Symbolic Model Checking in Puzzle Games

Automated Reachability Analysis

Jacob Eskildsen, Lars Hornbæk Jensen and Bjørn Mølgård Vester

Aalborg Universitet, DAT4
May 2005

Abstract. In this article we apply the technique of model checking to puzzle games. We develop a tool, *PUzzle checker for autoMAted reachability analysis* (PUMA), to check if puzzles can be solved from some given initial state using reachability analysis. For this purpose we have developed a language to describe models of puzzles and an encoder to translate it into Binary Decision Diagrams representing the model. To combat the state space explosion problem, PUMA uses various heuristics.

1 Introduction

Model checking is a rather new technique which is useful in the formal verification of different types of systems [CCLW99, CAB+98, EOH+93, Low96]. One of the main difficulties is the state space explosion problem, which to some extent can be countered with state space reduction techniques such as *partitioning* [SIJ+02] and *symmetry reduction* [EW05]. We have implemented a model checker (called PUMA), which can compile a one player puzzle game[1] from our own language, called *Input language for PUMA* (InPUMA), into a symbolic representation that can be analysed. This model checker is able to determine whether or not it is possible from the initial state of the puzzle to reach a goal state, i.e. solve a reachability problem. This is done by a symbolic search engine operating on data structures called *Binary Decision Diagrams* (BDDs, introduced by R. E. Bryant in 1986 [Bry86]). To speed up the search, the engine uses various heuristic techniques and optimisations.

Related Work Puzzle game model checking is not a widely explored field, but lends itself to a lot of other proved techniques, namely partitioning [SIJ+02], symmetry reduction [EW05] and sifting [MD03]. Edelkamp [Ede00] describes different heuristic search planning techniques, some of which we have based the symbolic search engine in PUMA on.

[1] A puzzle game contains no randomisation and has perfect knowledge (i.e. there is no hidden information). In solitaire games such as Klondike where some cards lie face down, the model checker still needs to know which cards are where.

Contribution We have implemented a new language to model puzzle games, which should be more intuitive, for puzzle games, than languages in more general model checkers like SPIN [Bel] and UPPAAL [Upp]. From that language we have implemented a compiler that compiles a model in the language into an internal representation based on BDDs using the JDD [Vah] package. From this representation the solvability of the model can then be determined

Outline Section 2 gives background information on BDDs and propositional logic which forms the foundation of the search algorithms and optimisations. Section 3 describes the new InPUMA language that we have developed and the compilation process. Section 4 explores different search techniques, and Section 5 deals with two optimisations: variable ordering (sifting) and partitioning. In Section 6 we look into the effects of the heuristics on two example models.

2 Background

There are two aspects of a puzzle game that we need to model: the states of the puzzle and the rules which describe the legal moves from a given state to another. Both these aspects can be represented using BDDs, which can be manipulated in the same way as formulae from propositional logic.

2.1 Binary Decision Diagrams

A binary decision diagram is a data structure for representing boolean functions. On a more abstract level, BDDs can be considered as a compressed representation of sets or relations.

Definition 1 (Binary Decision Diagram (BDD)). *A Binary Decision Diagram is a rooted, directed acyclic graph* (V, E)*. The set* V *contains one or two terminal vertices* $0, 1 \in V$*. The vertices* $v \in V \backslash \{0, 1\}$ *are non-terminal and have two edges* $low(v)$ *and* $high(v)$*. Each* v *also have a variable* $var(v)$*.*

Given a truth assignment of the variables, the value of the function is determined by traversing the BDD top down. The *high* edge is followed if the variable marking the current node is TRUE, the *low* edge otherwise. The value of the BDD is TRUE if the leaf marked 1 is reached, and FALSE otherwise.

Definition 2 (Ordered Binary Decision Diagram (OBDD)). *A BDD is* ordered *if on all paths through the graph, the variables respect a given total order.*

A BDD is *ordered* if each variable appears at most once on any path from a top node to a terminal node. The size of a BDD is sensitive to the chosen variable ordering. Finding an optimal ordering is a NP-complete problem [GLM03]. We will look into this issue in Section 5.1 on page 12.

Definition 3 (Reduced Ordered Binary Decision Diagram (ROBDD)). *An OBDD is reduced if for all non-terminal vertices v, u the following properties hold:*

1. ***Non-redundant:*** $low(u) \neq high(u)$
2. ***Uniqueness:*** $var(u) = var(v), low(u) = low(v), high(u) = high(v) \Rightarrow u = v.$

An OBDD is *reduced* if every node is non-redundant and unique. The non-redundant criteria makes sure that any node where the high and low edge points to the same node is removed. Uniqueness ensures that common sub-graphs are shared instead of pointing to several identical nodes [And98].

