Solving Logic Puzzles using Model Checking in LTSmin

Kamies, Bram
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
b.kamies@student.utwente.nl

ABSTRACT
Model checking is used for checking whether systems conform to a given specification. These could instead also model puzzles in order to find their solutions. But little is known about restrictions (such as specific game rules) to implementing these games or about how complex such an implementation would be. The paper will compile a list of puzzles and group them based on their mechanics. Then compare the implementation and results for these puzzles to find relations between the data and their game rules. Based on this the data showed that rules that require the board to be explored for groups of tiles and their shapes has the most effect on their complexity.

Keywords
Logic, model checking, puzzles, puzzles, state space, Sokoban, Bloxorz, rush hour.

1. INTRODUCTION
Model checkers are used for testing often complex parallel systems, but can also be used for solving puzzles. This is done by verifying the "unsolvable" property of the puzzle. Meaning that the model checker thinks the system is broken if there is a solution to the puzzle. An advantage of using model checking for solving puzzles is that if the model checker determines the puzzle is solvable, it also can give the steps necessary to get to the solution.

The model checker to be used for solving these puzzles is the LTSmin toolset [2][4]. Not all puzzles will be considered for the list of puzzles to model check. The main focus lays on pure logical puzzles, such that they have no hidden information, no random elements, no opponents and being playable on a grid or board. When talking about these limitations it is important to keep in mind that these reflect the puzzle rules, for example every puzzle has a hidden element namely the solution. When saying there is no hidden information or random elements it means there is no hidden information or random elements that influences the choice the player makes. The limitations will make it easier to simulate the puzzles as they are optimal for model checking. See Table 1 for a list of the selected puzzles and their grouping. Any of the puzzles can be encoded in mainly two ways, either a matrix with some extra variables or a list of entities that make up the puzzle, though it is unknown what approach is better for specific puzzles to turn them into model checking problems. First chapter 2 will define the research questions of this paper. In chapter 3 background information on the tools and chosen puzzles will be given. Chapter 4 credits related work. Chapter 5 shows the implementation for these puzzles and explains the working of their algorithms. Chapter 6 reports the results from testing the algorithms. The research questions are answered in chapter 7. Finally some future work is mentioned in chapter 8.

24th Twente Student Conference on IT, July 1st, 2016, Enschede, The Netherlands.

Table 1. Selected puzzles and their grouping

Puzzle name	Group
Fox, goose and bag of beans*	Transport puzzle
Bridge and torch problem	Transport puzzle
Bloxorz*	Block sliding puzzle
Sokoban	Block sliding puzzle
Picross*	Binary determination puzzle
Nurikabe	Binary determination puzzle
Sudoku*	Sudoku

*= primary puzzle of the group, to be model checked

2. RESEARCH QUESTIONS
The puzzles chosen have in common that they can be played on a grid or a board. The puzzles have no hidden information or random element and all puzzles are single player. These puzzles are essentially purely logical puzzles, because of these key game mechanics. This makes modelling them easier. By modelling different types of puzzles, relations between their implementation and rule set can be found. From this we can also analyze how well these puzzles translate to model a model checker.

Main research question: What puzzles from Table 1 can be efficiently solved using model checking?

1. What search strategy used to explore the state space fits best for the puzzles?
2. What are the effects the rules of the puzzles have on their state space?
3. What rules make the implementation of a puzzle more complex?

2.1 METHOD OF RESEARCH
The first step is to make implementations of the puzzles, starting with the smaller and simpler puzzles (from the transport puzzles group) to gain experience with programming for LTSmin. After that the next step is to make an implementation for one puzzle (the primary) of each group. With these implementations the first two questions can be answered. The implementations will be tested on a virtual machine running Ubuntu 64-bit 16.04 using 1 processor core, 4.3 GB ram, and 20 GB assigned hard drive space. For the Bloxorz and Picross puzzles a small stress test will be used to test DFS and BFS strategies for running time and

depth/transition count. Each test is run 5 times and the runtime is averaged. Finally based on results from testing and reviewing the algorithms used to solve the puzzles the last question can be answered.

The implementations will consist of two c-functions which are passed to LTSmin in order to calculate new states and to check whether the puzzle is solved or not. In the pseudocode these are `next_state` and `check_goal`. The `next_state` function reports newly found states through a callback which is called `add_state` in the pseudocode. For simplicity in pseudo code it is assumed that values passed to functions are copies of the local value, so passing state to a function that modifies it doesn't change the local value.

3. BACKGROUND

3.1 LTSmin

The LTSmin toolset is a model checking package with a core which links different specification languages (and C code) to an algorithm backend which allows for different methods to be used when solving the problems with model checking. Languages supported are (at time of writing) DVE, ETF, mCRL2 muCRL, Promela, UPPAAL and PBES. Modeling using C-code is done using the PINS interface. This interface allows for models to be expressed in C-code. On the backend of the toolset are different algorithms for exploring the statespace for a given model. Among the settings for these algorithms is an option for setting the strategy used for searching new states. The initial state forms the root of a tree that can either be explored using depth-first search (DFS) or breadth-first search (BFS). When using DFS LTSmin keeps a stack of the states that are queued to be explored for new states and picks the last state and gets the states that can be reached from that state. New states are added to the end of the queue. When using BFS the program explores the first item in the stack of states while still appending new states to the end of the stack. LTSmin is able to remember states in such a way that it can quickly identify whether a state has been previously found or not. This way loops in the state space are ignored and not revisited.

