DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Bounding Transition Systems' Diameters Using QBF

Alexander Schlenga

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Bounding Transition Systems' Diameters Using QBF

Obere Schranken für Durchmesser von Transitionssystemen mittels QBF

Author:	Alexander Schlenga
Supervisor:	Prof. Tobias Nipkow, Ph.D.
Advisor:	Mohammad Abdulaziz, Ph.D.
Submission Date:	15 September 2021

I confirm that this bachelor’s thesis is my own work and I have documented all sources and material used.

Karlsruhe, 15 September 2021

Alexander Schlenga

Acknowledgments

First, I would like to thank Prof. Tobias Nipkow for giving me the opportunity to write this thesis under his supervision.

Next, I would like to express my big gratitude to Mohammad Abdulaziz who not only advised me on technical questions but was a motivating mentor to me, gave me insights into the academic world and was always up for little personal chats that made everything more relaxed.

Last but not least, I want to thank my girlfriend and my family for supporting me the whole time, especially my father, who took the time to read and comment on my draft although he had enough other things to do.

Abstract

We talk about the problem of bounding transition systems which occurs in planning and model checking. It consists of answering the question how long transition sequences fulfilling certain criteria can be at maximum. The approach of compositional bounding splits a transition system into smaller parts, which then are bounded using base case functions, and composed to compute a total bound of the system. We use the recurrence diameter and the sublist diameter as base case functions. They are both properties of transition systems but impose bounds of different tightness on them. For the recurrence diameter, we give a SAT encoding and for the sublist diameter we give a QBF encoding. Both encodings are implemented and evaluated in Standard ML.

Contents

Acknowledgments **iii**

Abstract **iv**

1 Introduction **1**

1.1 Summary of the results . . . 1

1.2 Planning . . . 1

1.3 Balls in Boxes—a running example . . . 2

1.4 Compositional bounding . . . 4

1.5 Related work . . . 4

1.6 Terms and definitions . . . 4

1.6.1 STRIPS . . . 5

1.6.2 SAS+ . . . 6

2 SAT encoding of the recurrence diameter **8**

2.1 The recurrence diameter . . . 8

2.2 From SAS+ to STRIPS . . . 8

2.3 The encoding . . . 9

2.4 Implementation . . . 11

2.4.1 Gaining performance using functional fusion . . . 13

2.5 Experimental results . . . 14

3 QBF encoding of the sublist diameter **21**

3.1 Quantified Boolean Formulae . . . 21

3.2 The sublist diameter . . . 21

3.3 The encoding . . . 23

3.4 Implementation . . . 25

3.5 Experimental results . . . 26

4 Conclusion **29**

4.1 Future work . . . 29

List of Figures **30**

List of Tables **31**

Bibliography **32**

1 Introduction

Artificial intelligence is a topic that currently gains importance in informatics but also in society in general. One of the main research areas in artificial intelligence is *planning*. Often, there occur scenarios in which a complete plan to a problem is not needed or not feasible, and one is interested in a bound on the length of a plan.

More generally, we can obtain bounds on *transition systems*, which are a common way to describe planning problems. But transition systems can also describe *model checking* (Biere, Cimatti, Clarke, & Zhu, 1999).

This thesis deals with two different techniques for computing upper bounds on lengths of sequences in transition systems. They are both based on compositional bounding but use different base case functions, i.e., methods to bound the smaller instances:

- The recurrence diameter, encoded as a *propositional satisfiability problem (SAT)*
- The sublist diameter, encoded in a *quantified boolean formula (QBF)*

They are evaluated with respect to an earlier encoding of the recurrence diameter in *satisfiability modulo theories (SMT)*.

1.1 Summary of the results

With the SAT-based encoding of the recurrence diameter we were able to gain performance in some problem domains with respect to the SMT-based encoding. The QBF encoding of the sublist diameter, instead, is not yet suitable for practical bounding as the running times of QBF solvers are still too long.

1.2 Planning

Planning (sometimes also called automated planning) is a branch of artificial intelligence dealing with problems in which the main objective is to synthesize a *plan* that brings a system from one state into another. A typical planning problem can be described as:

- A transition system, consisting of

    - A world of possible states
    - Actions that can be executed and lead from one state to another

- An initial state
- A goal condition fulfilled by some of the states

(Ghallab, Nau, & Traverso, 2004, pp. 5–7)

Though there exist many different forms of planning, we focus on *classical planning* here because it is what the core of planning is about and can serve as a basis for other, more complex forms of planning. This most importantly means it is (Ghallab, Nau, & Traverso, 2004, p. 17):

Finite
The number of states is finite

Implicit time
Time is not considered explicitly, the system just goes from one state to the next

Deterministic
There is no random component, everything influencing the system is known in advance

Fully observable
There is no unknown component, we can always know every aspect of the system

1.3 Balls in Boxes—a running example

Consider the following scenario. There are two adjacent boxes, A and B. Furthermore, there are three balls, a red one, a green one, and a blue one. Every ball is always in one of the two boxes. There is some agent (a human or a robot) who can take the balls from one of the boxes and put them into the other. We consider the act of taking a ball and putting it into the other box as an atomic action. The position of a ball inside a box does not matter. This already gives us a transition system. Its eight states are depicted in Figure 1.1.

It is easier though, not to reference the states with images every time. Therefore we introduce the following notation: The three balls are denoted by the first letter of their color, i.e., "R"(ed), "G"(reen) and "B"(lue). The wall separating the two boxes is denoted by " | ". So the state where the red and the green ball are in the first box and the blue ball is in the second box is written as "RG | B".