Definition 4 (Characteristic function). *For a set of states S, the characteristic function $\phi_S(a)$ evaluates to* TRUE*, if a is the binary encoding of one state x in S.*

Given a fixed-length binary code for the state space of a planning problem, BDDs can be used to represent the characteristic function of a set of states. Operations on sets are reduced to boolean operations of the characteristic functions:

– Empty set: $\phi_\emptyset = 0$
– Union of sets: $\phi_{S \cup T} = \phi_S \vee \phi_T$
– Intersection of sets: $\phi_{S \cap T} = \phi_S \vee \phi_T$

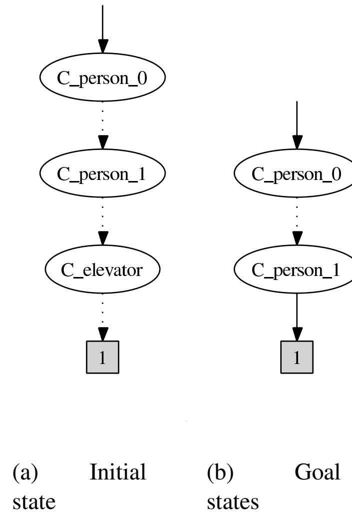
2.2 Propositional Logic

The reachability problem can be represented as a four-tuple (S, T, i, G), where S is a set of states, $T : S \times S$ is a transition relation, where $(s, s') \in T$ iff there is a path leading from s to s'. Variable i is the initial state of the search, and G is the set of goal states. A solution is a sequence of states $\vec{s} = s_0, \ldots, s_n$, where $s_0 = i$, $s_n \in G$ and $\bigwedge_{j=0}^{n-1}(s_j, s_{j+1}) \in T$ [JBV02].

To better understand how a puzzle can be translated into this representation of the reachability problem, we will present a simple example: an elevator is to transport a person from the ground floor to the top. The elevator can be on either the ground or the top floor. This can be represented by a single bit, x_0, which is FALSE if it is on the ground floor and TRUE if it is on the top floor. The person can be either on the ground floor, the top floor, or in the elevator. This requires an encoding length of two bits, x_1 and x_2, which can take the values $\neg x_1 \wedge \neg x_2$, $x_1 \wedge \neg x_2$ and $\neg x_1 \wedge x_2$, respectively. A state in the problem can now be described using an encoding of the variables $\vec{x} = (x_0, x_1, x_2)$.

If both the person and the elevator start at the ground floor, the boolean representation for the initial state i is given by $i = \neg x_0 \wedge \neg x_1 \wedge \neg x_2$. The goal condition is not dependent on the location of the elevator, and the *set* of goal states G is thus represented by $G = x_1 \wedge \neg x_2$. The BDDs representing the initial state and the goal states can be seen in Figure 1 on the following page. These figures have been automatically produced using the PUMA model checker, which is why the naming is a different than the one used here. In these figures, the prefix "C" represents the current variables and "N" the succeeding (new) ones. The zero sink and the edges leading to it has been omitted for aesthetic reasons.

In order to verify that the puzzle is solvable, we need to obtain a sequence of actions, or *transitions*, which transforms the initial state into one which satisfies the goal



(a) Initial state (b) Goal states

Fig. 1: Initial state and goal states of the elevator problem.

condition. These transitions are represented as *relations*, i.e. as sets of tuples of predecessor and successor states. We will use the notation of primed variables to identify the successor state.

Definition 5 (The transition relation). *Let (x, x') be a pair of states where x is the predecessor state of x'. The* transition relation *T is then defined as the disjunction of the characteristic functions of all such pairs.*

In the elevator problem, the rule "move_up" and "move_down" for the elevator is independent of the location of the person and can be formalised using the following functions[2]:

$$T_{move_up} = (\neg x_0 \wedge x'_o) \wedge (x_1 \leftrightarrow x'_1) \wedge (x_2 \leftrightarrow x'_2)$$
$$T_{move_down} = (x_0 \wedge \neg x'_o) \wedge (x_1 \leftrightarrow x'_1) \wedge (x_2 \leftrightarrow x'_2)$$

The rule "enter" and "exit" depends on the position of both the elevator and the person, and does not change the location of the elevator. These rules can be given by the functions:

$$T_{enter} = (x_0 \leftrightarrow x'_0) \wedge (x_0 \leftrightarrow x_1) \wedge \neg x_2 \wedge \neg x'_1 \wedge x'_2$$
$$T_{exit} = (x_0 \leftrightarrow x'_0) \wedge \neg x_1 \wedge x_2 \wedge (x_0 \leftrightarrow x'_1) \wedge \neg x'_2$$

The whole transition relation T for the elevator problem can be seen in Figure 2 on the facing page, and is then given by:

[2] The $\leftrightarrow$ sign is the bi-implication operator.