3.2 State space

The state space is the space of possible states a model can be in. Theoretically any PSPACE problem can be solved on a Turing machine but some problems would need a very large amount of memory in order to find the solution, which causes model checkers to run out of memory. Reducing the state space is the main challenge for model checkers.

3.3 Sudoku

A popular number-placement puzzle with a set of rules dictating what numbers can and can't be placed. Variations of this puzzle that have been chosen are Nonomino and KenKen. Nonomino: also known as jigsaw Sudoku, where the original 3x3 regions have been replaced by 9 polygons made up of 1x1 squares connected by their edge. The rules of Sudoku have not changed otherwise. KenKen: a grid of 4x4 has to be filled with the numbers 1 through 4 with each number appearing once in each column and row. With that regions have been drawn and have been assigned an operation and an answer. This operation must hold for the numbers entered and be equal to the answer assigned to the region. There also exist versions of this puzzle with 5x5 and 6x6 grids.

3.4 Binary determination puzzles

In a binary determination puzzle the player has to make a binary choice, usually for a group of cells in a grid. The effects of this choice depend on the rules of the puzzle. For example in the puzzle picross the player has to draw a picture by filling in squares (based on hints for each column and row) or leaving them blank.

3.5 Block sliding puzzles

A block sliding puzzle is a puzzle where the player controls one or more blocks to navigate them in a grid. The puzzle most true to this concept is Klotski, where the player is required to move a 2x2 tile through the exit of the puzzle. The puzzles chosen have various variations on the concept. Rush Hour: similar to Klotski, but all blocks are cars in that they are always 1 tile wide and can't move sideways. Sokoban: a single man is controlled that has to push crates into storage locations. Blocnog: The player is required to extend their block into a specific shape before reaching the finish. Bloxorz: the block the player controls changes shape depending on the move made, this is because the player controls a 1x1x2 block. In Bloxorz the player is required to put the block straight up on a specific tile in the grid.

3.6 Transport puzzles

A transport puzzle is a puzzle where a group of items or persons have to be transported between locations, with certain restrictions to prevent combinations of items or persons to travel at the same time or be in the same place. One such puzzle is the Fox, goose and bag of beans puzzle, in which a farmer is transporting a fox, goose and bag of beans from one side of a river to another. With the following restrictions: the fox will eat the goose if they are left together, the goose will eat the beans if they are left together. Though not very challenging, it is a type of puzzle that lends itself for solving logistical problems. Another puzzle is the bridge and torch problem. The puzzle describes a group of four people trying to cross a bridge at night. Each of them runs at a different speed. At most two people can cross the bridge at once, but only while carrying a torch of which there is only one. The puzzle is solved when everyone is at the other side of the bridge and 15 minutes or less have passed

4. RELATED WORK

A lot of research has already been done into puzzles themselves. Some relevant research into puzzles are proofs that puzzles are NP-complete and PSPACE-complete. Sliding puzzles have been proven to be PSPACE-complete by Robert A. Hearn et al. [3]. Tom C. van der Zanden et al. have proven that the puzzle Bloxorz and puzzle puzzles in a broader sense are PSPACE complete [6].

Gihwon Kwon has done research into solving Sokoban puzzles using model checking [5]. In his research he makes several optimizations to reduce the space explosion problem in order to solve the toughest levels.

Vincent Bloemen has looked into the viability of using model checkers for probabilistic puzzles [1]. This research will, in a way, extend this as the viability of LTSmin for solving puzzles in a general manner is a byproduct of the research.

5. MODELLING

5.1 Fox, goose and the bag of beans

The state of the puzzle is stored in an array of integers which is cast into a struct (made up of integer fields), with 0 indicating the item is on the left side of the river, and 1 for the opposite side of the river. Aside from the items the man also has an entry in the list.