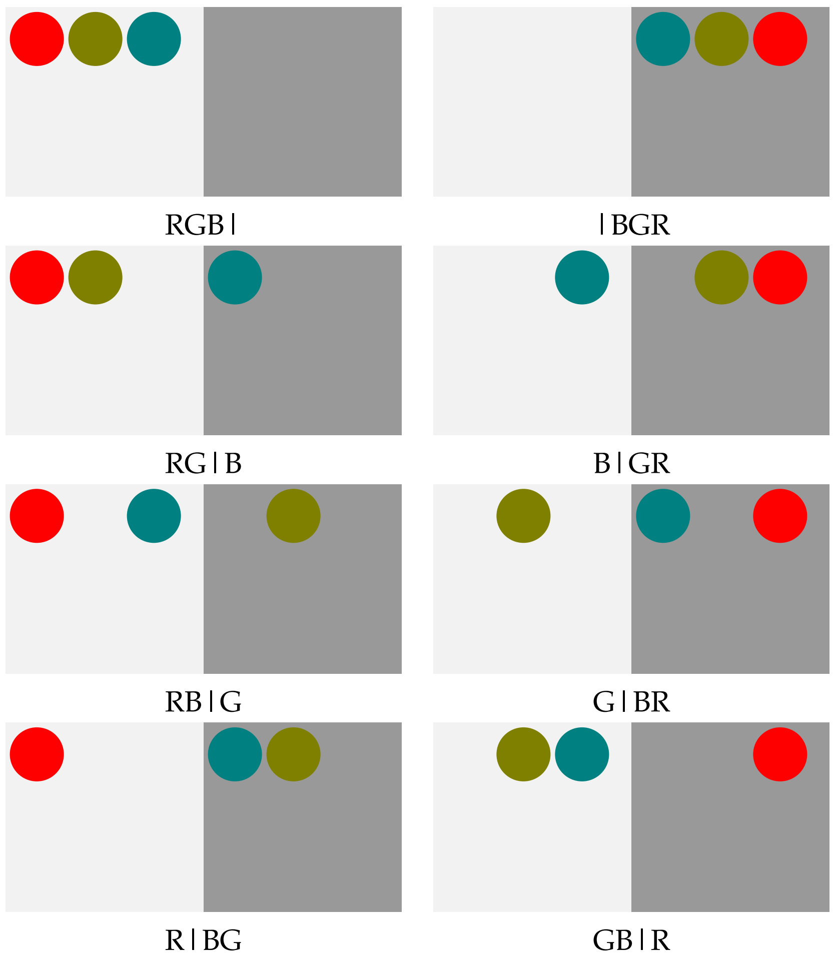

Figure 1.1: States of Balls in Boxes

The actions are all of the same kind. They move a specific ball from a specific box to the other box. We write “R ->” to describe the action taking the red ball from the first box and putting it into the second. Accordingly, “G <-” moves the green ball from the second to the first box.

Figure 1.2 shows the complete transition system as a graph. You can see the transition system is completely symmetric as moving balls always is possible in both directions and there is no one-way action like destroying a ball. Additionally, there are some symmetries between the states and corresponding automorphisms on the graph as, e.g., the two boxes are practically interchangeable.

We can now pose a planning problem e.g. by saying that the initial state is RGB |

and the goal condition is (informally) “The green ball must be in the second box and the red ball must be in a different box than the blue ball”. A plan solving the problem would be, e.g., the action sequence R ->, G ->.

1.4 Compositional bounding

To talk about the *length* of a plan, we restrict ourselves to sequential plans here. This means that no actions are to be executed in parallel. Instead, their execution has a total order on it.

Normally, shorter plans are what we want. Sometimes, there’s also a restriction on how long a plan may be to be usable at all and therefore one is interested in an upper bound on the length of a possible plan. This makes one motivation for upper bounding plan lengths. A second one is, that upper bounds can be used as thresholds for SAT-based planning: The question whether a planning problem has a solution of a given length is encoded in SAT, and the search for a solution can be stopped when the bound is reached. This use of the upper bounds also appears in *Bounded Model Checking*, where bounds can be used as completeness thresholds.

The compositional approach to bounding transition systems constitutes in computing bounds by partitioning a transition system into smaller abstractions. These abstractions are then passed to subroutines that compute bounds on them by means of a base case function. Here, we use the Hyb (Abdulaziz, Gretton, & Norrish, n.d.) algorithm/ program.

1.5 Related work

This work builds up on the compositional bounding techniques devised by Baumgartner, Kuehlmann, and Abraham, 2002, which later have been improved in various ways. The abstractions used first were projections. Abdulaziz, Gretton, and Norrish, 2015, first proposed the sublist diameter as a base case function, which is also used in the second part of this thesis. Abdulaziz, Gretton, and Norrish, 2017, introduced snapshots as another type of abstraction of factored transition systems, besides projections. Abdulaziz and Berger, 2021, gave an efficient encoding of the recurrence diameter and passed it to an SMT solver.

1.6 Terms and definitions

The following definitions are taken from Abdulaziz and Berger, 2021, and adapted.

1.6.1 STRIPS

STRIPS is a way of modeling planning problems only with propositional variables.

Definition 1 (STRIPS states and actions). *A STRIPS maplet $v \mapsto b$ maps a variable v—i.e. a state-characterizing proposition—to a boolean value b. A STRIPS state x is a finite set of STRIPS maplets where a variable is never mapped to more than one value. We write $\mathcal{D}(x)$ to denote $\{v \mid (v \mapsto b) \in x\}$, the domain of x, and $x(v)$ to denote the value that v is mapped to in x. For states x_1 and x_2, the union $x_1 \uplus x_2$ is defined as $\{v \mapsto b \mid v \in \mathcal{D}(x_1) \cup \mathcal{D}(x_2) \wedge \text{if}\, v \in \mathcal{D}(x_1)\, \text{then}\, b = x_1(v)\, \text{else}\, b = x_2(v)\}$. Note that the state x_1 takes precedence. An action is a pair of states (p, e) where p represents the* preconditions *and e represents the* effects. *For an action $\pi = (p, e)$, the domain of that action is $\mathcal{D}(\pi) \equiv \mathcal{D}(p) \cup \mathcal{D}(e)$.*

For the following definitions in this section, we do not write "STRIPS" explicitly anymore.

Definition 2 (Execution). *When an action $\pi(= (p, e))$ is executed at state x, it produces a successor state $\pi(x)$, formally defined as $\pi(x) = \text{if}\, p \not\subseteq x\, \text{then}\, x\, \text{else}\, e \uplus x$. We lift execution to lists of actions $\overrightarrow{\pi}$, so $\overrightarrow{\pi}(x)$ denotes the state resulting from successively applying each action from $\overrightarrow{\pi}$ in turn, starting at x.*

A *factored transition system* is a concise representation of a transition system. Rather than explicitly giving all states and enumerating all transitions between them, we write down a set of actions with preconditions and effects, where the preconditions only impose requirements on *some* variables, and the effects only influence *some* variables. This way, one action expresses many state transitions.