$$T = T_{move_up} \vee T_{move_down} \vee T_{enter} \vee T_{exit}$$

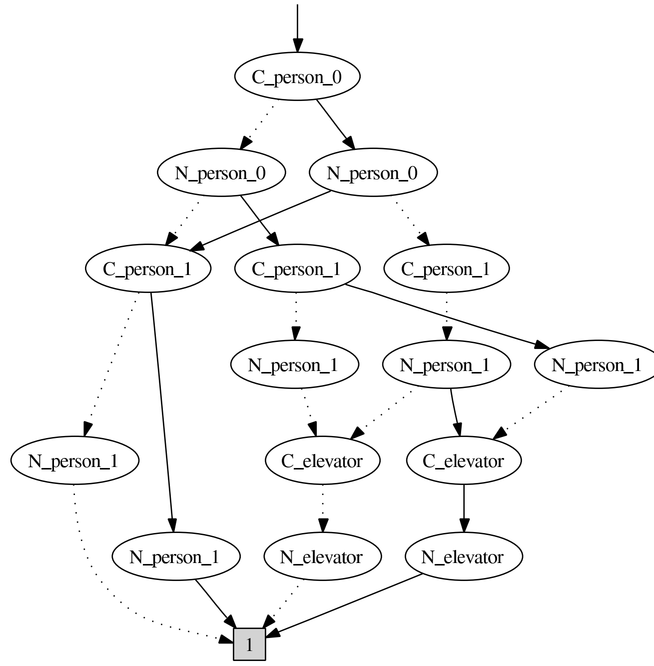


Fig. 2: The transition relation of the elevator problem.

The basic step when calculating the set of successor states is with the use of an expression called the *relational product*. Here, a state $\vec{x}'$ (encoded as a bit vector of boolean variables) belongs to a set S_i if it has a predecessor $\vec{x}$ in the set S_{i-1}, and it can be transformed into $\vec{x}$ using the transition relation T [Ede00].

Definition 6 (Calculation of the successor states). *Let S_n be the set of states reachable from the initial state i in n steps, initialised by $S_0 = i$. The following equation determines ϕ_{S_n} given both $\phi_{S_{n-1}}$ and the transition relation:*

$$\phi_{S_n}(\vec{x}) = (\exists \vec{x}(\phi_{S_{n-1}}(\vec{x}) \wedge T(\vec{x}, \vec{x}')))[\vec{x}/\vec{x}']$$

In the elevator example, the first iteration of finding the successor states from the initial state is given by:

$$\begin{aligned}\phi_{S_1} &= (\exists \vec{x}(\neg x_0 \wedge \neg x_1 \wedge \neg x_2) \wedge T(x_0, x_1, x_2, x'_0, x'_1, x'_2))[\vec{x}/\vec{x}'] \\ &= ((\neg x'_0 \wedge \neg x'_2) \vee (x'_0 \wedge \neg x'_1 \wedge \neg x'_2))[\vec{x}/\vec{x}'] \\ &= (\neg x_0 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge \neg x_2)\end{aligned}$$

3 Input Language

We have designed a language which works as the input to the PUMA model checker. The main goal of the creation of the language was to make it relatively easy and straightforward to model puzzle games. In comparison, we could have used other languages for model checking such as PROMELA/SPIN [Bar] (an open source model checker developed by Bell Labs) or STRIPS/PDDL [FL03] (an attempt at standardising modelling languages for the planning domain). While these are very expressive, they are also very low-level and as a result, not very intuitive to use in the puzzle games domain.

3.1 Syntax

The InPUMA language is very specialised in the sense that it does not include many of the constructs usually associated with programming languages, such as the *if* and *for* constructs. Instead, it is more similar to logic programming languages, and several new constructs has been included to make it easy to describe the rules of the game and to accommodate the differences of each class of puzzle. An example of a puzzle modelled in InPUMA can be seen in Figure 3. The complete grammar of InPUMA can be found on the project home page [EJV] along with several models of puzzle games, such as Rubik's Cube, FreeCell, Peg Solitaire, Sokoban and others. A shortened extract from the grammar in *EBNF* for our language is given in Figure 4 on the facing page.

```
1  Init {
2      int(2) person = 0;          // Ground floor
3      int(1) elevator = 0;        // Ground floor
4  }
5
6  Goals {
7      Goal(person == 1);          // Person is on Top floor
8  }
9
10 Rules {
11      Rule(person == elevator) { // Let person in
12          person = 2;            // Person is in elevator
13      }
14
15      Rule(person == 2) {        // Let person out
16          person = elevator;     // Person is on same floor as elevator
17      }
18
19      Rule(elevator == 0) {      // Move elevator up
20          elevator = 1;
21      }
22
23      Rule(elevator == 1) {      // Move elevator down
24          elevator = 0;
25      }
26 }
```

Fig. 3: The elevator problem modelled in InPUMA.

```
Puzzle      ::= init-block goals-block rules-block

init-block  ::= 'Init' '{' { statement } '}'
goals-block ::= 'Goals' '{' { goal } '}'
rules-block ::= 'Rules' '{' { pick-decl | rule } '}'

goal        ::= 'Goal' '(' boolean-exp ')' ';'
rule        ::= 'Rule' '(' boolean-exp ')' '{' { assignment } '}'
statement   ::= variable-decl | structure-decl | assignment
pick-decl   ::= 'reference' variable '=' 'pick' '(' exp-list ')' ';'
```

Fig. 4: A simplified grammar of InPUMA

A puzzle written in InPUMA is divided into three blocks: *init*, *goal* and *rules*. Below is a description of what each of these does:

Init: This block sets up the initial state of the puzzle game. Only variable declarations, structure definitions (which are user-defined composite types) and assignments are allowed here.