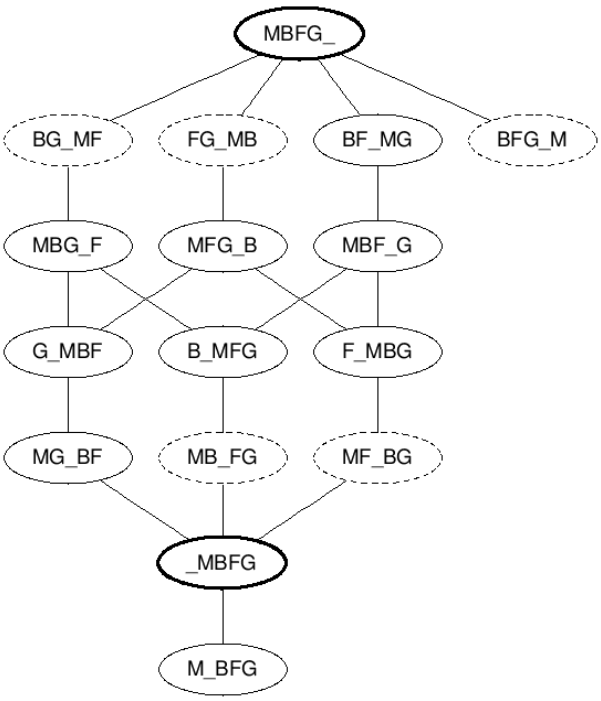

Figure 1. Complete state space of the fox, goose and bag of beans problem

```
#define LEFTSIDE 0
#define RIGHTSIDE 1
typedef state_t {
    int fox; int goose; int beans; int man;
}
```

In the `next_state` function the algorithm has to find all possible states that can be reached from a given state. LTSmin caches previous states and explores only new states. In the pseudocode this is done by calling `add_state` passing the newly found state. The state space is displayed Figure 1, where the underscore indicates the river, meaning letters to the left of the underscore are on the left side of the river and letters to the right are on the right side of the river. The letter M indicates the man with his boat, the F stands for the fox, the G for the goose and the B for the bag of beans. States that have a dashed outline are invalid according to the rules and won't be reported to the `add_state` function. The bold states are begin- and end-states.

```
next_state(state) {
  try_switch(state, fox)
  try_switch(state, goose)
  try_switch(state, beans)
  try_switch_man(state)
}

try_switch(state, item) {
  if (state[item] == state.man) {
    state[item] = 1 - state[item]
    state.man = 1 - state[item]
    if is_valid(state)
      add_state(state)
  }
}
try_switch_man(state) {
  state.man = 1 - state[item]
  if is_valid(state)
    add_state(state)
}
```

The `is_valid` function is used to verify that after a move the resulting state is valid. If it isn't it won't be reported as a new state. By the rules of the puzzle the state is invalid if the fox and goose or goose and bag of beans are left alone.

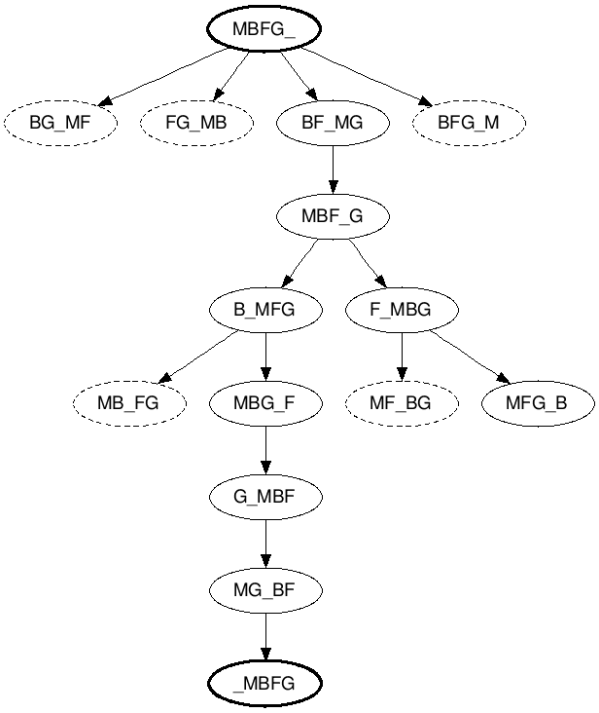

Figure 2. Actual state space as found with BFS strategy

```
is_valid(state) {
  return (state.fox != state.goose ||
    state.fox == state.man) &&
    (state.goose != state.beans ||
    state.goose == state.man)
}
```

The `check_goal` function checks whether the puzzle is solved using the `is_valid` function in combination with a check for the end conditions.

```
check_goal(state) {
  return is_valid(state) &&
    state.fox == RIGHTSIDE &&
    state.goose == RIGHTSIDE &&
    state.beans == RIGHTSIDE
}
```

As LTSmin explores the states that are being found a directed graph can be made of the states that are found. Figure 2 shows the states as they are found by LTSmin when using BFS. In the figure only lines to new states are drawn, meaning some lines to invalid states are missing as well as the line between MFG_B and G_MFB.