Definition 3 (Factored transition system). *A set of actions δ constitutes a factored transition system. $\mathcal{D}(\delta)$ denotes the domain of δ, which is the union of the domains of all the actions in δ. Let $\text{set}(\overrightarrow{\pi})$ be the set of elements in $\overrightarrow{\pi}$. The set of valid action sequences δ^* is $\{\overrightarrow{\pi} \mid \text{set}(\overrightarrow{\pi}) \subseteq \delta\}$. The set of valid states $\mathbb{U}(\delta)$ is $\{x \mid \mathcal{D}(x) = \mathcal{D}(\delta)\}$. $G(\delta)$ denotes the set of pairs $\{(x, \pi(x)) \mid x \in \mathbb{U}(\delta), \pi \in \delta\}$, which is all non self-looping transitions in the state space of δ.*

Definition 4 (Diameter). *The diameter of a transition system, written $d(\delta)$, is the length of the longest shortest action sequence, formally*

$$\mathrm{d}(\delta) = \max_{\substack{x \in \mathbb{U}(\delta) \\ \overrightarrow{\pi} \in \delta^*}} \min_{\substack{\overrightarrow{\pi}(x) = \overrightarrow{\pi}'(x) \\ \overrightarrow{\pi}' \in \delta^*}} |\overrightarrow{\pi}'|$$

The diameter of Balls in Boxes is at most 3 because with 3 ball movements you can reach an arbitrary state. Indeed, the diameter happens to be exactly 3.

1.6.2 SAS+

SAS+ is similar to STRIPS, with the difference being that not only boolean but arbitrary finite-domain variables are allowed. We restrict ourselves to non-negative integer-valued SAS+ though.

Definition 5 (SAS+ states). *A SAS+ maplet $v \mapsto n$ maps a variable v to a non-negative integer value n.*

The rest is analogous to STRIPS. In this thesis, whenever it's not explicitly stated, the STRIPS versions of the objects are meant.

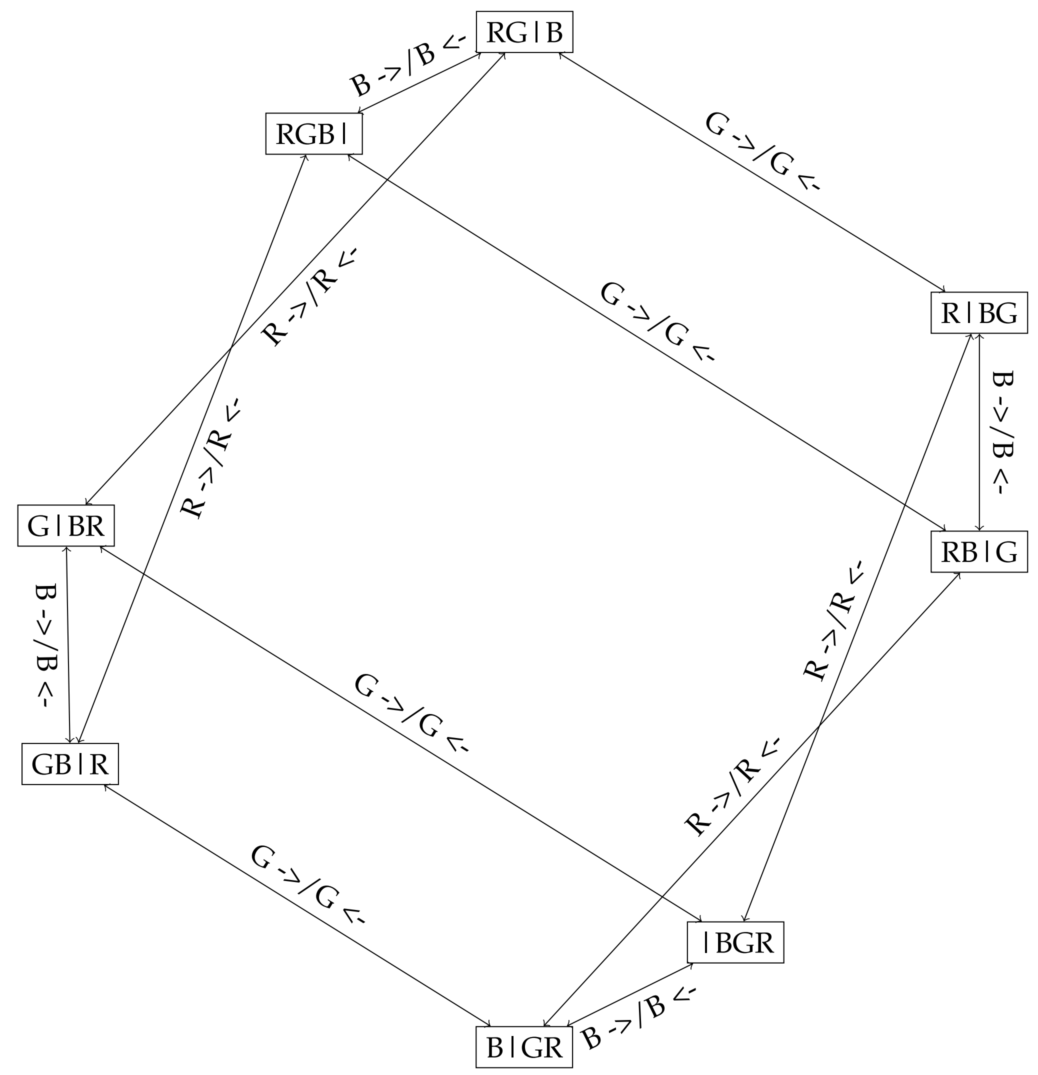

Figure 1.2: Transition system of Balls in Boxes

2 SAT encoding of the recurrence diameter

2.1 The recurrence diameter

The recurrence diameter is the length of the longest simple, i.e., loop-free, path in the state space of a transition system. It was devised by Biere, Cimatti, Clarke, and Zhu, 1999, p. 203.

Definition 6 (Recurrence diameter)**.** *Let* $\text{distinct}(x, \overrightarrow{\pi})$ *denote that all states traversed by executing* $\overrightarrow{\pi}$ *at x are distinct states. The recurrence diameter is the length of the longest simple path in the state space, formally:*

$$\text{rd}(\delta) = \max_{\substack{x \in \mathbb{U} \\ \overrightarrow{\pi} \in \delta^* \\ \text{distinct}(x, \overrightarrow{\pi})}} |\overrightarrow{\pi}|$$

In our example Balls in Boxes the recurrence diameter is 7. In Figure 2.1 you see a path of length 7 marked in blue, which visits no state twice. It is goes RB | G, RGB | , RG | B, R | BG, | BGR, B | GR, GB | R, G | BR. This is an example for a maximal recurrence diameter—it equals the number of states minus one because there is a simple path visiting all states.