Goals: This block defines the goal conditions, and only the `goal` statement are allowed here. A `goal` has a boolean expression which evaluates to TRUE whenever the goal conditions has been met, and FALSE otherwise.

Rules: This block defines the transitions between a given state and the states reachable according to the rules of the puzzle. Only the `rule` and `pick` statements are allowed here. A `rule` statement consists of a boolean expression and a number of variable assignments. If the expression evaluates to TRUE (the rule is applicable), the assignments describes how the state changes.

A `pick` statement is a variable declaration, assigned with a value, which is chosen non-deterministic among a given set of values. This set consists of comma-separated values, where integer ranges may be given with the "`..`" keyword. For instance, if we want to pick a cell in a 5×5 array we could construct two `pick` statements like this:

```
reference rowNumber    = pick(0 .. 4);
reference columnNumber = pick(0 .. 4);
```

3.2 Compilation

Internally, the model checker uses boolean logic both to represent the state space and the transition relations. We have created a compiler for the InPUMA language, which build these internal structures. There are four main BDDs which must be constructed:

varSpace :	Represents the variables which can be manipulated in the rules.
start :	Represents the starting configuration of the variables in the state space.
goal :	Represents the goal configuration of the variables in the state space.
transRel :	Represents all rules as one transition relation.

We will break the compilation into three parts, corresponding to the three blocks of the program:

Init For each variable declaration, a number of BDD variables is added to the **varSpace** BDD, according to the number of bits needed to represent the variable.

Assignments in the init block are transformed to BDDs, combined using the $\wedge$ operator and stored in the **start** BDD. For instance, the assignment "a = 2;", where "a" is an integer of 3 bits, the following BDD is created: "$\neg a_2 \wedge a_1 \wedge \neg a_0$"

An issue when compiling InPUMA into boolean logic is that variables must be of a finite domain. One could argue that a type such as the integer is usually represented by a fixed length of 32 bits and is thus finite. This, however, might not be a very efficient solution, in the case of integer variables, the programmer has the possibility of specifying the number of bits, such as "`int(3) smallInt`'.

Finishing the compilation of the init block, the compiler checks that all variables have been instantiated. All variables in the state space is then duplicated and stored for use in the transition relation.

Goals The goal expressions are transformed to BDDs, combined using the $\wedge$ operator, and stored in the **goal** BDD. The boolean expressions &&, || and ! can be translated almost directly. The equality operator == is translated using the bidirectional operator $\leftrightarrow$. For instance, "a == 2" is translated to "$(a_2 \leftrightarrow \textsc{false}) \wedge (a_1 \leftrightarrow \textsc{true}) \wedge (a_0 \leftrightarrow \textsc{false})$", where "FALSE, TRUE, FALSE" is the binary encoding of the integer "2". Since this expression is created using constants, the resulting BDD can be reduced to "$\neg a_2 \wedge a_1 \wedge \neg a_0$". Notice that this is the same BDD as the assignment operator in the init block.

Rules The rules of the puzzle is already somewhat similar to the transition relations of boolean logic, since they consist of a number of prerequisites and a number of new variable assignments. This is to some extent what we used in the notion $(\vec{x}, \vec{x}')$ in the elevator example from Section 2.2 on page 3 about propositional logic. The assignments in a rule represent the new values of the variables in the new state. All BDDs which are created from the rules are finally combined using the $\vee$ operator, and then stored in the **transRel** BDD.

The rules may need to be expanded if any "`pick`" variables are given. For each rule which includes a "`pick`" variable, this rule is expanded to several new rules – one rule for each value that the variable may assume.

4 State Space Exploration

For the symbolic exploration, the set of states is combined with the transition relation. By querying for all instances of $\vec{x}'$ we find all states which are reachable from the input set $\vec{x}$. Starting from the initial state, each iteration of search algorithms explores the previous search horizon until eventually the whole set of reachable states has been covered. In the following sections we presents the algorithms used for the symbolic search in PUMA.

4.1 Breadth-First Search

A breadth-first search algorithm explores the frontier between discovered and undiscovered states uniformly across the breadth of the frontier. In the following we apply this algorithm to the symbolic domain.

We will let *Open* be the set of states which marks the search horizon, and *Succ* be the set of successor states after one iteration of the algorithm. The search is terminated when the set *Open* contains a state in ϕ_G, as illustrated in Figure 5. This is constructed by testing the equivalence of the intersection of these two sets, with the trivial zero function (the empty set). The final breadth-first search algorithm is shown in Figure 6.