5.2 Bridge and torch

The bridge and torch problem's main difference with the fox goose and beans problem is the time limit. The boat has been replaced with a torch and the focus of the puzzle lays on getting everyone across under a certain time limit. Each person walks at a different speed which affects the time it takes them to cross the bridge. The times it takes for each person to cross are 1, 3, 5 and 8 minutes. In the pseudo code person A is the fastest, taking only 1 minute, followed by person B taking 3 minutes, person C takes 5 minutes and person D takes 8 minutes. When two people cross the bridge at the same time, the faster walker needs to slow down to stay with the other person.

```
#define LEFTSIDE 0
#define RIGHTSIDE 1
typedef state_t {
int personA; int personB;
int personC; int personD;
int torch; int time;
}

next_state(state) {
  //Switch single person
  try_switch1(state, 1, personA)
  try_switch1(state, 2, personB)
  try_switch1(state, 5, personC)
  try_switch1(state, 8, personD)
  //Switch 2 persons
  try_switch2(state, 2, personA, personB)
  try_switch2(state, 5, personA, personC)
  try_switch2(state, 8, personA, personD)
  try_switch2(state, 5, personB, personC)
  try_switch2(state, 8, personB, personD)
  try_switch2(state, 8, personC, personD)
}

try_switch1(state, inctime, person) {
  if state.torch == state[person] {
    state.torch = 1 - state.torch
    state[person] = 1 - state[person]
    state.time += inctime
    if is_valid(state)
      add_state(state)
  }
}

try_switch2(state, inctime, person1, person2) {
  if state.torch == state[person1] &&
     state.torch == state[person2] {
    state.torch = 1 - state.torch
    state[person1] = 1 - state[person1]
    state[person2] = 1 - state[person2]
    state.time += inctime
    if is_valid(state)
      add_state(state)
  }
}
```

Since the `try_switch` functions prevents crossing the bridge without the torch, the `is_valid` function only needs to check that the time.

```
is_valid(state) {
  return (state.time <= 15)
}
```

The `check_goal` function uses the `is_valid` function to check the time constraint and confirms everyone is has crossed the bridge.

```
check_goal(state) {
  return is_valid(state) &&
    state.personA == RIGHTSIDE &&
    state.personB == RIGHTSIDE &&
    state.personC == RIGHTSIDE &&
    state.personD == RIGHTSIDE
}
```

5.3 Sokoban

The Sokoban algorithm was made by storing the entire board into an array and keeping the state of the player separate. Each cell is coded with a value indicating what is in that cell. Since the player and boxes need to be able to stand on top of goal tiles the number stored in each cell is encoded using a bitflag. The `WIDTH` and `HEIGHT` of the board are put into define statements and need to be adjusted for different sized boards.

```
#define EMPTY 0
#define WALL 1
#define BOX 2
#define GOAL 4
#define MAN 8
typedef state_t {
  int board[WIDTH * HEIGHT];
}
```

In order to access the board as a two dimensional structure helper functions have been made to read and write to the board using an x- and y-coordinate. If a coordinate passed to a read and write operation lies outside of the board it pretends the tile is a permanent wall.

```
out_of_bounds(x, y) {
  return x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT
}

get_value(state, x, y) {
  if out_of_bounds(x, y) { return WALL }
  return state.board[(y * WIDTH) + x]
}

set_value(state, x, y, value) {
  if out_of_bounds(x, y) { return }
  state.board[(y * WIDTH) + x] = value
}
```

The code also uses some helper functions to decode the bitflag values. To check if a specific flag is set on the bitflag the other bits are filtered out using a bitwise and operation.

```
is_empty(cell) {
  return cell == EMPTY || cell == GOAL
}
is_wall(cell) { return cell & WALL }
is_box(cell) { return cell & BOX }
is_goal(cell) { return cell & GOAL }
is_man(cell) { return cell & MAN }
```

The moves a player can make at a given moment is to either move in one of the four cardinal directions, or push (a box) in one of the four cardinal directions. When pushing a box onto a tile, it into must be empty or a goal tile and the man must push it from the opposite side. The following code segment shows the code used for moving up and pushing a box up. Similar functions exist for moving and pushing in the other directions.

```
find_man(state) {
  for x=0,WIDTH {
    for y=0,HEIGHT {
      if is_man(get_value(state, x, y))
        return x, y
    }
  }
}

try_move_up(state) {
  x, y = find_man(state)
  current = get_value(state, x, y)
  above = get_value(state, x, y - 1)
  if is_empty(above) {
    set_value(state, x, y, current - MAN)
    set_value(state, x, y - 1, above + MAN)
    if is_valid(state)
      add_state(state)
  }
}

try_push_up(state) {
  x, y = find_man(state)
  current = get_value(state, x, y)
  box = get_value(state, x, y - 1)
  target = get_value(state, x, y - 2)
  if is_empty(target) && is_box(box) {
    set_value(state, x, y, current - MAN)
    set_value(state, x, y - 1, (box - BOX) + MAN)
    set_value(state, x, y - 2, target + BOX)
    if is_valid(state)
      add_state(state)
  }
}

next_state(state) {
  try_move_up(state)
  try_move_down(state)
  try_move_left(state)
  try_move_right(state)
  try_push_up(state)
  try_push_down(state)
  try_push_left(state)
  try_push_right(state)
}
```

The `is_valid` function makes sure the man and all boxes are not put inside of a wall. If it finds a man on top of a box or wall it returns false. It also returns false if a box is inside of a wall.