In general, we can predict the range in which the recurrence diameter must lie.

Proposition 1 (Bounds of the recurrence diameter)**.** *The recurrence diameter of a transition system is bound below by its diameter and above by the number of states.*

$$\text{d}(\delta) \leq \text{rd}(\delta) < |\mathbb{U}(\delta)|$$

(Abdulaziz & Berger, 2021)

2.2 From SAS+ to STRIPS

In order to encode the recurrence diameter of a SAS+ factored transition system in SAT, we have to translate it to a STRIPS factored transition system. This is because SAS+ allows arbitrary finite-domain values for the variables whereas STRIPS restricts them to booleans and booleans are what we need for a SAT encoding. The translation process we use was inspired by Abdulaziz and Kurz, n.d. It has been adapted to Standard ML and, to fit the existing code base, some preprocessing was necessary.

We start with a SAS+ transition system δ_{SAS+} and create a STRIPS transition system $\delta_{STRIPS} = t(\delta_{SAS+})$, where t is the translation function defined below.

- One SAS+ variable is translated to multiple STRIPS variables. Specifically, for a variable $v_{SAS+} \in \mathcal{D}(\delta_{SAS+})$, for each integer n, such that $\exists x : (v_{SAS+} \mapsto n) \in x \wedge x \in \mathbb{U}(\delta_{SAS+})$ (i.e., n is a possible value for v_{SAS+}), we create $v_{n,STRIPS}$.
- One SAS+ state is translated to one STRIPS state. Specifically, for a state $x_{SAS+} \in \mathbb{U}(\delta_{SAS+})$, for every $(v_{SAS+} \mapsto n) \in x$, we create $t(v_{SAS+} \mapsto n) = \{v_{n,STRIPS} \mapsto \top\} \cup \{v_{m,STRIPS} \mapsto \bot \mid m \neq n\}$. Then $x_{STRIPS} = t(x_{SAS+}) = \bigcup_{(v_{SAS+} \mapsto n) \in x_{SAS+}} t(v_{SAS+} \mapsto n)$
- One SAS+ action is translated to one STRIPS action. Specifically, for an action $\pi_{SAS+} \in \delta_{SAS+}$ with $\pi_{SAS+} = (p, e)$, we create $t(\pi_{SAS+}) = (t(p), t(e))$.

Then $\delta_{STRIPS} = t(\delta_{SAS+}) = \{t(\pi_{SAS+}) \mid \pi_{SAS+} \in \delta_{SAS+}\}$.

2.3 The encoding

The encoding expresses in a propositional formula whether a certain number is greater equal the recurrence diameter. It comprises four parts. The first three stem from Abdulaziz and Berger, 2021, p. 5. The fourth part is needed because the STRIPS transition system is derived from a SAS+ transition system.

The variables in the encoding are in shorter forms than above. Instead of writing $v_{n,STRIPS}$, we simply write v for a variable. v_i stands for variable v's value at step i.

The first part, $F_1(\delta, k)$, states that activated actions must be legal, i.e., at the step of their activation, their preconditions are met, and one step later, their effects have been applied. Additionally, no other variables should change their value (frame condition). This implies that per step at most one action can be activated (except for actions with identical effects).

For an action π, let pre(π) denote its preconditions and eff(π) denote its effects. For a state x, let x_i denote the formula $(\bigwedge_{(v \mapsto \top) \in x} v_i) \wedge (\bigwedge_{(v \mapsto \bot) \in x} \neg v_i)$.

$$F_1(\delta, k) := \bigwedge_{1 \leq i \leq k} \bigwedge_{\pi \in \delta} \pi_i \implies \text{pre}(\pi)_i \wedge \text{eff}(\pi)_{i+1} \wedge \bigwedge_{v \in \mathcal{D}(\delta) \setminus \mathcal{D}(\text{eff}(\pi))} v_i \iff v_{i+1}$$

The second part, $F_2(\delta, k)$, requires that per step at least one action is activated.

$$F_2(\delta, k) := \bigwedge_{1 \leq i \leq k} \bigvee_{\pi \in \delta} \pi_i$$

The third part, $F_3(\delta, k)$, ensures that the action sequence produces a simple path. No state can be visited twice, and therefore, for the states at two arbitrary steps, at least one variable must differ.

$$F_3(\delta, k) := \bigwedge_{1 \le i < j \le k+1} \bigvee_{v \in \mathcal{D}(\delta)} \neg(v_i \iff v_j)$$

The last part, $F_4(\delta, k)$, ensures that we are always in a legal state with respect to the original SAS+ problem. As we have propositional variables in the translated STRIPS version, that state whether a variable in the SAS+ problem has a certain value or not, we need to make sure that for all original SAS+ variables, of all the STRIPS variables derived from it, there is always exactly one which is true.

Let $\text{samesas}(u, v)$ mean that u and v are derived from the same SAS+ variable.

$$F_4(\delta, k) := \bigwedge_{1 \le i \le k+1} \left(\bigwedge_{v \in \mathcal{D}(\delta)} \bigvee_{\substack{u \in \mathcal{D}(\delta) \\ \text{samesas}(u,v)}} u_i \right) \wedge \bigwedge_{\substack{v,u \in \mathcal{D}(\delta) \\ \text{samesas}(u,v)}} \neg v_i \vee \neg u_i$$

We put those four together to obtain a lower bound on the recurrence diameter. Let $k \in \mathbb{N}_0$ and δ be a factored transition system. Then

$$\text{rd}(\delta) \ge k \equiv F_1(\delta, k) \wedge F_2(\delta, k) \wedge F_3(\delta, k) \wedge F_4(\delta, k)$$

To turn it into an encoding suitable for a SAT solver, we need to express it in a DIMACS string, and therefore have to convert it to Conjunctive Normal Form (CNF). $F_2(\delta, k)$ and $F_4(\delta, k)$ are already in CNF, and for $F_1(\delta, k)$ this is relatively trivial. We express the implication as a disjunction with the first argument, π_i, negated. Then we push $\neg\pi_i$ inwards into the clauses.