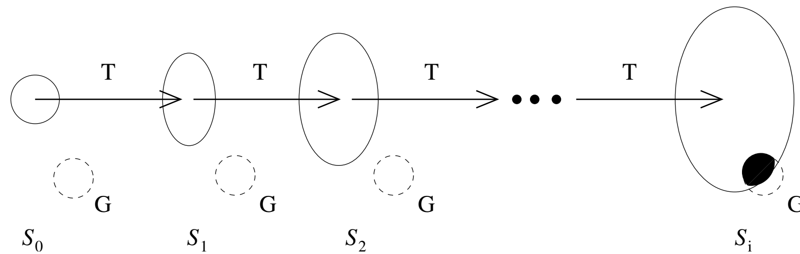


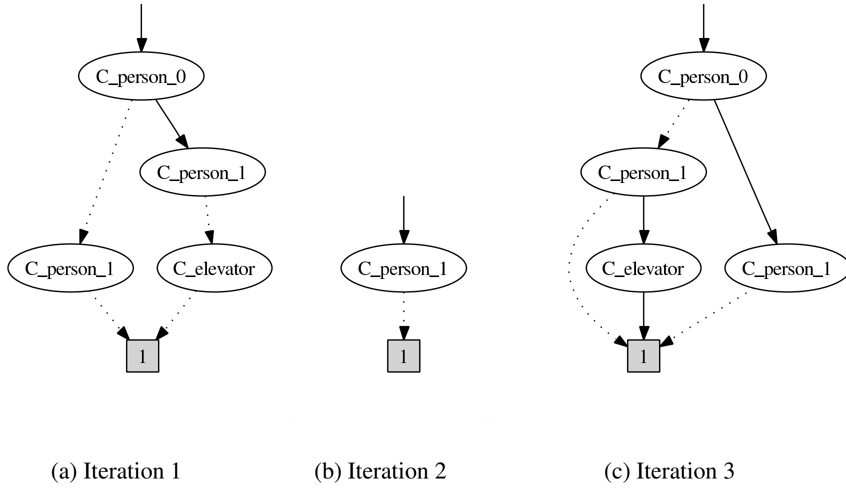
Fig. 5: An example of a forward pass of the algorithm. After i steps, the sets S_i and G overlap.

Breadth-First-Search

1 $Open \leftarrow \phi_{\{s\}}$
2 **do**
3 $\quad Succ \leftarrow \exists x(Open(x) \wedge T(x, x'))$
4 $\quad Open \leftarrow Succ[x/x']$
5 **while** $(Open \wedge \phi_G \equiv 0)$

Fig. 6: Pseudo code for the breadth-first search algorithm.

When applying this algorithm to the BDDs we constructed as an example in Section 2.2 on page 3, the algorithm will terminate after three iterations. The BDDs representing the *Open* set after each iteration can be seen in Figure 7. As the figures indicate, the symbolic representation for a large set of states is typically smaller than the cardinality of the represented set.



(a) Iteration 1 (b) Iteration 2 (c) Iteration 3

Fig. 7: Search iterations of the elevator problem.

4.2 Bidirectional Search

Since we have the goal states encoded as a BDD, just as the initial state, and T has been defined as a relation, we can take advantage of these facts by searching backwards instead. That is, we start with the goal set and iterate until we encounter the start state. The formula for finding these successive states is similar to the forward search, except the existential quantification is now over the primed (successor) variables:

$$\phi_{B_i}(\vec{x}) = \exists \vec{x}'((\phi_{B_{i-1}}(\vec{x})[\vec{x}'/\vec{x}]) \land T(\vec{x}, \vec{x}'))$$

In bidirectional breadth-first search, forward and backward search are carried out concurrently. We keep track of both the forward search frontier F_f (which initially is $F_0 = s$) and the backward search frontier B_b (which initially is $B_0 = G$). When the two frontiers intersect ($\phi_{F_f} \land \phi_{B_b} \not\equiv 0$), we have found the optimal solution of length $f + b$. These two search horizons are stored in *fOpen* and *bOpen* and the function *forward*() returns TRUE or FALSE according to which frontier should be explored for the current iteration. The bidirectional version of the breadth-first search algorithm is shown in Figure 8 on the facing page.

```
BIDIRECTIONAL-BREADTH-FIRST-SEARCH()
 1  fOpen ← φ_{s}
 2  bOpen ← φ_G
 3  do
 4     if (forward())
 5          Succ ← ∃x(fOpen(x) ∧ T(x, x′))
 6          fOpen ← Succ[x/x′]
 7     else
 8          succ ← bOpen[x′/x]
 9          bOpen ← ∃x′(bOpen(x′) ∧ T(x, x′))
10  while (fOpen ∧ bOpen ≡ 0)
```

Fig. 8: Pseudo code for the bidirectional breadth first search.

The *forward*() function does not simply alternate equally between TRUE and FALSE. The optimisation comes from the fact that we can select the search direction according to certain criteria such as BDD size, the number of states encoded, or the time spent on the last exploration step in that direction.