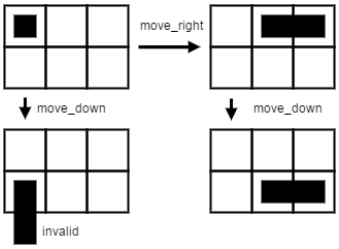

Figure 3. The orientation of the block changes the way it moves around the board

```
is_valid(state) {
  for x=0,WIDTH {
    fox y=0,HEIGHT {
      cell = get_value(state, x, y)
      if is_man(cell) && (is_box(cell) ||
         is_wall(cell))
        return 0
      if is_box(cell) && is_wall(cell)
        return 0
    }
  }
}
```

The `check_goal` function uses the `is_valid` function and confirms all goals have a box on top of them.

```
check_goal(state) {
  if !is_valid(state) { return 0 }
  for x=0,WIDTH {
    for y=0,HEIGHT {
      cell = get_value(state, x, y)
      if is_goal(cell) && !is_box(cell)
        return 0
    }
  }
  return 1
}
```

5.4 Bloxorz

Because in Bloxorz the state of the object the player controls has more data associated with it than in Sokoban, a different approach in encoding the state of the puzzle was chosen. Instead of encoding the player into the board, the board and the player information was kept separate. This was done because for Bloxorz the state of a player also involves the rotation of the block (standing up or laying down on either axis). Implementing this by encoding the squares a player occupies would add the overhead of locating and identifying the orientation of the player in every `next_state` function call.

```
#define STANDING 0
#define LAYING_X 1
#define LAYING_Y 2
#define EMPTY 0
#define FLOOR 1
#define GOAL 2
typedef state_t {
  int x; int y; int rot;
  int board[WIDTH * HEIGHT];
}
```

The position of the player is measured as the top left square the player occupies. When the player is lying along the x-axis this would be the left tile and the top tile when lying along the y-axis.

Because the orientation of the player changes the way the player navigates the board each direction the player can move in handles the player's orientation differently. An example of this is shown in Figure 3, in which the black block is the current tiles the player occupies. In the following code snippet the functions to move the player up and down are shown.

```
try_move_up(state) {
  switch(state.rot) {
    case STANDING:
      state.y -= 2
      state.rot = LAYING_Y
    case LAYING_X:
      state.y -= 1
    case LAYING_Y:
      state.y -= 1
      state.rot = STANDING
  }
  if is_valid(state) {
    add_state(state)
  }
}

try_move_down(state) {
  switch(state.rot) {
    case STANDING:
      state.y -= 1
      state.rot = LAYING_Y
    case LAYING_X:
      state.y -= 1
    case LAYING_Y:
      state.y -= 2
      state.rot = STANDING
  }
  if is_valid(state)
    add_state(state)
}
```

The `is_valid` function ensures that at all times the player is fully supported by the level. It uses a helper function `is_floor` to detect whether a specific tile can support the player (both `FLOOR` and `GOAL` can support the player). Tiles outside of the board are assumed to be `EMPTY`.

```
is_floor(state, x, y) {
  if x < 0 || x >= WIDTH || y < 0 || y >= HEIGHT
    return 0
  return state.board[(y * WIDTH) + x] != EMPTY
}

is_valid(state) {
  if !is_floor(state, state.x, state.y)
    return 0
  switch(state.rot) {
    case STANDING:
      return 1
    case LAYING_X:
      return is_floor(state, state.x + 1, state.y)
    case LAYING_Y:
      return is_floor(state, state.x, state.y + 1)
  }
}
```

A Bloxorz level is solved by putting the player's block standing on top of a `GOAL` tile. The `check_goal` function tests this by confirming the player is standing and on top of a `GOAL` tile.

```
is_goal(state, x, y) {
  return state.board[(y * WIDTH) + x)] == GOAL
}

check_goal(state) {
  return state.rot == STANDING &&
         is_goal(state, state.x, state.y)
}
```

5.5 Sudoku

The state of Sudoku is stored in an 81 slot array. Accessing the board using x- and y-coordinates is done as with previous puzzles using helper functions. But instead of tracking a player object every time a next state has to be determined it finds the next empty slot and tries the values 1-9 and reports back all states that still meet the Sudoku requirements.

```
#define EMPTY 0
typedef state_t {
  int board[WIDTH * HEIGHT];
}

get_value(state, x, y) {
  return state.board[(y * WIDTH) + x)]
}

set_value(state, x, y, value) {
  state.board[(y * WIDTH) + x)] = value
}

next_empty(state) {
  for y=0,HEIGHT
    for x=0,WIDTH
      if get_value(state, x, y) == EMPTY
        return x, y
  return -1, -1
}

next_state(state) {
  x, y = next_empty(state)
  if x == -1 { return }
  for i=1,10 {
    try_value(state, x, y, i)
  }
}

try_value(state, x, y, value) {
  set_value(state, x, y, value)
  if is_valid(state) {
    add_state(state)
  }
}
```

