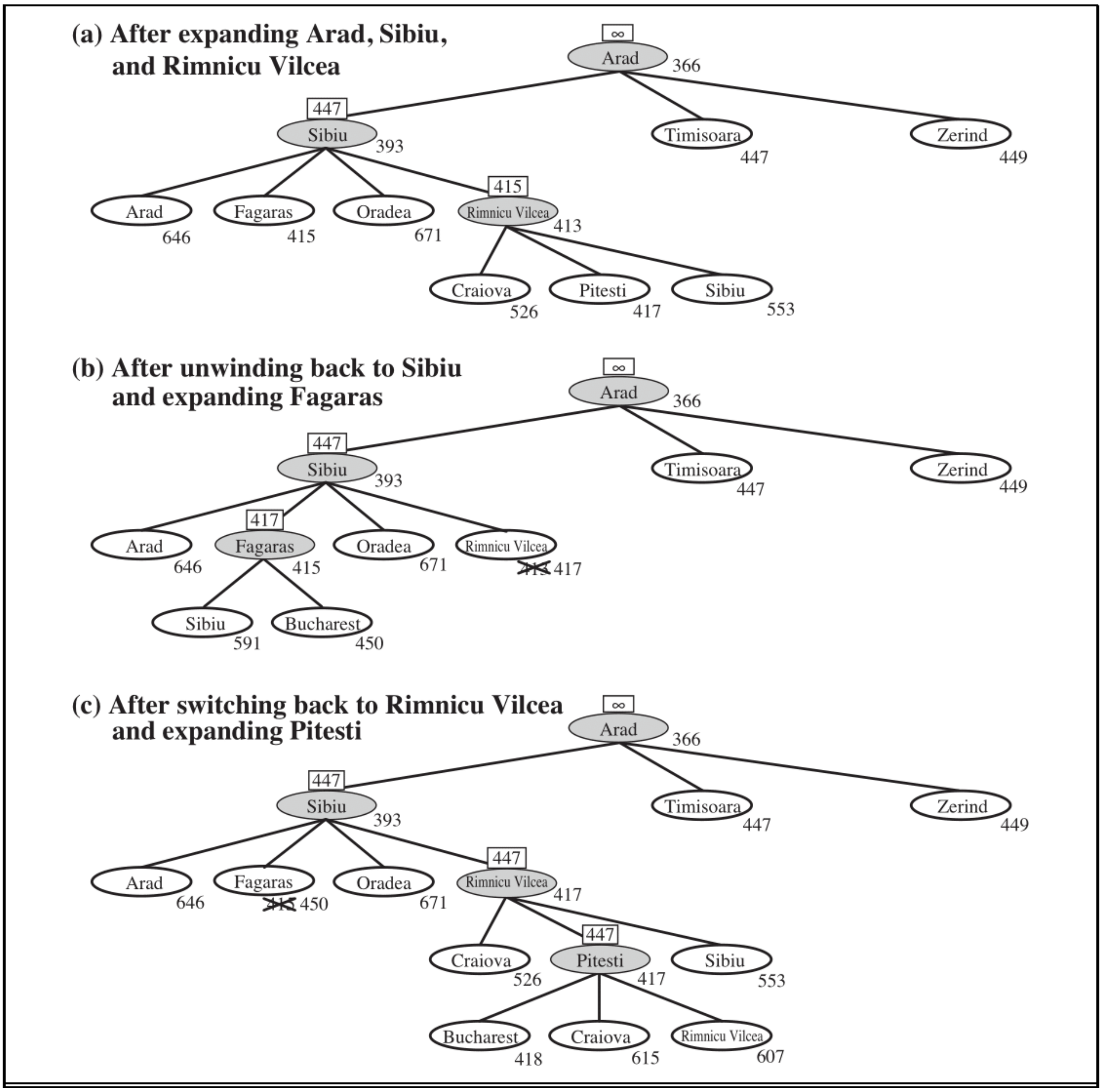

Figure 1.2: Finding a path from Arad to Bucharest by using recursive best-first search. The number above a node is its f-limit, and the number on the right of it is its f-value. The searching branch was switched two times. [4]

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". $[\![P]\!]$ represents the truth value of the proposition P.

- $[\![P]\!] = 1 \iff P$ is true
- $[\![P]\!] = 0 \iff P$ 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 $p = [\![P]\!]$ and a has the truth value of that information about A, e.g. $a = [\![\text{“}A \text{ is a knight”}]\!]$ 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:

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

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.

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

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

- $ri = [[$"There is a lady in room i"$]] = 1 + [[$"There is a tiger in room i"$]]$
- $mi = [[$"Message on the door of room i"$]]$

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 <-> 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 <-> (l1 | l2) & (t1 | 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 <-> -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 form 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 <-> (l1 | 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 <-> 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 <-> 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 <-> t1 | 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 <-> 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 <-> m2.
```

Modular arithmetics

```
m1 = m2.
```

Resolution

The system of equations representing the knowledge base:

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

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 <-> l1 & l2.
```

Modular arithmetics

```
m1 = r1 * r2.
```

Propositional logic

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

Modular arithmetics

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

Resolution

The system of equations representing the knowledge base:

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

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 <-> (l1 & l2) | (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 <-> 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 -> m1. t1 -> -m1.
l2 -> -m2. t2 -> m2.
```

Modular arithmetics

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

Resolution

The system of equations representing the knowledge base:

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

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 <-> (l1 & t2) | (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 <-> 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 -> m1. t1 -> -m1.
l2 -> -m2. t2 -> m2.
```

Modular arithmetics

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

Resolution

The system of equations representing the knowledge base:

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

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:

mij: 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

```
m11 <-> t1.
```

Modular arithmetics

```
m11 = r1 + 1
```

If the message is on the second door:

Propositional logic

```
m12 <-> t2.
```

Modular arithmetics

```
m12 = r2 + 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

```
m21 <-> t1 & t2.
m22 <-> m21.
```

Modular arithmetics

```
m21 = (r1 + 1) * (r2 + 1).
m22 = m21.
```

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

```
l1 -> m11 | m21. t1 -> -m11 | -m21.
l2 -> -m12 | -m22. t2 -> m12 | m22.
```

Modular arithmetics

```
(r2 + 1)*(m12 * m22 + m12 + m22) +
    r2 = 1.
r2 * ((m12 + 1) * (m22 + 1) + m12 +
```

```
    m22) + r2 + 1 = 1.
```

Resolution

The system of equations representing the knowledge base:

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

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:

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

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

```
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 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 <-> l2.
```

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 -> -m2 & -m3.
m2 -> -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 = [["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]]

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:

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

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

$$\begin{cases} l1 = 1 \\ l2 = 0 \\ l3 = 0 \\ t1 = 0 \\ t2 = 1 \\ t3 = 0 \\ e1 = 0 \\ e2 = 0 \\ e1 = 1 \end{cases}$$

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

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

Command object creation

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

Mace4 formulas

```
formulas(assumptions).
    l1 -> -t1.
    l2 -> -t2.
end_of_list.
```

Formulas object creation

```
Formulas(
    name = "assumptions",
    formula_list = [
        "l1 -> -t1",
        "l2 -> -t2"
    ])
```

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

Formulas class variables

```
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 ($\rightarrow$)	`->`	1
Biconditional ($\leftrightarrow$)	`<->`	1
Inclusive disjunction ($\vee$)	`|`	2
Conjunction ($\wedge$)	`&`	3
Negation ($\neg$)	`-`	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 form 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, teh 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

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

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

formulas(assumptions).

    % 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.
    % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant

    % No tiger in the room where the lady stays
    l1 -> -t1.
    l2 -> -t2.

    % Message on door #1: In this room there is a lady, and in the other
   room there is a tiger.
    m1 <-> l1 & t2.

    % Message on door #2: In one of these rooms there is a lady, and in one
   of these rooms there is a tiger.
    m2 <-> (l1 | l2) & (t1 | t2).

```

```
    % One of the messages is true, but the other one is false.
    m1 <-> -m2.
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

```
$ python translator.py -f input.txt
assign(max_models, -1).
set(arithmetic).
assign(domain_size, 2).

formulas(assumptions).
        l1*t1 + 1 = 1.
        l2*t2 + 1 = 1.
        l1*t2 + m1 + 1 = 1.
        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.
        m1 + m2 = 1.
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

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

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

formulas(substitutions).
    t1 <-> -l1.
    t2 <-> -l2.
end_of_list.

formulas(assumptions).