4.3 Forward Set Simplification

In order to reduce the number of states in the search frontier, we will introduce a set called *Closed* containing all previously expanded states. This both avoids exploring the same states more than once, and, more importantly, ensures that the search terminates in case of validation failure. In ordinary memory structures, this is commonly implemented as a hash table, referred to as a *transposition table*, whereas in symbolic search it is called *forward set simplification*. This technique, applied to the breadth-first search algorithm from Figure 6 on page 9, can be seen in Figure 9.

```
FORWARD-SET-SIMPLIFICATION
1  closed ← open ← φ_{s}
2  do
3     Succ ← ∃x(Open(x) ∧ T(x, x′))[x/x′]
4     Open ← Succ ∧¬ Closed
5     Closed ← Closed ∨ Succ
6  while (Open ∧φ_G ≡ 0)
```

Fig. 9: Pseudo code for a search with the forward set simplification technique.

5 Optimisations

In the following sections we will look at two optimisations for the symbolic search. An algorithm for dynamic variable ordering and a partitioning technique.

5.1 Sifting

There are two main ways of doing variable ordering: static and dynamic. Static ordering determines an order for the variables at the beginning of the process and never changes it. Dynamic ordering attempts to optimise the ordering between each operation on the BDD.

One such algorithm is called sifting [Rud93]. It is based on finding the optimal position for a variable assuming the other positions are fixed. For a BDD with n variables leads to n possible positions to try including the current one. The goal is then to find the position that minimises the size of the BDD. This is done by swapping the variable with an adjacent variable up and down the BDD until all positions have been tried, and then swapping it back to the best position.

5.2 Partitioning

In the worst case scenario, the size of a BDD may grow exponentially with the number of variables. In these situations it may be advantageous to partition the BDD into smaller parts. Especially in the successor/predecessor computation, the intermediate BDDs tend to be large compared to the BDD representing the result [JBV02]. A technique to avoid this problem is partitioning of the transition relation. Partitioning can be in either disjunctive or conjunctive form, where each partition T_i is implicitly combined using the $\vee$ or the $\wedge$ operator, respectively, to form the full transition relation. The relational product, as defined in Section 2.2 on page 3 can then be optimised using these partitions, without ever constructing the BDD for the full transition relation.

To make a disjunctive partitioning, the transition relation is partitioned according to what variables is modified. If $\vec{x_i}$ is the variables which are updated in T_i, the relational product computation with a disjunctive partitioned relation, is on the form:

$$\begin{aligned}
\phi_{S_n}(\vec{x}) &= (\exists\vec{x}(\phi_{S_{n-1}}(\vec{x}) \wedge (T_1(\vec{x},\vec{x}') \vee \cdots \vee (T_j(\vec{x},\vec{x}'))))[\vec{x}/\vec{x}'] \\
&= (\exists\vec{x_1}\,(\phi_{S_{n-1}}(\vec{x_1}) \wedge T_1(\vec{x_1},\vec{x_1}\,')))[\vec{x_1}/\vec{x_1}\,'] \\
&\vee \ldots \\
&\vee (\exists\vec{x_j}\,(\phi_{S_{n-1}}(\vec{x_j}) \wedge T_j(\vec{x_j},\vec{x_j}\,')))[\vec{x_j}/\vec{x_j}\,']
\end{aligned}$$

This will reduce the problem of computing ϕ_{S_n} to one of computing a series of relational products involving smaller BDDs. The complexity of the successor computation depends on the number of partitions. For this reason, the best performance is often obtained by merging some of the partitions according to an upper bound on the size of the BDD representing a partition. In particular, partitions with strongly interacting components are good candidates for merging [Yan99].

6 Experimental Results

We have implemented the PUMA model checker in Java using a BDD package called JDD [Vah]. PUMA consists of a compiler from the InPUMA language to an internal representation based on BDDs, a graphical user interface and a symbolic search engine which implements the following techniques:

- Breadth-first search
- Forward set simplification
- Bi-directional search
  - Direction chosen from BDD sizes
  - Direction chosen from shortest time spend last on each direction

The JDD package is built on the BUDDY [LN] interface, but does not have support for variable reordering. Therefore sifting and partitioning has not been implemented.

We have tested PUMA on models of puzzle games and are successfully able to compute whether or not they are solvable. In the following two sections we will examine two puzzles and the effect of the heuristics: bidirectional search and forward set simplification on those.

6.1 Lights Out

The Lights Out puzzle consists of a grid of cells, which have lights in them. By toggling a switch inside a cell, the lights in the cell, and in the non-diagonally adjacent cells, change to the opposite of what they were. The puzzle start with all lights off and the goal is to light them all. The model of the 5×5 variant is shown in Figure 12 on page 19 in the appendix. This version has 2^{23} different combinations reachable from the initial configuration [Sch].

In Figure 10(a) on the following page we see the size of the *Open* BDD at the different iterations of the search algorithm, using none of the heuristics. Lights Out is an example of a model where the backwards BDD starts in an equivalent situation as the *Open* BDD, only with all values inverted. Because of this, Lights out benefits highly from using bidirectional search as seen in Figure 10(b) on the following page. In fact using bidirectional search will never have a negative effect, but is more beneficial in situations where the sizes of the backwards and forwards BDD expand at a similar rate.

6.2 Peg Solitaire

The Peg solitaire puzzle consists of a board with holes. The puzzle starts with pegs in each hole except the middle one. The goal is to remove pegs by jumping with a peg over another peg. The peg which has been jumped over is removed. The game is won when only one peg remains. An unsolvable 5×5 variant is shown in Figure 13 on page 20 in the appendix.