The `is_valid` function checks for each row and column that each number only appears once (ignoring `EMPTY` values). Also it does this for each 3-by-3 block. Only the implementation for validating columns is given, but validating rows works the same apart from using a different axis.

```
is_valid(state) {
  for x=0,9
    if !is_valid_column(state, x)
      return 0
  for y=0,9
    if !is_valid_row(state, x)
      return 0
  for x=0,3
    for y=0,3
      is_valid_block(state, x*3, y*3)
}

is_valid_column(state, x) {
  int count[9] = {}//filled with 0's
  for y=0,9 {
    value = get_value(state, x, y)
    if value != EMPTY
      count[value]++
      if count[value] > 1
        return 0
  }
  return 1
}

is_valid_block(state, _x, _y) {
  int count[9] = {}
  for x=_x, _x + 3 {
    for y=_y, _y + 3 {
      value = get_value(state, x, y)
      if value != EMPTY {
        count[value]++
        if count[value] > 1
          return 0
      }
    }
  }
  return 1
}
```

In order to check whether the puzzle is solved the algorithm only needs to check that there is no empty cell in the grid and that `is_valid` returns true.

```
check_goal(state) {
  x, y = next_empty(state)
  if x == -1 {
    return is_valid(state)
  }
  return 0
}
```

5.6 Picross

The implementation made for picross is similar to that of Sudoku in the way that the algorithm finds the next empty cell and gives it a value. For picross this value is either white or black. The hints for the rows and columns are stored on the state struct. A constant `MAX_HINTS` indicates the largest number of hints any row of column can have.

```
#define NO_HINT 0
#define EMPTY 0
#define WHITE 1
#define BLACK 2
typedef state_t {
  int hints_x[WIDTH][MAX_HINTS];
  int hints_y[HEIGHT][MAX_HINTS];
  int board[WIDTH * HEIGHT];
}
```

The pseudo code for helper functions and `next_state` is nearly identical to that of Sudoku, with the difference of the values being tried.

```
get_value(state, x, y) {
  return state.board[(y * WIDTH) + x)]
}

set_value(state, x, y, value) {
  state.board[(y * WIDTH) + x)] = value
}

next_empty(state) {
  for y=0,HEIGHT
    for x=0,WIDTH
      if get_value(state, x, y) == EMPTY
        return x, y
  return -1, -1
}

next_state(state) {
  x, y = next_empty(state)
  if x == -1 { return }
  try_value(state, x, y, WHITE)
  try_value(state, x, y, BLACK)
}

try_value(state, x, y, value) {
  set_value(state, x, y, value)
  if is_valid(state)
    add_state(state)
}
```

The most complex part of the algorithm comes from the `is_valid` function, which needs to analyze each row and column to verify it obeys the hints.

```
is_valid(state) {
  for x=0,WIDTH
    if !is_valid_column(state, x)
      return 0
  for y=0,HEIGHT
    if !is_valid_row(state, y)
      return 0
}
```

The following function verifies a given column obeys to the hints of that column. A similar function exists for verifying rows. The function starts by fetching the array of hints, and then loops through this array to read every hint. This is done by either checking all or reaching a hint of length 0 (`NO_HINT`).

To check a hint first the next black square in the column is found. If the end of the column is reached then the function returns false because one or more hints were missed. After finding the first black square it verifies that the number of consecutive black squares after the first black square matches that of the hint. The algorithm also makes sure there is a white

square after the row of black squares. The variable _y tracks the current position in the column. If at any point a blank square is found, the function returns true. This makes sure that the hints up to the blank square are correct, without skipping the column entirely.

```
is_valid_column(state, x) {
  hints = state.hints_x[x]
  y = 0
  for i = 0,MAX_HINTS {
    hint = hints[i]
    if hint == NO_HINT
      break
    if y >= HEIGHT {
      return 0
    //Find next black square
    while y < HEIGHT {
      int value = get_value(state, x, y)
      if value == BLACK
        break
      if value == EMPTY
        return 1
      y++
    }
    //Read hint number of squares
    for j = 0, hint {
      if y >= HEIGHT
        return 0
      int value = get_value(state, x, y)
      if value == WHITE
        return 0
      if value == EMPTY
        return 1
      y++
    }
    //In case the hint doesn't end with the column
    if y < HEIGHT {
      //Check next is not black
      int value = get_value(puzzle->board, x, y)
      if (value == BLACK)
        return 0
      if (value == EMPTY)
        return 1
    }
  }
  //Check remaining tiles to be not black
  while y < HEIGHT {
    if get_value(state, x, y) == BLACK)
      return 0
    y++
  }
  return 1
}
```

5.7 Nurikabe

An attempt was made to implement this puzzle, though it didn't yield any timing or depth result, it did give an insight into the difficulty of programming its possible solution. The part of the algorithm that couldn't be devised was about verifying the state of the puzzle. The rules of the puzzle state that a hint indicates how many white squares are connected to it and are not allowed to be connected together. A final rule requires all black squares to be connected and not fill any 2x2 area.

The algorithm would have to explore the board like a maze multiple times, once for each hint and once to check the black squares. While checking the black squares it would also have to confirm it doesn't contain any squares of 2x2.

Checking all these requirements made the algorithm complex to the point where it could not be completed in the given timeframe. This showed that rules involving groups of tiles that can vary in shape and require the algorithm to explore the board make the algorithm more complex.