```

```
    % 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.
    % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant

    % No tiger in the room where the lady stays
    l1 -> -t1.
    l2 -> -t2.

    % Message on door #1: In this room there is a lady, and in the other room there is a tiger.
    m1 <-> l1 & t2.

    % Message on door #2: In one of these rooms there is a lady, and in one of these rooms there is a tiger.
    m2 <-> (l1 | l2) & (t1 | t2).

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

```
$ python translator.py -s -f input.txt
assign(max_models, -1).
set(arithmetic).
assign(domain_size, 2).

formulas(assumptions).
        1 = 1.
        1 = 1.
        l1*l2 + l1 + m1 + 1 = 1.
        l1 + l2 + m2 + 1 = 1.
        m1 + m2 = 1.
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

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

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

formulas(assumptions).
        1 = 1.
        1 = 1.
        l1*(l2 + 1) + m1 + 1 = 1.
        l1 + l2 + m2 + 1 = 1.
        m1 + m2 = 1.
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

```
$ python translator.py -s -c -m -f input.txt
assign(max_models, -1).
set(arithmetic).
assign(domain_size, 2).

formulas(assumptions).
        m2*(l1*l2*m1 + l1*l2 + l2*m1 + l2) = 1.
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

```
$python translator.py -s -c -m --mod -f input.txt
assign(max_models, -1).
set(arithmetic).
assign(domain_size, 2).

formulas(assumptions).
        (m2*(l1*l2*m1 + l1*l2 + l2*m1 + l2)) mod 2 = 1.
```

```
end_of_list.
```

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

```
$python translator.py -s -c -m --mod -f input.txt | mace4 | interpformat

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

------ process 55484 exit (all_models) ------
interpretation( 2, [number = 1,seconds = 0], [
    function(l1, [0]),
    function(l2, [1]),
    function(m1, [0]),
    function(m2, [1])]).
```

Chapter 3

A3: Planning

3.1 Introduction

The purpose of this chapter is to present the problem of cats and mice which can be solved using planning. Furthermore, descriptions of the domains and problems are presented, with gradually increasing complexity.

The "Fast Downward" planner was used for verifying the correct behavior of the description. It is assumed that the executable file is pointed by the PATH variable, and that it is called `fd`.

3.2 Problem specification

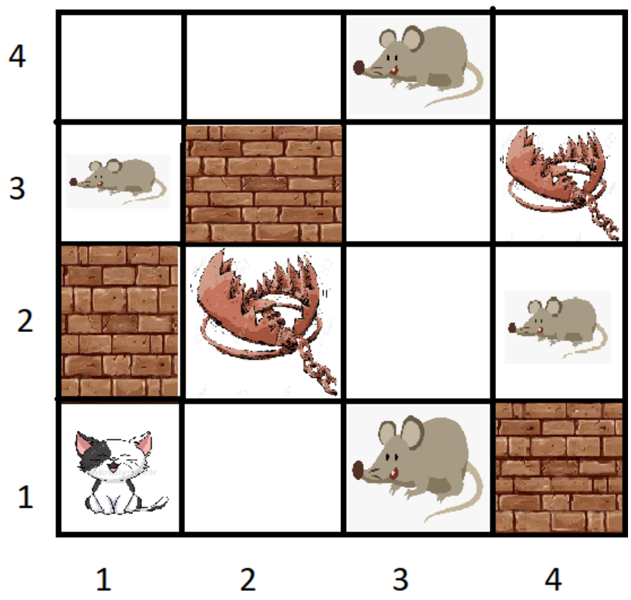

Figure 3.1: Example instance of the cat-mouse world

The world is represented by 16 rooms (4×4) or extended to 36 rooms (6x6). Each room is connected to others through walkways (no rooms are connected diagonally). The knowledge-based agent is a cat which starts from room (1, 1). The goal of the cat is to get rid of the mice: by eating them or by dropping them on traps. After each mouse the cat receives a reward. The walls block the path of the cat, forcing it to go around. The trap kills the animal that lands on it, however it can be used only once. Figure 3.1 shows an example instance of the problem.

3.3 PDDL definition

3.3.1 Catching mice

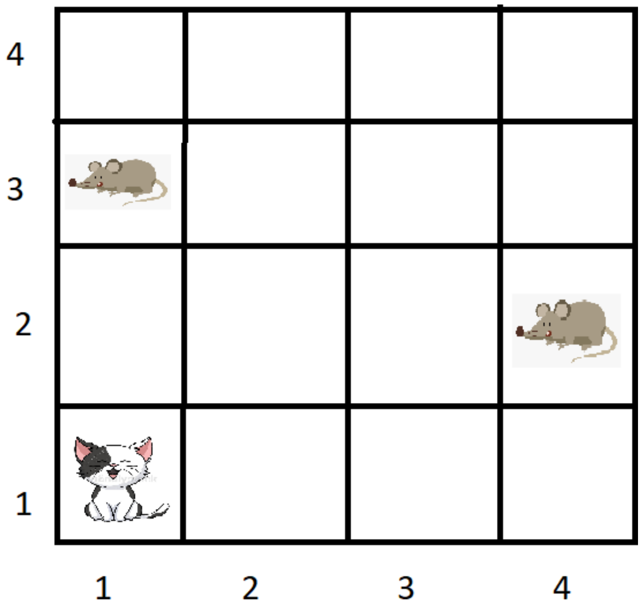

Figure 3.2: Problem 1: Catching two mice

The first step to solve the problem is to define the two fundamental actions the cat can take: moving to neighboring rooms and eating a mouse.

Domain

The first types of the domain represent the type of elements that can be inside a room: cat and mouse. The square type represents a room, identified by the row and column indices.

The `at` predicate shows whether an object is inside a given room or not. The `adj` predicate indicates whether two squares are adjacent or not. Finally the `dead` predicate shows whether the given mouse is dead or not.

Listing 3.1: Domain: types and predicates

```
(define (domain cat)
    (:requirements :strips)
    (:types cat mouse square)
    (:predicates
        (at ?who ?where)
        (adj ?s1 ?s2 - square)
        (dead ?m)
    )
```

To move from one square to another, the two squares must be adjacents to each other, and the cat should be inside the first one. Moving means to leave the current square and to enter the next one.

Listing 3.2: Domain: the move action

```
    (:action move
        :parameters (?c - cat ?s1 - square ?s2 - square)
        :precondition (and
            (adj ?s1 ?s2)
            (at ?c ?s1)
```

```
        )
        :effect (and
            (not (at ?c ?s1))
            (at ?c ?s2)
        )
    )
```

The cat can eat a mouse when they are both in the same square. Eating a mouse means to remove it from the world and mark it dead.

Listing 3.3: Domain: the eat action

```
    (:action eat
        :parameters (?c - cat ?s - square ?m - mouse)
        :precondition (and
            (at ?m ?s)
            (at ?c ?s)
        )
        :effect (and
            (not (at ?m ?s))
            (dead ?m)
        )
    )
```

Problem

The first problem is to catch two mice. There are no constraints besides the rule of moving only to adjacent cells. The initial state can be seen on figure 3.2.

The objects that appear in the problem are: a cat, two mice and 16 squares.

Listing 3.4: Problem: objects

```
(define (problem cat-p01)
    (:domain cat)
    (:objects
        c - cat
        m1 m2 - mouse
        s11 s12 s13 s14
        s21 s22 s23 s24
        s31 s32 s33 s34
        s41 s42 s43 s44
        - square
    )
```

To describe the initial state, first we must define the adjacency relationship between the squares.

Listing 3.5: Problem: adjacency relationships

```
    (:init
        ; square adjacency
        (adj s11 s12) (adj s12 s11)
        (adj s11 s21) (adj s21 s11)
        (adj s12 s13) (adj s13 s12)
        (adj s12 s22) (adj s22 s12)
        (adj s13 s14) (adj s14 s13)
        (adj s13 s23) (adj s23 s13)
        (adj s14 s24) (adj s24 s14)
        (adj s21 s22) (adj s22 s21)
        (adj s21 s31) (adj s31 s21)
        (adj s22 s23) (adj s23 s22)
```

```
        (adj s22 s32) (adj s32 s22)
        (adj s23 s24) (adj s24 s23)
        (adj s23 s33) (adj s33 s23)
        (adj s24 s34) (adj s34 s24)
        (adj s31 s32) (adj s32 s31)
        (adj s31 s41) (adj s41 s31)
        (adj s32 s33) (adj s33 s32)
        (adj s32 s42) (adj s42 s32)
        (adj s33 s34) (adj s34 s33)
        (adj s33 s43) (adj s43 s33)
        (adj s34 s44) (adj s44 s34)
        (adj s41 s42) (adj s42 s41)
        (adj s42 s43) (adj s43 s42)
        (adj s43 s44) (adj s44 s43)
```

Then the initial locations of the cat and the mice are specified:

Listing 3.6: Problem: initial locations

```
        ; cat
        (at c s11)

        ; mice
        (at m1 s31)
        (at m2 s24)
```

Finally the goal is defined: to get rid of all the mice.

Listing 3.7: Problem: goal

```
    (:goal (and (dead m1) (dead m2)))
```

Result

One should type the following line to run the example:

```
fd cat-eat.pddl cat-p01.pddl --heuristic "h=ff( )" --search "astar(h)"
```

The program prints the following solution:

Listing 3.8: Output plan