In Figure 11(a) on page 15 we see the size of the *Open* BDD at the different iterations of the search algorithm. In this case using none of the heuristics. In contrast to the Lights Out search, the *Open* BDD continues to grow with the number of represented

Iteration	f_Open
0	0
1	316
2	1.906
3	6.827
4	17.120
5	44.559
6	123.695
7	272.700
8	438.909
9	522.420
10	502.910
11	378.820
12	192.615
13	64.395
14	15.093
15	69

(a)

Iteration	Direction	f_Open	b_Open
0	-	0	0
1	f	316	0
2	b	316	316
3	f	1.906	316
4	b	1.906	1.906
5	f	6.827	1.906
6	b	6.827	6.827
7	f	17.125	6.827
8	b	17.120	17.120
9	f	44.559	17.120
10	b	44.559	44.559
11	f	123.695	44.559
12	b	123.695	123.695
13	f	272.700	123.695
14	b	272.700	272.700
15	f	438.909	272.700

(b)

Fig. 10: (a): Lights Out 5×5: Solved in 492 sec. (b): Lights Out 5×5: Using bidirectional and solved in 118 sec.

states, until a point where the size remain almost constant. The forward set simplification technique lends itself well to this situation because it reduces the number of states represented by the BDD, which in turn decreases the size. Figure 11(b) shows the iterations if this technique is applied. If used in the Lights Out situation we saw before, we would force the BDD into a situation where it might represent fewer states, but the actual BDD representing it is larger.

Iteration	f_Open
0	0
1	74
2	241
3	549
4	1.354
5	3.096
6	6.781
7	13.717
8	24.961
9	40.422
10	57.945
11	73.693
12	84.449
13	90.085
14	91.685
15	90.652
16	88.506
17	86.428
18	84.913
19	83.985
20	83.616
21	83.465
22	83.415

(a)

Iteration	f_Open	f_closed
0	0	0
1	62	74
2	220	241
3	499	549
4	1.222	1.354
5	2.731	3.096
6	5.674	6.781
7	10.638	13.717
8	17.941	24.961
9	26.193	40.422
10	32.487	57.945
11	33.547	73.693
12	28.533	84.449
13	22.299	90.085
14	16.482	91.685
15	11.237	90.652
16	7.007	88.506
17	4.065	86.428
18	2.190	84.913
19	1.093	83.985
20	553	83.616
21	270	83.465
22	140	83.415

(b)

Fig. 11: (a): Peg Solitaire 5×5: Proved non-solvable in 19,4 sec. (b): Peg Solitaire 5×5: Using forward set simplification and proved non-solvable in 11,8 sec.

7 Conclusion

In this article we have developed a language to describe models of puzzles and an encoder to translate it into Binary Decision Diagrams. In addition, a symbolic model checker for the encoded puzzles was implemented using bidirectional search and forward set simplification. In the experimental results we found that in some scenarios, these two heuristics reduced the search times by several orders of magnitude, whereas in others they did not have any effect, or even increased the search time.

8 Future Work

One feature we have not looked at in this article, which could help combat the state space explosion problem, is how to exploit symmetry in a puzzle. The state space is reduced by considering states which are equivalent, for instance by permutations which interchange the identities of some of its components. This equivalence is commonly known as the *orbit relation* and gives rise to a bisimilar structure over the equivalence classes [EW05]. The InPUMA language would have to be extended in order of specifying which components are symmetrical.

Another path worth examining would be to replace the BDD package. An interesting package is JavaBDD [Wha] which has support for variable reordering, and may interface directly to the native BuDDy library (written in C) [LN].

References

[And98] Henrik Reif Andersen. An introduction to binary decision diagrams. 1998. Lecture notes for 49285 Advanced Algorithms E97.

[Bar] Ian Barland. Promela and spin reference. http://cnx.rice.edu/content/m12318/latest/.

[Bel] Bell Labs. SPIN. http://www.spinroot.com.

[Bry86] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[CAB+98] W. Chan, R. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. Reese. Model checking large software specifications, 1998.

[CCLW99] P. Chauhan, E. Clarke, Y. Lu, and D. Wang. Verifying ip-core based system-on-chip design, 1999.

[Ede00] Stefan Edelkamp. Heuristic Search Planning with BDDs. Berlin, Humboldt University, 2000. Submitted to the 14th Workshop on New Results in Planning, Scheduling and Design (PUK).

[EJV] Jacob Eskildsen, Lars Hornbæk Jensen, and Bjørn Mølgård Vester. Project home page. http://www.LarsHJ.dk/PUMA.

[EOH+93] E.M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D.E. Long, K.L. McMillan, and L.A. Ness. Verification of the Futurebus+ Cache Coherence Protocol. In D. Agnew, L. Claesen, and R. Camposano, editors, *The Eleventh International Symposium on Computer Hardware Description Languages and their Applications*, pages 5–20, Ottawa, Canada, 1993. Elsevier Science Publishers B.V., Amsterdam, Netherland.