$$\begin{aligned}
F_1'(\delta, k) := \bigwedge_{1 \le i \le k} \bigwedge_{\pi \in \delta} & \left(\bigwedge_{(v \mapsto \top) \in \text{pre}(\pi)} \neg\pi_i \vee v_i \right) \wedge \left(\bigwedge_{(v \mapsto \bot) \in \text{pre}(\pi)} \neg\pi_i \vee \neg v_i \right) \\
& \wedge \left(\bigwedge_{(v \mapsto \top) \in \text{eff}(\pi)} \neg\pi_i \vee v_{i+1} \right) \wedge \left(\bigwedge_{(v \mapsto \bot) \in \text{eff}(\pi)} \neg\pi_i \vee \neg v_{i+1} \right) \\
& \wedge \left(\bigwedge_{v \in \mathcal{D}(\delta) \setminus \mathcal{D}(\text{eff}(\pi))} (\neg\pi_i \vee v_i \vee \neg v_{i+1}) \wedge (\neg\pi_i \vee \neg v_i \vee v_{i+1}) \right) \\
\equiv F_1(\delta, k) &
\end{aligned}$$

For $F_3(\delta, k)$ though, naive transformation would result in an exponential blowup of the formula. One way to illustrate this is to have a conjunction over all valid states, which are exponentially more than the variables. This conjunction over the states has no semantic meaning here and merely serves as generating all possible boolean strings over the variables:

$$F_3'(\delta, k) := \bigwedge_{1 \leq i < j \leq k+1} \bigwedge_{x \in \mathbb{U}(\delta)} \left(\bigvee_{(v \mapsto \top) \in x} v_i \vee v_j \right) \vee \left(\bigvee_{(v \mapsto \bot) \in x} \neg v_i \vee \neg v_j \right) \equiv F_3(\delta, k)$$

An exponentially increased formula would make the encoding perform poorly. Therefore, we introduce auxiliary variables $a_{i,j,v}$, which is fine because we only need an *equisatisfiable* encoding. This way, we obtain a more compact encoding which will perform better.

$$\begin{aligned} F_3''(\delta, k) := \bigwedge_{1 \leq i < j \leq k+1} \Bigg(& \bigwedge_{v \in \mathcal{D}(\delta)} (\neg a_{i,j,v} \vee v_i \vee v_j) \wedge (\neg a_{i,j,v} \vee \neg v_i \vee \neg v_j) \\ & \wedge (a_{i,j,v} \vee \neg v_i \vee v_j) \wedge (a_{i,j,v} \vee v_i \vee \neg v_j) \Bigg) \wedge \bigvee_{v \in \mathcal{D}(\delta)} a_{i,j,v} \\ \equiv_{SAT} F_3(\delta, k) & \end{aligned}$$

Theorem 1 (Recurrence diameter in CNF). *Let $k \in \mathbb{N}_0$ and δ be a factored transition system. Then the recurrence diameter of δ is at least k if and only if the conjunction of the above formulae holds.*

$$\mathrm{rd}(\delta) \geq k \equiv F_1(\delta, k)' \wedge F_2(\delta, k) \wedge F_3''(\delta, k) \wedge F_4(\delta, k)$$

2.4 Implementation

The implementation was done in Standard ML '97. Consequently, the implementation was all done in functional programming style. It builds up on an existing code base of a tool used to compute bounds for transition systems. The old version of the tool uses an SMT encoding of the recurrence diameter.

Of the abstractions computed by the program, only some are bounded using the recurrence diameter as a base case function. The others are bounded using different base case functions which are easier to compute. The decision, when to use the recurrence diameter, is based on a threshold. If, in an abstraction, the number of states is not greater than the threshold, the recurrence diameter is used.

It would take too much space to show and explain the entire code here. Therefore, we stick with one example to give a rough idea of the implementation style.

The encoding is based on a datatype for propositional formulae:

```
datatype 'a formula = Atom of 'a
                    | Not of 'a formula
                    | And of 'a formula * 'a formula
                    | Or of 'a formula * 'a formula
```

We use a folding function to do a conjunction over lists. The same function is implemented for disjunctions too.

```
fun big_and [x] = x
  | big_and (x :: xs) = And (x, big_and xs)
  | big_and [] = raise Empty
```

An example from the encoding. This part ensures that at least one action is activated per step.

```
fun encode_at_least_one_operator_per_step (prob : (int * int) strips_problem) k =
let
  val ops = #operators_of prob
in
  big_and (map (fn i =>
    big_or (map (fn opr => Atom (Operator (i, index ops opr))) ops)
  ) (tl (list_up_to k)))
end
```

To translate the formula datatype into DIMACS, we need to encode variables as natural numbers. Below, you can see the corresponding code, which is tailored to the encoding and the implementation. A more general approach such as Cantor's encoding would produce too large numbers.

```
fun var_to_int k nops nvars (Operator (i, opr)) = 1 + i + (k + 2) * opr
  | var_to_int k nops nvars (State (i, var)) = 1 + i + (k + 2) * (nops + var)
  | var_to_int k nops nvars (Auxiliary (i, j, var)) =
  1 + i + (k + 2) * (nops + var) + (k + (k + 2) * (nops + nvars)) * (j - 1)
```

The iterative testing for the recurrence diameter uses Theorem 1.

```
fun RDsat_algo sas_prob max_d =
let
  val path_to_solver = "/usr/local/bin/kissat"
  val solver_options = ["-q", "-n"]
  val (sat_fmt, unsat_fmt) = ("s␣SATISFIABLE\n", "s␣UNSATISFIABLE\n")

  val strips_prob = SasToStrips.translate sas_prob

  fun sat_for_d d =
    ...

  fun iterate_d d = if d > max_d then max_d
            else if not (sat_for_d d) then d - 1 else iterate_d (d + 1)
in
  iterate_d 1
end
```

For SAT solving, Kissat, version 2.0.0, (Biere, Fazekas, Fleury, & Heisinger, 2020) was used. The encoding is expressed as a DIMACS string. The SAT solver is spawned as a child process with the `Unix.execute` function. The DIMACS string was finally fed to the SAT solver using a `TextIO.outstream` and the result was read using a `TextIO.instream`. Both were created with the `Unix.streamsOf` function.