```
(move c s11 s21)
(move c s21 s31)
(eat c s31 m1)
(move c s31 s21)
(move c s21 s22)
(move c s22 s23)
(move c s23 s24)
(eat c s24 m2)
; cost = 8 (unit cost)
```

Figure 3.3 visualizes the resulted plan. The cat goes straight to the closest mouse, eats it, than moves on the shortest path to the second one and eats it too.

3.3.2 Adding walls

3.3.3 Disabling traps

For making the problems harder, we created a bigger field, containing 36 fields. We also introduced the ability to disable traps.

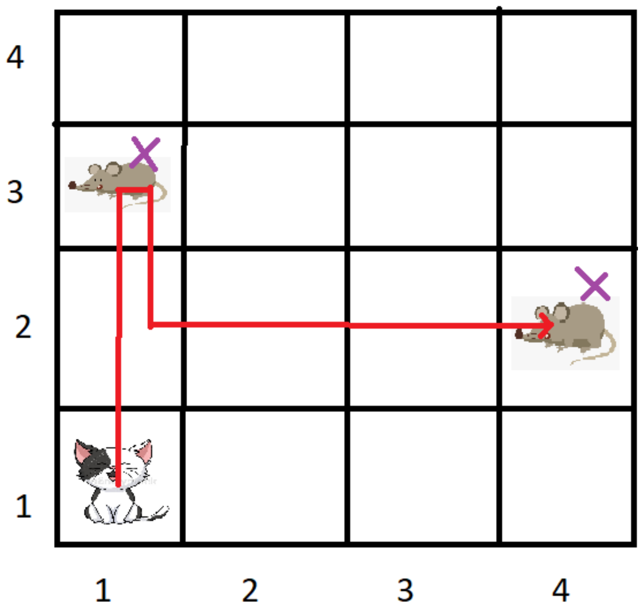

Figure 3.3: Problem 1: The resulted plan. The red line shows the trajectory of the cat. The purple 'x' marks that the cat ate the mouse

From now on, we introduce a new domain, which contains some additional actions: grabbing a mouse and disabling the trap with it. In order to pass through a trap, the cat has to grab a mouse instead of eating it, and throw it in the trap this way, the trap will not act on the cat, so it is as if it does not exist.

The grab and disable actions were defined as follows:

```
(:action grab
        :parameters (?c - cat ?s - square ?m - mouse)
        :precondition (and
            (not (dead ?c))
            (at ?m ?s)
            (at ?c ?s)
            (not (exists (?m1 - mouse) (has ?c ?m1)))
        )
        :effect (and
            (not (at ?m ?s))
            (has ?c ?m)
        )
    )
```

```
(:action disable-trap
        :parameters (?c - cat ?s1 - square ?s2 - square ?m - mouse ?t - trap
    )
        :precondition (and
            (not (dead ?c))
            (adj ?s1 ?s2)
            (at ?c ?s1)
            (at ?t ?s2)
            (has ?c ?m)
        )
        :effect (and
            (not (at ?t ?s2))
            (dead ?m)
            (not (has ?c ?m))
        )
    )
```

First example

In this example we have 8 mice, 5 traps and 5 walls. Here the cat is forced only once to disable a trap. When passing past the mouse at (5,1) the cat is forced to grab the mouse, to disable the trap at (6,1) to reach the mouse at (6,2).

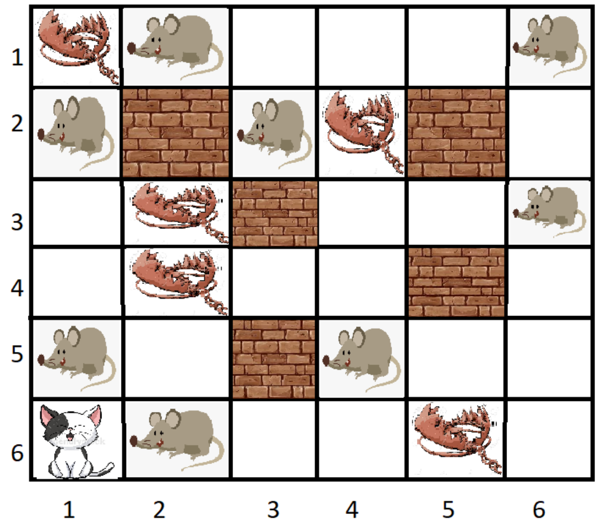

Figure 3.4: Problem 9: Catching eight mice, disabling traps, avoid walls

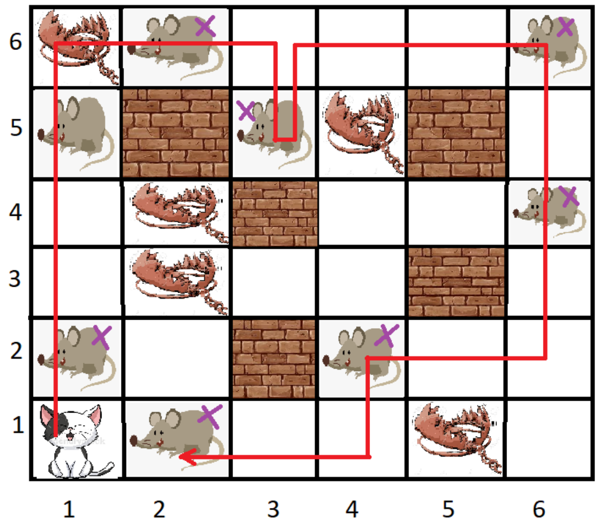

Figure 3.5: Problem 9: Catching five mice, disabling traps, avoid walls

Second example

The solution can be found below, where the mouses marked with a purple X were eaten by the cat, the other one were 'thrown' into the trap, so the cat could pass through it:

As we can see, in the first step, the cat having no other option, it had to grab the mouse at (2,1) and throw it into the trap at (2,2) so that it could go forward to eat the other mice.

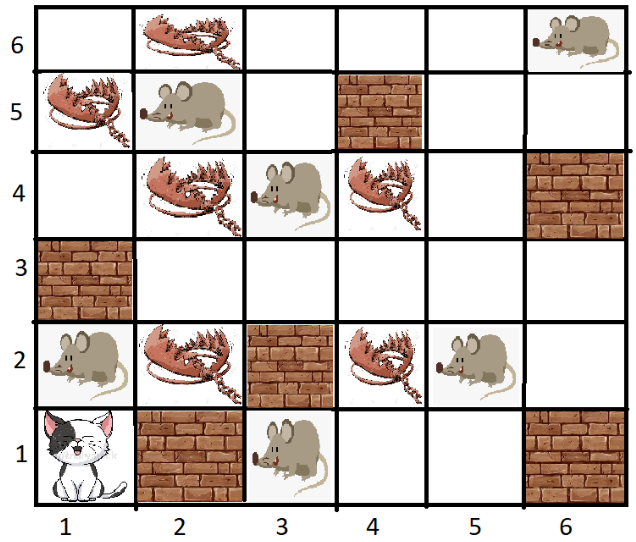

Figure 3.6: Problem 9: Catching five mice, disabling traps, avoid walls

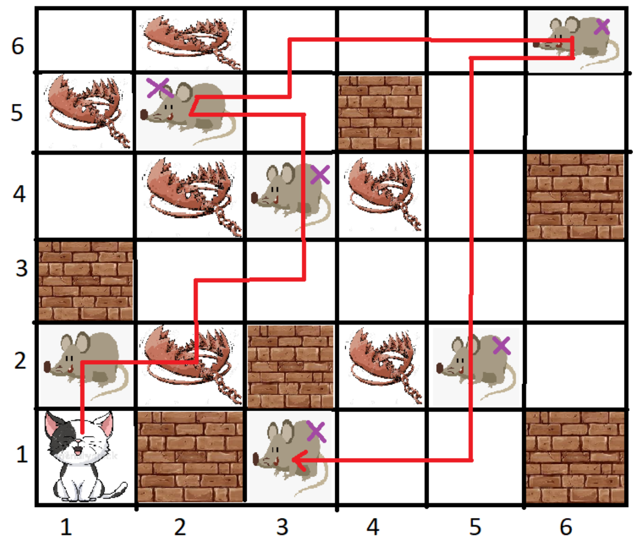

Figure 3.7: Problem 9 solution

Third example

In this example, the cat is trapped in the first row, so it is forced to throw a mouse in one of the traps from the second row. Similarly in the sixth row, the mouse is place between two traps and a wall, so the cat is forced to disable one of the traps, using a mouse. Below we can see the found solution:

Third example with actions having costs

We also implemented the second examples with actions that have costs, to force the fast downward planner to search for the optimal solution. The costs of the actions were implemented as follows:

```
(:functions
        (total-cost)
    )