6. RESULTS

The version of LTSmin used is tacas2015-dirty, which is used on Ubuntu 16.04 running inside a virtual machine. The host OS is Windows 7 64-bit, the virtual machine software is VMware Workstation 12 Player. The physical hardware the VM has access to consists of an Intel i7 processor running at 2.30 GHz, 4.3GB of RAM and 20GB of hard-drive space.

Table 2. Timing results for DFS and BFS tests

Puzzle	Duration DFS	Duration BFS
Fox, goose and bag of beans	0.002s	0.002s
Bridge and torch	0.004s	0.002s
Bloxorz	0.258s	0.002s
Sudoku	0.047s	0.081s

Table 3. Test results for various sizes of Bloxorz puzzles

Level size	Duration BFS	Depth BFS	Duration DFS	Depth DFS
10x10	0.005s	13	0.010s	58
20x20	0.069s	27	0.051s	199
50x50	4.271s	67	1.535s	1329
100x100	1m16.073s	133	25.989s	5578
150x150	13m38.848s	201	2m38.044s	13150

Table 4. Test results for various sizes of Sokoban puzzles

Level size	Duration BFS	Depth BFS	Duration DFS	Depth DFS
10x10	0.128s	20	0.060s	72
20x20	9.087s	40	1.491s	152
50x50	>30m	NA	4m37.992s	392
100x100	>30m	NA	>30m	NA
150x150	>30m	NA	>30m	NA

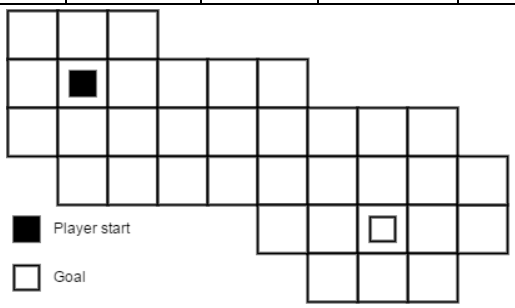

Figure 4. The first level of Bloxorz

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		6			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figure 5. The board used for testing the Sudoku algorithm

The commands used to test implementations with DFS and BFS are listed below.

```
time pins2lts-seq game.so --strategy=bfs
  --invariant="! goal"
time pins2lts-seq game.so --strategy=dfs
  --invariant="! goal"
```

The Linux command 'time' was used to measure the time taken to find the solution to a puzzle. Of the output from this command the 'real' time was used. The times it took for the

implementations to find a solution to the puzzle is shown in Table 2. The level used for testing Bloxorz is shown in Figure 4. For Sudoku the board shown in Figure 5 was used for testing.

6.1 Sokoban

In the tests to find the limits for Sokoban a square level of different sizes and without walls was used. The player is placed in the top-left corner of the map with a box placed one down and one to the right of the player. The goal of the level is placed in the very bottom-right corner of the map.

6.2 Picross

The results of testing the picross algorithm with puzzles of different sizes are shown in Table 5.

7. CONCLUSIONS

First the sub-questions will be answered before answering the main research question.

7.1 What search strategy used to explore the state space fits best for the puzzles?

The method of implementation for the binary determination group and Sudoku naturally favors DFS because the puzzles require a specific amount of answers to be filled in before the puzzle can be considered complete. The underlying reason for this is that the DFS strategy is able to reach the required depth to fill the entire board sooner than that of BFS. Intuitively this also means that BFS does take longer to reach its required depth but can quickly enumerate all possible answers once the required depth has been reached.

For puzzles where the player controls an entity (Sokoban and Bloxorz) showed that using the BFS strategy yielded better results (using less transitions), but DFS was still faster on time. These puzzles don't require a specific amount of transitions before reaching a definite answer. This means that DFS can find a path faster, which may be longer but won't contain loops. Whereas BFS finds an answer using the minimal amount of transitions but needs to explore much more states to reach the same depth DFS would need.

7.2 What are the effects the rules of puzzles have on their state space?

If you compare the results for Bloxorz in Table 3 with the results for Sokoban in Table 4 it is clear that Sokoban has a faster growing state space than Bloxorz. This can be explained by the fact that in Sokoban the player can move boxes. When a player moves a box in Sokoban it is essentially modifies the board area in which the man can move. In the tests the level only consisted of a big square room with one box and one goal. Whenever the man pushed the box onto a new square the man was able to traverse the entire board without reaching a previous state. While in Bloxorz any state the player is in can only occur once, since the rest of the state is a static board. So the number of pieces that the player can move directly or indirectly severely impacts the state space.

For the Sudoku game the rules allowed the algorithm to optimize the exploration of the state space. As soon as the algorithm finds any part of the board to be invalid it is able to terminate all states that would come after it. This is also seen for Picross and explains why the test with a 30-by-30 board took much less time than the previous 25-by-25 board. Because the only difference besides the size of the board is the hints for the puzzle, it was the hints that allowed the algorithm to find a solution to the puzzle quicker. The hints of that particular Picross puzzle were able to eliminate more possible solutions than the hints for the 25-by-25 board could.