2.4.1 Gaining performance using functional fusion

During the experiments (more on that later), it became clear that most of the time was consumed on building the DIMACS string of the encoding formula and not, as one could have thought, on SAT solving. As we wanted SAT-solving to be the bottleneck, we implemented a pipelining process which creates a part of the DIMACS string and directly forwards it to the SAT-solver. The concept is reminiscent of functional stream fusion (Coutts, Leshchinskiy, & Stewart, 2007, 9) and we will therefore call it the fusion approach.

```
  fun sat_for_d d =
  let
    val sat_proc =
      Unix.execute (path_to_solver, solver_options)
    val (instrm, outstrm) = Unix.streamsOf sat_proc
    val _ = TextIO.output (outstrm, RDCheckSAT.get_header (strips_prob, d))
    val encodings = RDCheckSAT.get_encodings (strips_prob, d)
    val _ = app (fn enc =>
      TextIO.output (outstrm, RDCheckSAT.to_dimacs
      (strips_prob, d, enc) ^ "␣0\n")) encodings
    val _ = TextIO.closeOut outstrm
    val ret = TextIO.input instrm
    val status = Unix.reap sat_proc
  in
    if ret = sat_fmt then true
    else if ret = unsat_fmt then false
    else raise Fail ("Something␣went␣wrong␣with␣the␣SAT␣solver:\n" ^ ret)
  end
```

2.5 Experimental results

The experiments were conducted on this hardware:

- x86_64 architecture
- Intel Xeon Processor (Skylake, IBRS)
- 2394.372 MHz
- 32KB L1 data cache
- 32KB L1 instruction cache
- 4096KB L2 cache
- 16384KB L3 cache
- 44GB RAM

For the performance measurements, several common planning benchmarks were used. Table 2.1 shows a first comparison of the two program versions. It is grouped by the domains of the benchmarks. The first column shows the domain name, and the second column gives the total number of problems that were used during that run in that domain. Then follow the number of bounded instances and the average computation time taken to compute the bounds, both for the SMT-based (old) version of the program and for the SAT-based (new) version which was implemented in this thesis. A timeout of one hour was used.

This first comparison is based on the program version which does not use the fusion approach. It can be seen that the SAT-based program performs worse on some of the benchmarks and almost never better than the SMT-based. Therefore we did measurements with a higher threshold, conjecturing that the relative performance would be better because SAT should be more efficient than SMT for bigger instances.

		SMT-basd		SAT-based	
Domain	Total	Bounded	Average time	Bounded	Average time
elevators	210	164	120	114	237
hiking	40	21	261	21	271
logistics	407	407	245	356	252
nomystery	124	124	5	124	4
rover	104	50	214	51	278
tpp	89	12	556	12	550
transport	203	14	49	14	44
zeno	50	48	118	48	485

Table 2.1: Comparison without fusion with threshold 50

In Table 2.2 you see a comparison with a threshold of 100 for the recurrence diameter subroutine. Here, we increased the timeout to two hours and left it at two hours for the rest of the measurements. It can be seen that a higher threshold indeed slightly increases the relative performance of the SAT approach. Nevertheless, it still falls behind our expectations.

		SMT-based		SAT-based	
Domain	Total	Bounded	Average time	Bounded	Average time
elevators	188	142	313	88	837
hiking	40	22	457	22	494
logistics	285	253	315	277	501
nomystery	124	124	5	124	3
rover	58	26	700	26	701
tpp	47	13	929	13	938
transport	203	14	51	14	45
zeno	50	40	213	39	685

Table 2.2: Comparison without fusion with threshold 100

After investigating what takes the most time during the runs of the SAT-based program, we ran the improved version with functional fusion for the string-building process. Table 2.3 shows the results of the new measurements. The fusion string building decreases the running time of the new program, so that for all domains except "elevators" it performs at least as good as the old SMT-based version, and performs better on some domains.

		SMT-based		SAT-based	
Domain	Total	Bounded	Average time	Bounded	Average time
elevators	210	164	380	129	649
hiking	40	22	469	22	512
logistics	305	253	320	303	416
nomystery	124	124	5	124	3
rover	85	47	587	49	619
tpp	53	13	930	13	930
transport	203	14	50	14	41
zeno	50	40	216	43	409

Table 2.3: Comparison with threshold 100

Table 2.4 shows similar results for a threshold of 1000. It is interesting that the performance of both program versions only decreases little compared to using 100 as the threshold.

		SMT-based		SAT-based	
Domain	Total	Bounded	Average time	Bounded	Average time
elevators	210	159	382	124	678
hiking	40	22	475	22	506
logistics	316	251	318	301	382
nomystery	124	106	107	112	268
rover	92	40	522	43	645
tpp	67	13	940	13	946
transport	203	14	54	14	45
zeno	50	28	259	31	554

Table 2.4: Comparison with threshold 1000

Figure 2.2, Figure 2.3, Figure 2.4 and Figure 2.5 show scatter plots comparing the running times for the computed bounds. During the experiments it became clear that the process of SAT solving takes almost no time in comparison to the process of building the encoding. We have no comprehensive measurements for that, but for example, for one instance in the elevators domain, the SAT solving took less than a millisecond for all but two calls, the two exceptions being 1 and 3 milliseconds. In contrast to that, building the encoding took multiple seconds in about half of the calls.

Beyond the runtime measurements it became clear that for another benchmark domain, newopen, the old SMT-based program produced false results for the bounds by claiming the recurrence diameter would always be 0. We did not have the time to further investigate in that, though.