    (:action move
```

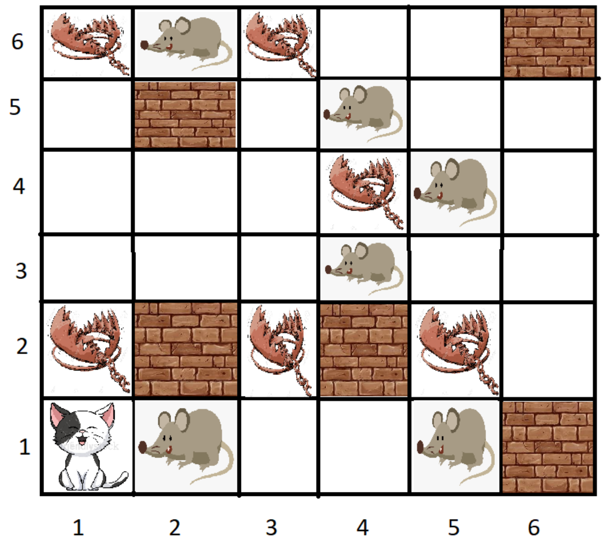

Figure 3.8: Problem 10: Catching five mice, disabling traps, avoid walls

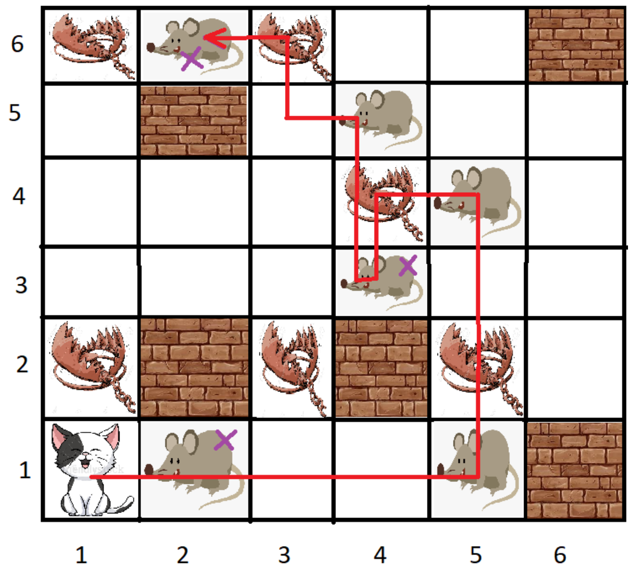

Figure 3.9: Problem 10 solution

```
        :parameters (?c - cat ?s1 - square ?s2 - square)
        :precondition (and
            (not (dead ?c))
            (adj ?s1 ?s2)
            (at ?c ?s1)
            (not (exists (?w - wall) (at ?w ?s2)))
        )
        :effect (and
            (not (at ?c ?s1))
            (when (exists (?t - trap) (at ?t ?s2)) (dead ?c))
            (when (not(exists (?t - trap) (at ?t ?s2))) (at ?c ?s2))
            (increase (total-cost) 50)
        )
    )

    (:action eat
        :parameters (?c - cat ?s - square ?m - mouse)
```

```
        :precondition (and
            (not (dead ?c))
            (at ?m ?s)
            (at ?c ?s)
            (not (exists (?m1 - mouse) (has ?c ?m1)))
        )
        :effect (and
            (not (at ?m ?s))
            (dead ?m)
            (increase (total-cost) 0)
        )
    )

    (:action grab
        :parameters (?c - cat ?s - square ?m - mouse)
        :precondition (and
            (not (dead ?c))
            (at ?m ?s)
            (at ?c ?s)
            (not (exists (?m1 - mouse) (has ?c ?m1)))
        )
        :effect (and
            (not (at ?m ?s))
            (has ?c ?m)
            (increase (total-cost) 50)
        )
    )

    (:action disable-trap
        :parameters (?c - cat ?s1 - square ?s2 - square ?m - mouse ?t - trap
    )
        :precondition (and
            (not (dead ?c))
            (adj ?s1 ?s2)
            (at ?c ?s1)
            (at ?t ?s2)
            (has ?c ?m)
        )
        :effect (and
            (not (at ?t ?s2))
            (dead ?m)
            (not (has ?c ?m))
            (increase (total-cost) 50)
        )
    )
```

Each action has a cost of 50, except for the eat action, which has 0 cost. Below we can see the solution for the weighted actions:

As it can be seen, after disabling the trap at (4,4) the cat did not eat the mouse at (3,4) but grabbed the mouse at (5,4) instead and went to disable the trap at (6,3), ate the mouse at (6,2) and only after that it went back to eat the mouse at (3,4)

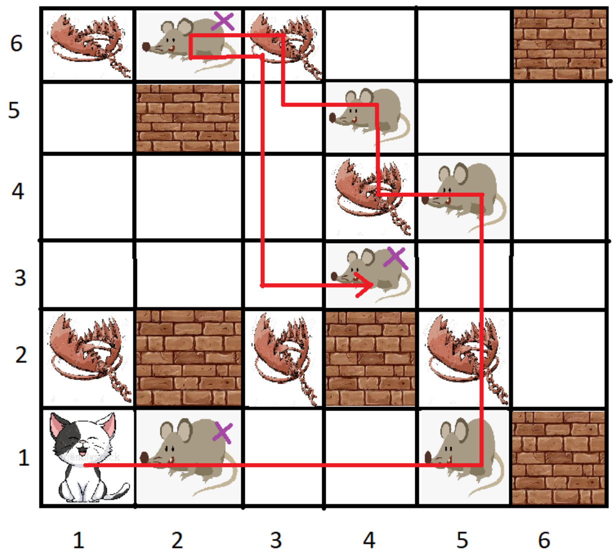

Figure 3.10: Problem 10 solution with weighted actions

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

```
class Node:
    """
    Node used for search algorithms.
    """

    def __init__(self, state, action, cost, parent, f=0):
        """
        state: (x,y) coordinates
        action: direction from Directions of game.py
        cost: cost of reaching the node
        parent: parent node
        f: the backed-up f-value
        """
        self.state = state
        self.action = action
        self.cost = cost
        self.parent = parent
        self.f = f
```

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

```
def reconstructPath(node):
    """
    Reconstructs a path, whose end node is given as a parameter, by
   iterating through the parent references.

    node: search node, instance of Node
    """

    if not node:
```

```
        return None

    path = []
    while node.parent is not None:
        path = path + [node.action]
        node = node.parent
    path.reverse()
    return path
```

A.2 Bidirectional Search

```
def bidirectionalSearch(problem):
    """
    Simultaneously searches from both the start and the goal positions.
    Stops when the two frontiers intersect.
    The shallowest node is taken first (BFS).
    """

    forward = util.Queue()
    start = problem.getStartState()
    exploredForward = {start}
    forward.push((start, []))

    backward = util.Queue()
    goal = problem.goal
    exploredBackward = {goal}
    backward.push((goal, []))

    visitedForward = set()
    visitedBackwards = set()

    while not forward.isEmpty() and not backward.isEmpty():

        # Forward searching
        currentNode, currentActions = forward.pop()

        if currentNode not in visitedForward:
            visitedForward.add(currentNode)
            if currentNode in exploredBackward:
                while not backward.isEmpty():
                    node, actions = backward.pop()
                    if node == currentNode:
                        solution = currentActions + actions.reverse()
                        return solution

            for (childState, childAction, childCost) in problem.getSuccessors(currentNode):
                forward.push((childState, currentActions + [childAction]))
                exploredForward.add(childState)

        # Backward searching
        currentNode, currentActions = backward.pop()

        if currentNode not in visitedBackwards:

            visitedBackwards.add(currentNode)
            if currentNode in exploredForward:

```

```
                while not forward.isEmpty():
                    node, actions = forward.pop()
                    if node == currentNode:
                        backwardActions = [reverseAction(action) for action in currentActions]
                        backwardActions.reverse()
                        solution = actions + backwardActions
                        return solution

            for (childState, childAction, childCost) in problem.getSuccessors(currentNode):
                backward.push((childState, currentActions + [childAction]))
                exploredBackward.add(childState)
    return None


def reverseAction(action):
    """Changes the action to its inverse."""

    if action == 'North':
        return 'South'
    elif action == 'South':
        return 'North'
    elif action == 'West':
        return 'East'
    else:
        return 'West'
```

A.3 Depth Limited Search

```
def depthLimitedSearch(problem, limit):
    """
    Search the deepest nodes in the search tree first. The search depth is limited by the given parameter.

    returns a (node, cutoff) tuple:
    - node is the goal node containing reference to its parent.
    - cutoff is True if no solution was found in the given limit, False otherwise.
    """

    start_node = Node(problem.getStartState(), None, 0, None)
    visited = {start_node.state}
    return recursiveDLS(problem, visited, start_node, limit)


def recursiveDLS(problem, visited, node, limit):
    """
    Helper function for depthLimitedSearch(problem, limit).

    returns a (node, cutoff) tuple:
    - node: the goal node containing reference to its parent, or False if no solution was found.
    - cutoff: True if no solution was found in the given limit, False otherwise.
    """

    if problem.isGoalState(node.state):
```

```
        return node, False
    if limit == 0:
        return None, True

    cutoff_occurred = False
    for child_state, child_action, child_cost in problem.getSuccessors(node.state):
        if child_state not in visited:

            child = Node(child_state, child_action, child_cost, node)

            visited.add(child_state)
            (result, cutoff) = recursiveDLS(problem, visited, child, limit - 1)
            visited.remove(child_state)

            if cutoff:
                cutoff_occurred = True
            elif result:
                return result, False

    if cutoff_occurred:
        return None, True
    else:
        return None, False
```

A.4 Iterative Deepening Depth-First Search

.