Table 5. Testing duration of picross puzzles

Level size	Duration BFS	Duration DFS
5x5	0.020s	0.022s
10x10	0.018s	0.048s
25x25	2m14.450s	41.035s
30x30	29.037s	14.603s
50x50	>30mins	>30mins

7.3 What rules make the implementation of a puzzle more complex?

The implementation of the transport puzzles was a straightforward translation of the rules. These only involved binary like conditions which translate with little effort to code. Beyond this their algorithm only needed to try all possible moves and confirm each new state conforms to the restrictions the rules state.

Puzzles from the block sliding group were more complex because each move had its own requirements. Rather than only checking the end state to be valid, first a move needs to be possible. Comparing the solutions to Bloxorz and Sokoban, Bloxorz showed that separating movable objects in the puzzle from the board of the puzzle can simplify implementation when the objects become more complicated. Bloxorz player object could not only move but also change shape. Separating these removes the overhead of identifying the information when calculating next possible states.

The puzzles from the binary determination group turned out to be the most difficult to implement compared to their description. Their rules described patterns spanning over multiple tiles, for example Nurikabe requires the algorithm to search and count all tiles connected to a tile that contains a hint as well as demanding certain shapes do not exist in the shape the connected tiles form. Even though Sudoku is similar to puzzles from the binary determination group, its complexity was much lower due to its rules affecting specific groups of tiles (rows, columns and 3-by-3 blocks) which didn't require the algorithm to explore the board but only check these known groups on certain conditions.

7.4 What puzzles can be efficiently solved using model checking?

The transport puzzles showed that it takes little effort to implement puzzles where the rules only limit the possible states by denying specific combinations from existing.

Implementations (and lack thereof) for puzzles in the binary determination group show that puzzles with rules about patterns and shapes make their implementation more complex and time consuming.

The Bloxorz and Sokoban puzzles showed that for puzzles where the player controls one or more objects, the number of objects cause the state space to grow.

When solving a puzzle, using DFS can improve the runtime of the model checking, though it doesn't make any guarantees on the depth the solution will be found at. This makes using DFS and BFS a tradeoff between speed and accuracy respectively. If the solutions to a puzzle are found by filling in the puzzle then DFS is better. This is because this method of solving the puzzle ensures the solution is found at a specific depth because every part of the puzzle is filled in (guessed) only once.

8. FUTURE WORK

The effect of having multiple movable pieces has been shown by Sokoban. Another block sliding game RushHour allows the player to directly control multiple pieces either horizontally or vertically. Researching this game could show what effect having a certain number of pieces has on the statespace.

Since the implementation of binary determination games were the most complex of the set it could be worthwhile to look into possible ways to reduce the complexity of algorithms that have to explore the board of the puzzle.

Similarly the implementation for Sokoban used could be compared to an implementation which only tracks the areas in which the player can move. This nullifies the state transitions that have to be made to get the man from one position on the board to another and would only consider the moving of a box as a move.

The games from the transport puzzle group (logistical puzzles) showed to be straightforward in their implementation but didn't lend themselves to be scaled up like other games did. Investigating this could show the potential to solve problems based around combinatory binary rules that restrict the state.

9. REFERENCES

[1] V. Bloemen. Analyzing old puzzles with modern techniques: Probabilistic model checking using SCOOP and PRISM. Diss. Bachelor's thesis, University of Twente, 2012. http://fmt.cs.utwente.nl/files/sprojects/130.pdf

[2] S. Blom, J. van de Pol, M. Weber. "Bridging the gap between enumerative and symbolic model checkers". Technical Report TR-CTIT-09-30 (http://eprints.eemcs.utwente.nl/15703/), Centre for Telematics and Information Technology University of Twente, Enschede. ISSN 1381-3625.

[3] R. A. Hearn, E. D. Demaine. "PSPACE-completeness of sliding-block puzzles and other problems through nondeterministic constraint logic model of computation", Theoretical Computer Science Volume 343(issues 1-2): 72-96. 2005. DOI: 10.1016/j.tcs.2005.05.008

[4] G. Kant, A. Laarman, J. Meijer, J. van de Pol, S. Blom, T. van Dijk. "Ltsmin: High-performance language-independent model checking". In: Tools and Algorithms for the Construction and Analysis of Systems. Springer Berlin Heidelberg, 692-707, 2015. DOI: 10.1007/978-3-662-46681-0_61

[5] G. Kwon. "Applying Model Checking Techniques to Puzzle Solving". In: Software Engineering Research and Applications. Springer Berlin Heidelberg, 290-303, 2004. DOI: 10.1007/978-3-540-24675-6_23

[6] T. C. van der Zanden, H. L. Bodlaender. "PSPACE-Completeness of Bloxorz and of Puzzles with 2-Buttons". In: Algorithms and Complexity. Springer Berlin Heidelberg, 403-415, 2015. DOI: 10.1007/978-3-319-18173-8_30