[EW05] E. Allen Emerson and Thomas Wall. Dynamic symmetry reduction. *N. Halbwachs and L. Zuck (Eds.): TACAS 2005, LNCS 3440*, pages 382–396, 2005.

[FL03] Maria Fox and Derek Long. pddl2.1 : An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, December 2003.

[GLM03] Orna Grumberg, Shlomi Livne, and Shaul Markovitch. Learning to order bdd variables in verification. *Journal of Artificial Intelligence Research*, 18:83–116, 2003.

[JBV02] Rune M. Jensen, Randy E. Bryant, and Manuela M. Veloso. An efficient BDD-based A* algorithm. In *Proceedings of AIPS'02 Workshop on Planning via Model Checking*, Toulouse, April 2002.

[LN] Jørn Lind-Nielsen. Buddy. http://sourceforge.net/projects/buddy.

[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1055, pages 147–166. Springer-Verlag, Berlin Germany, 1996.

[MD03] D. Michael Miller and Rolf Drechsler. Augmented sifting of multiple-valued decision diagrams. *ismvl,33rd International Symposium on Multiple-Valued Logic*, 33:375, May 2003.

[Rud93] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press.

[Sch] Jaap Scherphuis. Jaap's puzzle page - lights out. `http://www.geocities.com/jaapsch/puzzles/lights.htm`.

[SIJ+02] Debashis Sahoo, Subramanian Iyer, Jawahar Jain, Chriastian Stangier, Amit Narayan, Davis L. Dill, and E. Allen Emerson. A partitioning methodology for bdd-based verification. 2002.

[Upp] Uppsala University and Aalborg University. UPPAAL. `http://www.uppaal.com`.

[Vah] Arash Vahidi. Jdd, a pure java bdd and z-bdd library. `http://javaddlib.sourceforge.net/jdd`.

[Wha] John Whaley. JavaBDD. `http://javabdd.sourceforge.net`.

[Yan99] Bwolen Yang. *Optimizing model checking based on bdd characterization*. PhD thesis, 1999. Chair-David R. O'Hallaron.

Appendix A
Puzzle Models

```
1  Init {
2      boolean[5][5] board; // 5x5 board
3      board.fill(false);   // Assign "false" to all cells
4  }
5
6  Goals {
7      // All cells in the array must have the value "true"
8      Goal(board.allEquals(true));
9  }
10
11 Rules {
12     reference p1 = pick(0..4); // Pick a row number
13     reference p2 = pick(0..4); // Pick a column number
14
15     Rule (true) { // This rule is always applicable
16         // Switch the boolean value of the cell and its neighbours
17         board[p1][p2] = !board[p1][p2];
18         board[p1 + 1][p2] = !board[p1 + 1][p2];
19         board[p1 - 1][p2] = !board[p1 - 1][p2];
20         board[p1][p2 + 1] = !board[p1][p2 + 1];
21         board[p1][p2 - 1] = !board[p1][p2 - 1];
22     }
23 }
```

Fig. 12: The Lights Out puzzle modeled in InPUMA.

```
Init {
        //      0 1 2 3 4
        //   0 o o o o o
        //   1 o o o o o
        //   2 o o E o o
        //   3 o o o o o
        //   4 o o o o o

        boolean[5][5] board; // false == empty, true == peg
        board.fill(true);     // Fill with pegs
        board[2][2] = false; // Empty
        int(5) pegs = 24;     // Number of pegs
}

Goals {
        Goal(pegs == 1);
}

Rules {
        reference p1 = pick(0..4);
        reference p2 = pick(0..4);

        Rule( // Try to move right
              board[p1][p2] == true &&
              board[p1 + 1][p2] == true &&
              board[p1 + 2][p2] == false
            ) {
                   board[p1][p2] = false;
                   board[p1 + 1][p2] = false;
                   board[p1 + 2][p2] = true;
                   pegs = pegs - 1;
              }

        Rule( // Try to move left
              board[p1][p2] == true &&
              board[p1 - 1][p2] == true &&
              board[p1 - 2][p2] == false
            ) {
                   board[p1][p2] = false;
                   board[p1 - 1][p2] = false;
                   board[p1 - 2][p2] = true;
                   pegs = pegs - 1;
              }

        Rule( // Try to move up
              board[p1][p2] == true &&
              board[p1][p2 - 1] == true &&
              board[p1][p2 - 2] == false
            ) {
                   board[p1][p2] = false;
                   board[p1][p2 - 1] = false;
                   board[p1][p2 - 2] = true;
                   pegs = pegs - 1;
              }

        Rule( // Try to move down
              board[p1][p2] == true &&
              board[p1][p2 + 1] == true &&
              board[p1][p2 + 2] == false
            ) {
                   board[p1][p2] = false;
                   board[p1][p2 + 1] = false;
                   board[p1][p2 + 2] = true;
                   pegs = pegs - 1;
              }
}
```

Fig. 13: The 5x5 Peg Solitaire puzzle modeled in InPUMA.