```
def iterativeDeepeningSearch(problem):
    """
    Search the deepest nodes in the search tree first. The search depth is limited, but the limit is increased in each
    iteration.

    returns:
    - node: the goal node containing reference to its parent, or False if no solution was found.
    """

    depth = 0
    while True:
        (result, cutoff) = depthLimitedSearch(problem, depth)
        if not cutoff:
            return reconstructPath(result)
        depth += 1
```

A.5 Iterative Deepening A*

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

```
    start = problem.getStartState()
    bound = heuristic(start, problem)
    path = [(start, None, 0)]
    visited = {start}

    while True:
        t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound)

        if t is None:
            return [action for (state, action, cost) in path if action is not None]
        if t == sys.maxint:
            return None

        bound = t


def iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound):
    """
    Utility function for iterativeDeepeningAStarSearch.
    Returns the  minimum cost of all values that exceeded the current bound.
    """
    state, action, cost = path[-1]
    f = cost + heuristic(state, problem)

    if f > bound:
        return f
    if problem.isGoalState(state):
        return None

    min = sys.maxint

    for child_state, child_action, child_cost in problem.getSuccessors(state):
        if child_state not in visited:
            visited.add(child_state)
            path.append((child_state, child_action, child_cost + cost))

            t = iterativeDeepeningAStarSearchUtil(problem, heuristic, path, visited, bound)

            if t is None:
                return None
            if t < min:
                min = t

            path.pop()
            visited.remove(child_state)

    return min
```

A.6 Recursive Best-First Search

```
def recursiveBestFirstSearch(problem, heuristic=nullHeuristic):
    """
```

```
    Similarly to the DFS, search the deepest nodes in the search tree first, but uses the f_limit variable to keep
    track of the f values. The f value is the largest reached g(n) + h(n) value upon one path after the search is
    stopped because of the f_limit. This f values is backed up to the parent of a node upon backtracking. The f_limit of
    the best child path is the f value of the best alternative path.

    returns:
    - node: the goal node containing reference to its parent, or False if no solution was found.
    """
    start_node = Node(problem.getStartState(), None, 0, None)
    start_node.f = 0
    visited = {start_node.state}
    result, _ = RBFS(problem, heuristic, visited, start_node, float('inf'))

    return reconstructPath(result)


def RBFS(problem, heuristic, visited, node, f_limit):
    """
    Helper function of recursiveBestFirstSearch(problem, heuristic).

    returns:
    - node: the goal node containing reference to its parent, or None if no solution was found.
    - f-cost: the f value obtained on the path.
    """

    if problem.isGoalState(node.state):
        return node, None

    successors = util.PriorityQueueWithFunction(lambda n: n.f)

    for child_state, child_action, child_cost in problem.getSuccessors(node.state):
        if child_state not in visited:
            path_cost = child_cost + node.cost
            child_f = max(path_cost + heuristic(child_state, problem), node.f)
            child = Node(child_state, child_action, path_cost, node, child_f)
            successors.push(child)

    if successors.isEmpty():
        return None, float('inf')

    while True:
        best = successors.pop()

        if best.f > f_limit:
            return None, best.f

        if successors.isEmpty():
            alternative_f = float('inf')
        else:
            alternative = successors.pop()
            alternative_f = alternative.f
            successors.push(alternative)
```

```
        visited.add(best.state)
        result, best.f = RBFS(problem, heuristic, visited, best, min(f_limit
    , alternative_f))
        visited.remove(best.state)

        if result:
            return result, best.f

        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`

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

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

formulas(assumptions).

    % 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.
    % l1 & l2 | l1 & t2 | t1 & l2 | t1 & t2. % redundant

    % No tiger in the room where the lady stays
    l1 -> -t1.
    l2 -> -t2.

    % Message on door #1: In this room there is a lady, and in the other room there is a tiger.
    m1 <-> l1 & t2.

    % Message on door #2: In one of these rooms there is a lady, and in one of these rooms there is a tiger.
    m2 <-> (l1 | l2) & (t1 | t2).

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

end_of_list.
```

B.1.2 Modular arithmetic

Listing B.2: `arithmetic/1.in`

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

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

formulas(assumptions).

    % Message on door #1: In this room there is a lady, and in the other
   room there is a tiger.
    m1 = r1 * (r2 + 1).

    % Message on door #2: In one of these rooms there is a lady, and in one
   of these rooms there is a tiger.
    m2 = r1 + r2.

    % One of the messages is true, but the other one is false.
    m1 + m2 = 1.

end_of_list.
```

B.2 The Second Trial

B.2.1 Propositional logic

Listing B.3: `propositional/2.in`

```

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%At least one of these room contains a lady
m1 <-> l1 | l2.
```

```

%A tiger is in the other room
m2 <-> t1.

%The messages are either both true or both false
m1 <-> m2.

end_of_list.
```

B.2.2 Modular arithmetic

Listing B.4: `arithmetic/2.in`

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

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

formulas(assumptions).

    % Message on door #1: At least one of these rooms contains a lady
    m1 = r1 * r2 + r1 + r2.

    % Message on door #2: A tiger is in the other room
    m2 = r1 + 1.

    % Either both are true or both are false
    % (m1 + m2) mod 2 = 0.
    m1 = m2.

end_of_list.
```

B.3 The Third Trial

B.3.1 Propositional logic

Listing B.5: `propositional/3.in`

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

```

```
%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%Either a tiger is in this room, or a lady is in the other room
m1 <-> t1 | l2.

%A lady is in the other room
m2 <-> l1.

%The messages are either both true or both false
(-m1 & -m2) | (m1&m2).

end_of_list.
```

B.3.2 Modular arithmetic

Listing B.6: `arithmetic/3.in`

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

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

formulas(assumptions).

    % Either a tiger is in this room or a lady is in the other room
    % m1 = ((r1 + 1) * r2 + (r1 + 1) + r2) mod 2.
    % 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"
    m1 = (r1 * (r2 + 1) + 1) mod 2.

    % A lady is in the other room
    m2 = r1.

    % Either both are true or both are false
    m1 = m2.

end_of_list.
```

B.4 The Fourth Trial

B.4.1 Propositional logic

Listing B.7: `propositional/4.in`

```
set(arithmetic).
```

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%Both rooms contains ladies
m1 <-> l1 & l2.

%Both rooms contains ladies
m2 <-> m1.

%If lady in first room, message is true. If tiger in first room, message is false
l1 -> m1.
t1 -> -m1.

%If tiger in second room, message is true, if lady in second room, message is false
l2 -> -m2.
t2 -> m2.

end_of_list.
```

B.4.2 Modular arithmetic

Listing B.8: `arithmetic/4.in`

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

formulas(assumptions).

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

%Both rooms contain ladies
m1 = r1 * r2.

%Both rooms contain ladies
m2 = m1.
```

```
%If lady in first room, message is true
r1 = m1.


%If lady in second room, message is false
r2 = m2 + 1.

end_of_list.
```

B.5 The Fifth Trial

B.5.1 Propositional logic

Listing B.9: `propositional/5.in`

```

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%At least one room contains a lady
m1 <-> l1 | l2.

%The other room contains a lady
m2 <-> l1.

%If lady in first room, message is true. If tiger in first room, message is
    false
l1 -> m1.
t1 -> -m1.

%If tiger in second room, message is true, if lady in second room, message
    is false
l2 -> -m2.
t2 -> m2.

end_of_list.
```

B.5.2 Modular arithmetic

Listing B.10: `arithmetic/5.in`

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

formulas(assumptions).

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

%At least one room contains a lady
m1 = r1 * r2 + r1 + r2.

%The other room contains a lady
m2 = r1.



%If lady in first room, message is true
r1 = m1.


%If lady in second room, message is false
r2 = (m2 + 1) mod 2.

end_of_list.
```

B.6 The Sixth Trial

B.6.1 Propositional logic

Listing B.11: `propositional/6.in`

```

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
```

```
l2 -> -t2.

%It makes no difference which room you pick
m1 <-> (l1 & l2) | (t1 & t2).

%There is a lady in the other room
m2 <-> l1.

%If lady in first room, message is true. If tiger in first room, message is
    false
l1 -> m1.
t1 -> -m1.

%If tiger in second room, message is true, if lady in second room, message
    is false
l2 -> -m2.
t2 -> m2.

end_of_list.
```

B.6.2 Modular arithmetic

Listing B.12: `arithmetic/6.in`

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

formulas(assumptions).
% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

%It makes no difference which room you pick
m1 = (r1 + r2 + 1) mod 2.

%The other room contains a lady
m2 = r1.


%If lady in first room, message is true
r1 = m1.


%If lady in second room, message is false
r2 = (m2 + 1) mod 2.

end_of_list.
```

B.7 The Seventh Trial

B.7.1 Propositional logic

Listing B.13: `propositional/7.in`

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% m1: Message on the door of room 1
% m2: Message on the door of room 2

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%It does make a difference which room you pick
m1 <-> (l1 & t2) | (l2 & t1).

%You are better off choosing the other room
m2 <-> l1.

%If lady in first room, message is true. If tiger in first room, message is false
l1 -> m1.
t1 -> -m1.

%If tiger in second room, message is true, if lady in second room, message is false
l2 -> -m2.
t2 -> m2.

end_of_list.
```

B.7.2 Modular arithmetic

Listing B.14: `arithmetic/7.in`

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

formulas(assumptions).
% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]



%It does make a difference which room you pick
m1 = (r1 + r2) mod 2.
```

```
%You are better off choosing the other room
m2 = r1.


%If lady in first room, message is true
r1 = m1.


%If lady in second room, message is false
r2 = (m2 + 1) mod 2.

end_of_list.
```

B.8 The Eighth Trial

B.8.1 Propositional logic

Listing B.15: propositional/8.in

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% mij: Message i on the door of room j

%Either one lady and one tiger or two tigers or two ladies
l1 & l2 | t1&t2 | t1 & l2 | t2 & l1.

%If lady in room, there is no tiger
l1 -> -t1.
l2 -> -t2.

%This rooms contains a tiger
m11 <-> t1.
m12 <-> t2.

%Both rooms contain tigers
m21 <-> t1 & t2.
m22 <->t1 & t2.


%If lady in first room, message is true. If tiger in first room, message is
    false
l1 -> m11 | m21.
t1 -> -m11 | -m21.

%If tiger in second room, message is true, if lady in second room, message
    is false
l2 -> -m12 | -m22.
t2 -> m12 | m22.

```

```
end_of_list.
```

B.8.2 Modular arithmetic

Listing B.16: `arithmetic/8.in`

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

formulas(assumptions).
% mij = [["Message i is on the door of room j and it is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

%This room contains a tiger
m11 = (r1 + 1) mod 2.
m12 = (r2 + 1) mod 2.

%Both rooms contain a tiger
m21 = ((r1 + 1) * (r2 + 1)) mod 2.
m22 = m21.


%If lady in first room, message is true
(r1 * ( m11 * m21 + m11 + m21) + r1 + 1) mod 2 = 1.

((r1 + 1) * ((m11 + 1) * ( m21 + 1) + m11 + m21) + r1) mod 2 = 1.

%If lady in second room, message is false
((r2 + 1)*(m12 * m22 + m12 + m22) + r2)mod 2= 1.

(r2 * ((m12 + 1) * (m22 + 1) + m12 + m22) + r2 + 1) mod 2 = 1.

end_of_list.
```

B.9 The Ninth Trial

B.9.1 Propositional logic

Listing B.17: `propositional/9.in`

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% l3: There is a lady in room 3
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% t3: There is a tiger in room 3
% m1: Message on the door of room 1
```

```
% m2: Message on the door of room 2
% m3: Message on the door of room 3

 %One lady and two tigers
 l1 & t2 & t3 | l2 & t1 & t3 | l3 & t1 & t2.

 %no lady in more than 1 room; the princess is unique
 l1 -> -l2.
 l1 -> -l3.
 l2 -> -l1.
 l2 -> -l3.
 l3 -> -l1.
 l3 -> -l2.


 %no tiger in the room where the lady stays
 l1 -> -t1.
 l2 -> -t2.
 l3 -> -t3.

%A tiger is in this room
 m1 <-> t1.

%A lady is in this room
 m2 <-> l2.

%A tiger is in room two
 m3 <-> t2.

%At most one sign is true
m1 -> -m2 & -m3.
m2 -> -m3.

end_of_list.
```

B.9.2 Modular arithmetic

Listing B.18: arithmetic/9.in

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

formulas(assumptions).
% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% m3 = [["The message on the door of room 3 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]
% r3 = [["There is a lady in room 3"]]
%    = 1 + [["There is a tiger in room 3"]]

%Lady in one room and tigers in the other two rooms

% One lady and two tigers
r1 * r2 * r3  = 0.
r1 + r2 + r3  = 1.
```

```

%Tiger in this room
m1 = (r1 + 1) mod 2.

%Lady in this room
m2 = r2.


%Tiger in room 2
m3 = (r2 + 1)  mod 2.


%At most one of the three signs is true
m1 * m2 + m1 * m3 + m2 * m3 + 1 = 1.

end_of_list.
```

B.10 The Tenth Trial

B.10.1 Propositional logic

Listing B.19: `propositional/10.in`

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

formulas(assumptions).

% l1: There is a lady in room 1
% l2: There is a lady in room 2
% l3: There is a lady in room 3
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% t3: There is a tiger in room 3
% m1: Message on the door of room 1
% m2: Message on the door of room 2
% m3: Message on the door of room 3

 %One lady and two tigers
 l1 & t2 & t3 | l2 & t1 & t3 | l3 & t1 & t2.

 %no lady in more than 1 room; the princess is unique
 l1 -> -l2.
 l1 -> -l3.
 l2 -> -l1.
 l2 -> -l3.
 l3 -> -l1.
 l3 -> -l2.


 %no tiger in the room where the lady stays
 l1 -> -t1.
 l2 -> -t2.
 l3 -> -t3.

%A tiger is in room two
 m1 <-> t2.
```

```
%A tiger is in this room
 m2 <-> t2.

%A tiger is in room one
 m3 <-> t1.

%The sign on the door, where the lady is, is true
 l1 -> m1.
 l2 -> m2.
 l3 -> m3.

%From the other two signs at least one is false
m1 -> -m2 | -m3.
m2 -> -m1 | -m3.
m3 -> -m1|-m2.

end_of_list.
```

B.10.2 Modular arithmetic

Listing B.20: `arithmetic/10.in`

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

% m1 = [["The message on the door of room 1 is true"]]
% m2 = [["The message on the door of room 2 is true"]]
% r1 = [["There is a lady in room 1"]]
%    = 1 + [["There is a tiger in room 1"]]
% r2 = [["There is a lady in room 2"]]
%    = 1 + [["There is a tiger in room 2"]]

formulas(assumptions).
    % One lady and two tigers
    r1 * r2 * r3  = 0.
    r1 + r2 + r3  = 1.

    % The message on the door containing the lady is true
    (r1 * (m1 + 1)) mod 2  = 0.
    (r2 * (m2 + 1)) mod 2  = 0.
    (r3 * (m3 + 1)) mod 2  = 0.

    % At least one of the other two messages is false
    m1 * m2 * m3 = 0.
    ((m1 + 1) * (m2 + 1) * (m3 + 1)) mod 2  = 0.

    % Message on door #1: There is a tiger in room #2
    m1 = r2 + 1.

    % Message on door #2: There is a tiger here
    m2 = r2 + 1.

    % Message on door #3: There is a tiger in room #1
    m3 = (r1 + 1) mod 2 .
end_of_list.
```

B.11 First, Second, and Third Choice

B.11.1 Propositional logic

Listing B.21: propositional/11.in

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

formulas(assumptions).
% l1: There is a lady in room 1
% l2: There is a lady in room 2
% l3: There is a lady in room 3
% t1: There is a tiger in room 1
% t2: There is a tiger in room 2
% t3: There is a tiger in room 3
% e1: Room 1 is empty
% e2: Room 2 is empty
% e3: Room 3 is empty
% m1: Message on the door of room 1
% m2: Message on the door of room 2
% m3: Message on the door of room 3



 %there is a lady in room 1, 2 or 3
 l1 | l2 | l3.

 %no lady in more than 1 room; the princess is unique
 l1 -> -l2 & -l3.
 l2 -> -l1 & -l3.
 l3 -> -l1 & -l2.

 %no tiger in the room where the lady stays, room not empty if lady is in it
 l1 -> -t1 & -e1.
 l2 -> -t2 & -e2.
 l3 -> -t3 & -e3.


 t1| t2 | t1.


 %no tiger in more than 1 room
 t1 -> -t2 & -t3.
 t2 -> -t1 & -t3.
 t3 -> -t2 & -t1.

%if there is a tiger in the room it can not be empty
 t1 -> -e1.
 t2 -> -e2.
 t3 -> -e3.

 e1|e2|e3.

 %no more than one empty room
 e1 -> -e2 & -e3.
 e2 -> -e1 & -e3.
 e3 -> -e2 & -e1.

```

```
 %Room three is empty
 m1 <-> e3.

 %There is a tiger in room one.
 m2 <-> t1.

 %This room is empty
 m3 <-> e3.

 %If lady in room, the message is true
 l1 -> m1.
 l2 -> m2.
 l3 -> m3.

 %If tiger in room, the message is false
 t1 -> -m1.
 t2 -> -m2.
 t3 -> -m3.



end_of_list.
```

B.11.2 Modular arithmetic

Listing B.22: `arithmetic/11.in`

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

formulas(assumptions).

%mi = [["The message on the door of room i is true"]]
%li = [["There is a lady in room i"]]
%ei = [["Room i is empty"]]
%ti = [["There is a tiger in room i"]]


%One empty, one lady, one tiger
l1 * l2 * l3  = 0.
l1 + l2 + l3  = 1.


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


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

%One room can only contain a lady/tiger or it is empty
(l1 * ((t1 + 1) * (e1 + 1)) + l1 + 1) mod 2 = 1.
(l2 * ((t2 + 1) * (e2 + 1)) + l2 + 1) mod 2 = 1.
(l3 * ((t3 + 1) * (e3 + 1)) + l3 + 1) mod 2 = 1.


(t1 * ((e1 + 1) * (l1 + 1)) + t1 + 1) mod 2 = 1.
```

```
(t2 * ((e2 + 1) * (l2 + 1)) + t2 + 1) mod 2 = 1.
(t3 * ((e3 + 1) * (l3 + 1)) + t3 + 1) mod 2 = 1.

%Room three is empty
m1 = e3.

%There is a tiger in room one.
m2 = t1.

%This room is empty
m3 = e3.

%If lady in room, the message is true
(l1 * m1 + l1 + 1) mod 2 = 1.
(l2 * m2 + l2 + 1) mod 2 = 1.
(l3 * m3 + l3 + 1) mod 2 = 1.

%If tiger in room, the message is false
(t1 * (m1 + 1) + t1 + 1) mod 2 = 1.
(t2 * (m2 + 1) + t2 + 1) mod 2 = 1.
(t3 * (m3 + 1) + t3 + 1) mod 2 = 1.




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`

```
class Command:
    """
    Class for representing mace4 commands.
    Commands are given as: command_name(argument1[,argument2,...]).
    """
    def __init__(self, name, args):
        self.name = name
        self.args = args

    def __str__(self):
        return "Command(%s, %s)" % (self.name, self.args)

    def __repr__(self):
        return self.__str__()


class Formulas:
    """
    Class for representing mace4 formulas.
    Formulas are given as:
        formulas(name).
            predicate1.
            [predicate2.
            ...]
        end_of_list.
    """

    commandName = "formulas"
    endCommand = "end_of_list"

    def __init__(self, name, formula_list):
        self.name = name
        self.formulaList = formula_list

```

```
    def __str__(self):
        return "Formulas(%s, %s)" % (self.name, self.formulaList)

    def __repr__(self):
        return self.__str__()
```

C.2 Parsing the Mace4 input file

Listing C.2: `input.py`

```
"""
Functions for processing the input data.
"""

from commands import Command, Formulas

commentSymbol = "%"
lineSeparator = "."


def processLine(line):
    """
    Processes a single line of the input. It removes the comments and the
    leading and trailing whitespaces.

    :param line: a line of the input data
    :return: the result after processing, or None if the result is empty.
    """

    # Remove comments
    line = line.split(commentSymbol, 1)[0]

    # Remove leading and trailing whitespace
    line = line.strip()

    # Remove empty lines
    if not line:
        return None

    return line


def changeLineSeparator(lines, separator):
    """
    Given a list of lines, changes the line separator to the given one.
    Thus merges all the lines and separates them by the given separator.

    :param lines: list containing the lines of the input data
    :param separator: the desired line separator
    :return: list of strings representing the lines of the input data
    separated by the given parameter
    """

    lines = " ".join(lines).split(separator)

    if lines[-1]:
        raise ValueError("Terminating \"%s\" not found" % lineSeparator)

```

```
    return map(lambda x: x.strip(), lines[:-1])


def processLines(lines):
    """
    Processes each line of the input data.

    :param lines: list of lines of the input data
    :return: list of non-empty lines of the input data, separated by `lineSeparator`
    """

    # Process each line. Remove the empty lines.
    lines = filter(lambda x: x, map(processLine, lines))

    if not lines:
        raise ValueError("Input file is empty")

    # Change the line separator from the endline character to a different one.
    # Thus merging the lines not terminating in the new character.
    lines = changeLineSeparator(lines, lineSeparator)

    return lines


def linesToCommandsAndFormulas(lines):
    """
    Transforms the lines of data into lists of Command and Formulas objects.

    :param lines: lines of the input data
    :return: (commands, formulas) tuples obtained by parsing the lines
    """
    commands = []
    formulas = []
    formulaName = None
    propositions = []

    for line in lines:

        if formulaName:
            if line == Formulas.endCommand:
                formulas.append(Formulas(formulaName, propositions))
                formulaName = None
            else:
                propositions.append(line)
        else:
            try:
                name, rest = line.split("(", 1)
                args = rest.split(")")[0].split(",")
                args = map(lambda x: x.strip(), args)
            except ValueError:
                raise ValueError("Invalid command format: \"%s\"" % line)

            if name == Formulas.commandName:
                formulaName = args[0]
                propositions = []
            else:
                command = Command(name, args)
                commands.append(command)
```

```
    return commands, formulas
```

C.3 Creating/transforming Boolean ring expression trees

Listing C.3: `expression.py`

```
"""
Functions for
- creating sympy expression trees in mod 2 arithmetics from proposition
  logic.
- transforming sympy expression trees
"""
import re

import sympy
from sympy import Mul, Add, Symbol, Poly

from commands import Formulas

# Priorities of the propositional logic operators
priority = {
    "(": 0,
    ")": 0,
    "->": 1,
    "<->": 1,
    "|": 2,
    "&": 3,
    "-": 4
}


def is_symbol_char(char):
    """
    Verifies whether a character can be part of a symbol or not.

    :param char: character to be verified
    :return: True or False
    """
    return re.search("^[a-zA-z0-9]+$", char) is not None


def operatorStringToExpression(operator, *operands):
    """
    Given a string representing a propositional logic operator, returns a
    simpy expression which is the modular
    arithmetic translation of the operator.

    This function simplifies the expression according to the rules of
    general algebra and does not consider the rules of
    the mod 2 arithmetics.

    :param operator: string representing a propositional logic operator
    :param operands: list of simpy expressions or integers
    :return: simpy expression which results from applying the operands to
    the translated operator
    """
    if operator == "&":
```

```
        return Mul(*operands)
    elif operator == "-":
        return Add(operands[0], 1)
    elif operator == "|":
        return Add(Add(*operands), Mul(*operands))
    elif operator == "<->":
        return Add(Add(*operands), 1)
    elif operator == "->":
        return Add(Mul(*operands), Add(operands[0], 1))
    else:
        raise ValueError("Unknown operator \"%s\"" % operator)


def splitProposition(proposition):
    """
    Splits a proposition into terms of operands and operators.

    :param proposition: proposition to be split
    :return: list of strings representing the operands and operators of the
    proposition in order
    """
    terms = []
    is_symbol = is_symbol_char(proposition[0])
    term = ""

    def addTerm(t):
        t = t.strip()
        if t:
            terms.append(t)

    for char in proposition:
        is_symbol2 = is_symbol_char(char)

        if char in " ":
            addTerm(term)
            term = ""
            is_symbol = False
            continue

        if char in "()":
            addTerm(term)
            term = ""
            is_symbol = False
            terms.append(char)
            continue

        if is_symbol == is_symbol2:
            term = term + char
        else:
            addTerm(term)
            term = char

        is_symbol = is_symbol2

    if term:
        addTerm(term)

    return terms
```

```
def buildExpressionTree(proposition, substitutions=None):
    """
    Builds a sympy expression tree from the given proposition in propositional logic.

    The result is an expression (tree) in modulo 2 arithmetics. However it is simplified based only on the general rules
    of algebra. E.g. x * x will be x**2 instead of x.

    Substitutes the symbols to the simpy expressions associated with them in the substitutions dictionary.

    :param proposition: string, in propositional logic
    :param substitutions: a dictionary whose elements are (string_symbol, (expression, symbols)).
                          No substitution if None is given.
        string_symbol: the string which will be substituted
        expression: a sympy expression which substitutes the string
        symbols: the list of sympy symbols that are present in the expression
    :return: a simpy expression (tree)
    """
    operators = set(priority.keys())

    node_stack = []
    term_stack = []
    symbols = set()

    terms = splitProposition(proposition)

    def createAndPush():
        """
        Creates a new node and pushes to the node stack

        :return: None
        """
        op = term_stack.pop()

        if op == '-':
            # Unary operator
            node = operatorStringToExpression(op, node_stack.pop())
        else:
            # Binary operator
            right = node_stack.pop()
            left = node_stack.pop()
            node = operatorStringToExpression(op, left, right)

        node_stack.append(node)

    for term in terms:
        if term == '(':
            term_stack.append(term)
        elif term not in operators:
            if not substitutions or term not in substitutions:
                node = Symbol(term)
                symbols.add(node)
            else:
                node, node_symbols = substitutions[term]
                symbols.update(node_symbols)
            node_stack.append(node)
```

```
        elif priority[term] > 0:
            while term_stack \
                    and term_stack[-1] != '(' \
                    and priority[term_stack[-1]] >= priority[term]:
                createAndPush()

            term_stack.append(term)

        elif term == ')':
            while term_stack and term_stack[-1] != '(':
                createAndPush()
            term_stack.pop()

    while term_stack:
        createAndPush()

    return node_stack[-1], symbols


def simplifyExpression(expr, symbols):
    """
    Simplifies a sympy expression by applying also the rules of the boolean ring.

    1. Uses the sympy.simplify function for modulus 2 (implicitly, by creating a polynomial).
    2. Replaces all the powers with their bases. Because x * x = x for all x in {0, 1}.

    :param expr: sympy expression to be simplified
    :param symbols: symbols that are used in the expression
    :return: the simplified sympy expression
    """

    def recurse_replace(expr, func):
        """
        Traverses the expression tree in a DFS order and replaces all nodes with the result of applying them to the
        given function.

        :param expr: the expression tree
        :param func: the transformation function
        :return: the transformed expression tree
        """
        if len(expr.args) == 0:
            return expr
        else:
            new_args = tuple(recurse_replace(a, func) for a in expr.args)
            return func(expr, new_args)

    def rewrite(expr, new_args):
        """
        Traverses the arguments of a sympy expression node. If an argument is a power, it is replaced by the base of the
        power.

        :param expr: expression node or polynomial whose arguments are verified
        :param new_args: a list of nodes (simpy expression or integer) that would be the new arguments of the expr node.
```

```
        :return: an object having the same type as expr, and having as arguments the transformed new_args list.
        """
        args = []

        for arg in new_args:
            if hasattr(arg, 'is_Pow') and arg.is_Pow:
                args.append(arg.args[0])
            else:
                args.append(arg)

        if hasattr(expr, 'is_Poly') and expr.is_Poly:
            return Poly(args[0], expr.gens)
        else:
            new_node = type(expr)(*args)
            return new_node

    p = Poly(expr, symbols, modulus=2)
    p = recurse_replace(p, rewrite).set_modulus(2)

    return p.as_expr()


def getSymbolsByDecreasingFrequency(expression):
    """
    Returns all the symbols of the given sympy expression in decreasing order of their frequency of appearance.

    Example:
    expression: m1 + r1*r2 + r1 + 1
    frequency: {m1: 1, r2: 1, r1: 2}
    result: [r1, m1, r2]

    :param expression: sympy expression
    :return: list of sympy symbols
    """
    freq = {}
    for node in sympy.preorder_traversal(expression):
        if hasattr(node, 'is_Symbol') and node.is_Symbol:
            freq[node] = freq.get(node, 0) + 1
    symbols = sorted(freq.keys(), key=lambda k: freq[k], reverse=True)
    return symbols


def collectExpression(expression):
    """
    Groups the terms of the expressions, based on the frequency of its symbols.

    Example:
    expression: m1 + r1*r2 + r1 + 1
    frequency: {m1: 1, r2: 1, r1: 2}
    result: m1 + r1*(r2 + 1) + 1

    :param expression:
    :return:
    """
    symbols = getSymbolsByDecreasingFrequency(expression)
    return sympy.collect(expression, symbols)

```

```python

def formulasToExpressionTree(formulas, merge=False, simplify=False, collect=False, substitutions=None):
    """
    Transforms a Formulas object, by replacing the propositional logic propositions with sympy algebraic expressions.

    :param formulas: the Formulas object whose propositions are to be transformed
    :param merge: specifies whether to create a single expression from all the expressions of the Formulas object by
                  multiplying them (logical AND)
    :param simplify: specifies whether to simplify the expressions using the rules of the boolean ring.
                     If False is given, the result can have coefficients and powers that are not part of the ring.
    :param collect: specifies whether to collect the terms of each expression by the decreasing frequency of their
                    symbols.
    :param substitutions: a dictionary whose elements are (string_symbol, (expression, symbols)).
                          No substitution if None is given.
        string_symbol: the string which will be substituted
        expression: a sympy expression which substitutes the string
        symbols: the list of sympy symbols that are present in the expression
    :return: Formula object with the same name, but the propositions replaced with simpy expression(s)
    """
    symbols = set()

    def transform(prop):
        expr, sym = buildExpressionTree(prop, substitutions)
        symbols.update(sym)
        return expr

    expressions = map(transform, formulas.formulaList)

    if merge:
        expressions = [reduce(lambda a, b: Mul(a, b), expressions, 1)]

    if simplify:
        expressions = map(lambda e: simplifyExpression(e, symbols), expressions)

    if collect:
        expressions = map(collectExpression, expressions)

    return Formulas(formulas.name, expressions)


def createSubstitutions(predicates):
    """
    Given a list of predicates in the form of `a <-> f(b)`, returns a dictionary which associates
    to each symbol `a` the sympy expression representing `f(b)` and the used symbols.

    :param predicates: list of strings, each having the form `a <-> f(b)`, where `a` is a string propositional logic
```

```
                        symbol, and `f(b)` is the proposition which replaces
    `a`.
    :return: a dictionary whose elements are (string_symbol, (expression,
    symbols)).
        string_symbol: the string which will be substituted
        expression: a sympy expression which substitutes the string
        symbols: the list of sympy symbols that are present in the
    expression
    """
    substitutions = {}

    for predicate in predicates:
        terms = splitProposition(predicate)
        if len(terms) < 2 or terms[1] != '<->':
            raise ValueError('Substitutions of type `a <-> f(b)` are
    supported only')
        substitutions[terms[0]] = buildExpressionTree(" ".join(terms[2:]))

    return substitutions
```

C.4 Printing the result

Listing C.4: `output.py`

```
"""
Functions for printing the program output to STDOUT.
"""


def printCommand(command):
    """
    Prints an instance of the Command class.

    :param command: command to be printed
    :return: None
    """
    print command.name + '(' + ', '.join(command.args) + ').'


def printCommands(commands):
    """
    Prints a list of Command objects.

    :param commands: list of commands to be printed
    :return: None
    """
    for command in commands:
        if command.args and command.args[0] not in ['arithmetic', '
    domain_size']:
            printCommand(command)
    print "set(arithmetic)."
    print "assign(domain_size, 2)."


def printFormulas(formulas, mod2Output=False):
    """
    Prints an instance of the Formulas class.

```

```
    :param formulas: object to be printed
    :param mod2Output: if True: wraps each expression with "mod 2"
    :return: None
    """
    print formulas.commandName + '(' + formulas.name + ').'

    for expression in formulas.formulaList:
        if mod2Output:
            print '\t(' + str(expression) + ') mod 2 = 1.'
        else:
            print '\t' + str(expression) + ' = 1.'

    print formulas.endCommand + '.'


def printFormulasList(formulasList, mod2Output=False):
    """
    Prints a list of Formulas objects.

    :param formulasList: list of objects to be printed
    :param mod2Output: if True: wraps each expression with "mod 2"
    :return: None
    """
    for formulas in formulasList:
        printFormulas(formulas, mod2Output)
        print


def printCommandsAndFormulas(commands, formulas, mod2Output=False):
    """
    Prints a list of Command and a list of Formulas objects.

    :param commands: list of Command objects to be printed
    :param formulas: list of Formulas objects to be printed
    :param mod2Output: if True: wraps each formula expression with "mod 2"
    :return: None
    """
    printCommands(commands)
    print
    printFormulasList(formulas, mod2Output)
```

C.5 Main program

Listing C.5: translator.py

```
"""
Main program
"""

import getopt
import os
import sys

from expression import formulasToExpressionTree, createSubstitutions
from input import processLines, linesToCommandsAndFormulas
from output import printCommandsAndFormulas

# Name of the program
```

```python
programName = os.path.basename(__file__)

# Program configuration
inputFile = sys.stdin
mod2Output = False
collect = False
substitute = False
merge = False

# Help message
usage = "%s [options] [-f <inputfile>]" % programName
description = "A converter from propositional logic to modular arithmetic in the mace4 format"
help_message = """
Usage: %s

%s

Options:
    -c,\t--collect\tCollects the terms of each expression by the decreasing frequency of their symbols.
    -h,\t--help\t\tPrints help message
    -f,\t--file string\tPath to the input file. If the option is missing, the program reads from STDIN
    -m,\t--merge\t\tMerges all the expressions of a "formulas" section into a single expression.
    \t--mod\t\tWraps all expressions in "mod 2"
    -s,\t--subs\t\tSubstitutes symbols with their corresponding expression. The substitutions must be given in the "a <-> f(b)." format in "formulas(substitutions)."
""" % (usage, description)


def processArguments(argv):
    """
    Reads the command line arguments of the program and updates respectively the global variables

    :param argv: the command line arguments, list of strings
    :return: None
    """
    global inputFile, mod2Output, collect, substitute, merge

    try:
        opts, args = getopt.getopt(argv, "hcsmf:", ["mod", "help", "collect", "subs", "merge","file="])
    except getopt.GetoptError:
        print help_message
        sys.exit(2)
    for opt, arg in opts:
        if opt in ['-h', '--help']:
            print help_message
            sys.exit()
        elif opt in ['-f', '--file']:
            inputFile = open(arg, 'r')
        elif opt == '--mod':
            mod2Output = True
        elif opt in ['-c', '--collect']:
            collect = True
        elif opt in ['-s', '--subs']:
```

```python
            substitute = True
        elif opt in ['-m', '--merge']:
            merge = True


def main(argv):
    """
    Main program.

    Reads the command line arguments.
    Reads and processes the input data.
    Prints the result to STDOUT.

    :param argv: the command line arguments, list of strings
    :return: None
    """
    processArguments(argv)
    lines = processLines(inputFile.readlines())

    commands, formulas = linesToCommandsAndFormulas(lines)

    substitutions = None
    if substitute:
        try:
            substitutionFormulas = next(f for f in formulas if f.name == "substitutions")
        except StopIteration:
            raise ValueError('Cannot substitute. \"formulas(substitutions).\" does not exists')
        substitutions = createSubstitutions(substitutionFormulas.formulaList)
        formulas.remove(substitutionFormulas)

    formulas = map(lambda f: formulasToExpressionTree(f, simplify=True, collect=collect, substitutions=substitutions, merge=merge),
                   formulas)

    printCommandsAndFormulas(commands, formulas, mod2Output=mod2Output)


if __name__ == '__main__':
    try:
        main(sys.argv[1:])
    except ValueError as e:
        sys.stderr.write('Error: ' + str(e))
        sys.exit(1)
```