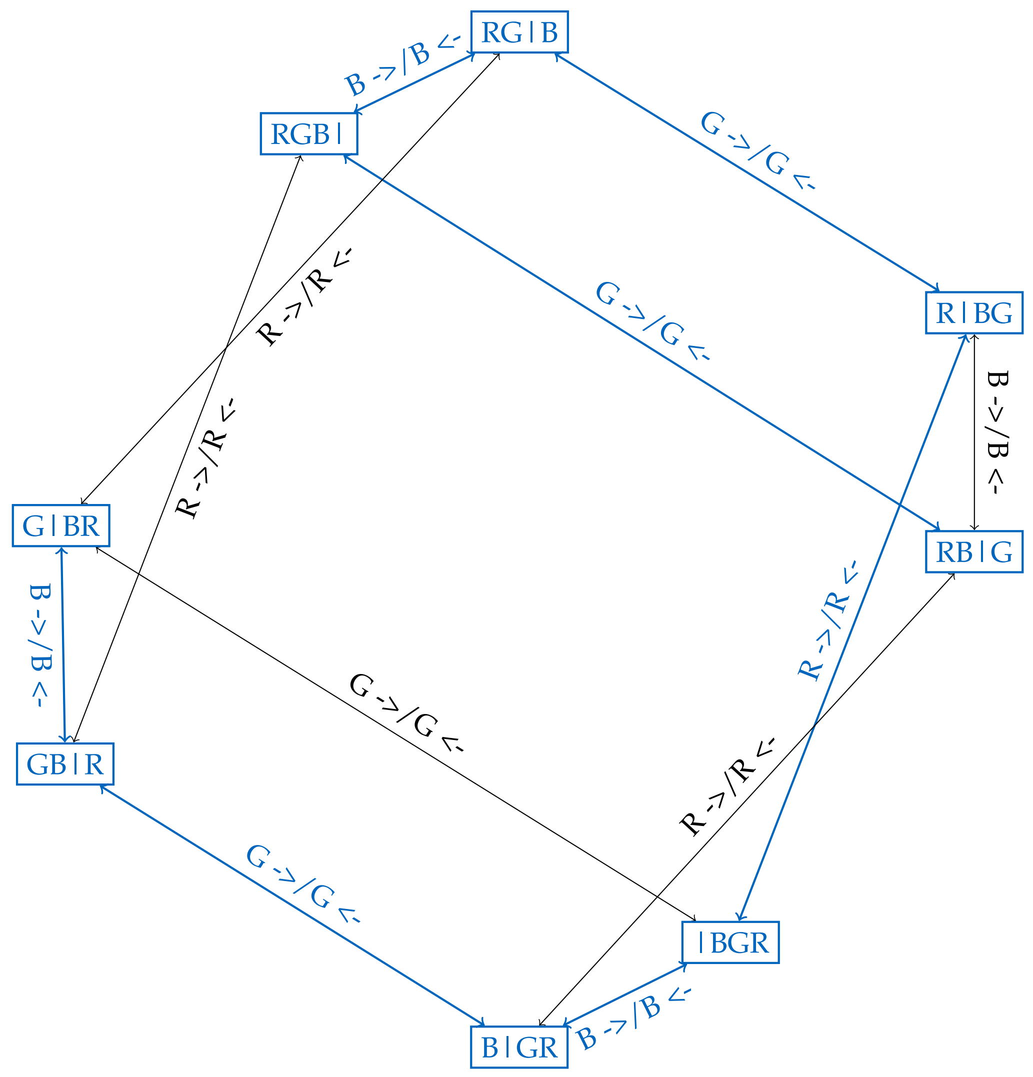

Figure 2.1: Recurrence diameter of Balls in Boxes

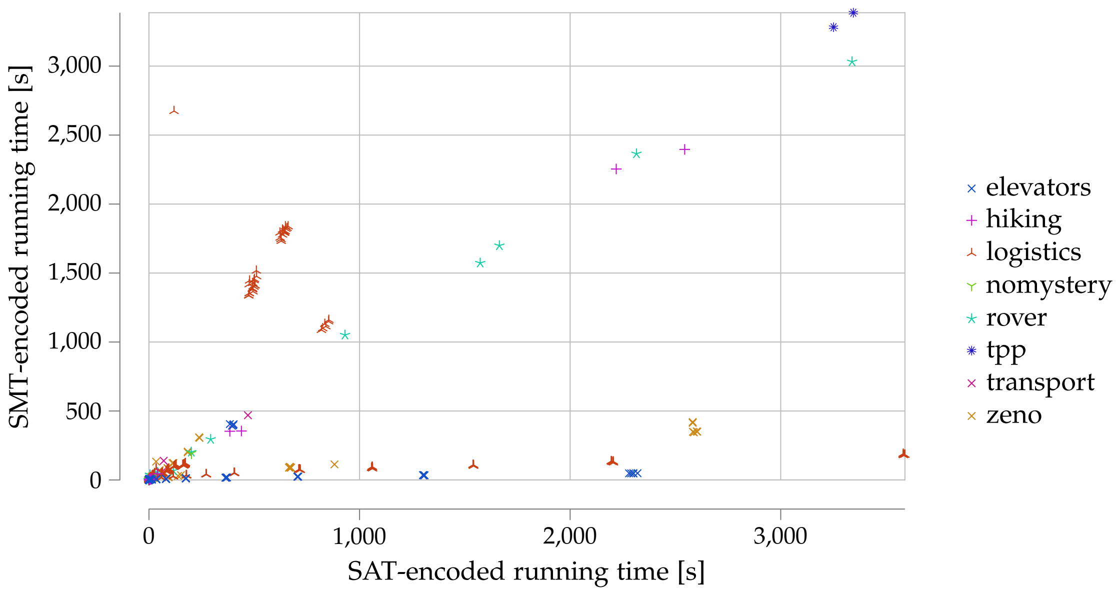

Figure 2.2: Comparison with threshold 50

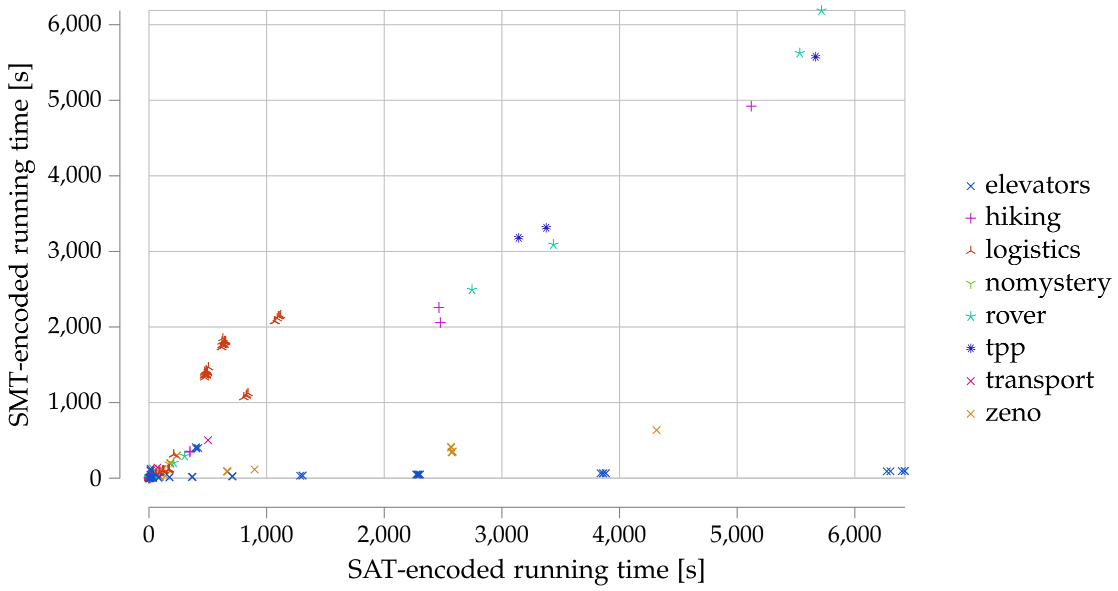

Figure 2.3: Comparison with threshold 100

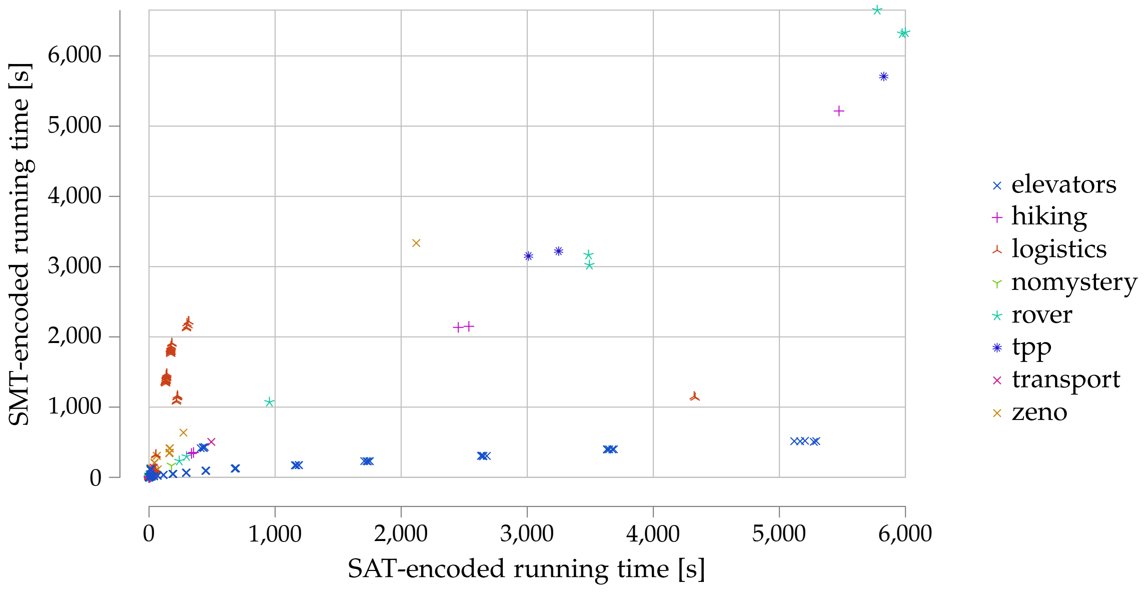

Figure 2.4: Comparison with threshold 100 with functional fusion for SAT version

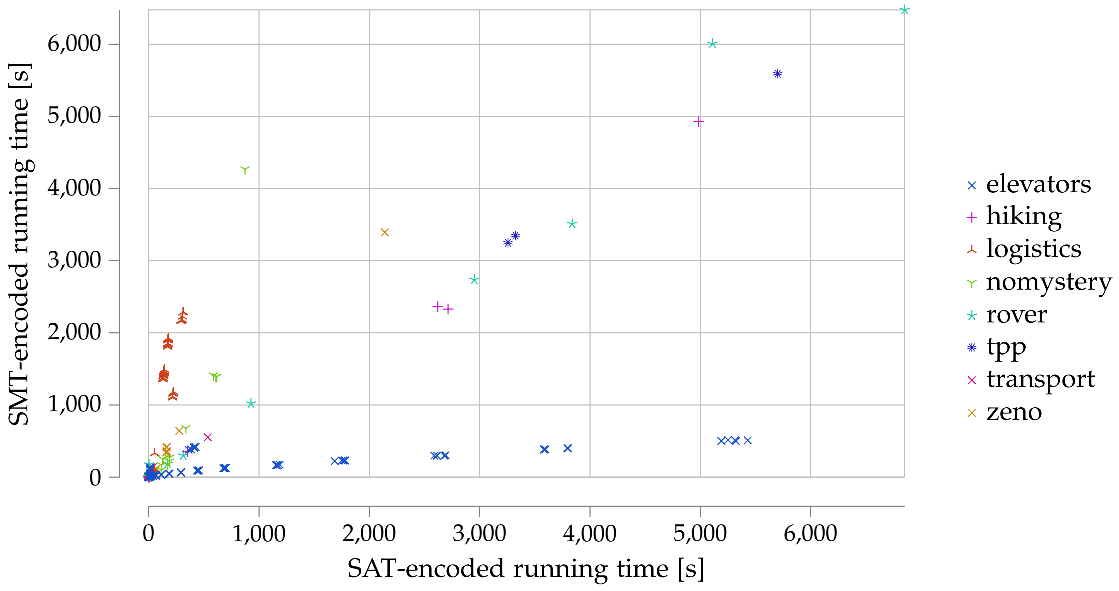

Figure 2.5: Comparison with threshold 1000 with functional fusion for SAT version

3 QBF encoding of the sublist diameter

Abdulaziz, Gretton, and Norrish, 2015, showed that the sublist diameter, in contrast to the diameter, can be used for compositional bounding.

3.1 Quantified Boolean Formulae

QBF describes a class of problems that is a superset of SAT. It covers satisfiability and validity problems but, in addition to plain propositional logic, allows to quantify over the possible values of propositional variables. For example:

$$\forall A : \exists B : (A \wedge B) \vee (\neg A \wedge \neg B)$$

is valid, but

$$\exists B : \forall A : (A \wedge B) \vee (\neg A \wedge \neg B)$$

is not. Note that free variables are allowed and are equivalent to outermost existentially quantified variables. So the last formula is equivalent to

$$\forall A : (A \wedge B) \vee (\neg A \wedge \neg B)$$

QBF is the canonical $PSPACE$-complete problem (Cadoli, Giovanardi, & Schaerf, 1998). Provided that $PSPACE \neq NP$, this means that QBF is harder than SAT.

3.2 The sublist diameter

A list l' is a sublist of l, written $l' \preceq\cdot\, l$, if and only if all the members of l' occur in the same order in l. We can treat action sequences as lists and therefore define the sublist diameter on them.

Definition 7 (Sublist diameter). *Given a factored transition system δ, the sublist diameter $\mathrm{sd}(\delta)$ is the length of the longest shortest equivalent sublist to any execution $\overrightarrow{\pi} \in \delta^*$ starting at any state $x \in \mathbb{U}(\delta)$. Formally,*

$$\mathrm{sd}(\delta) = \max_{\substack{x \in \mathbb{U} \\ \overrightarrow{\pi} \in \delta^*}} \; \min_{\substack{\overrightarrow{\pi}(x) = \overrightarrow{\pi}'(x) \\ \overrightarrow{\pi}' \preceq\cdot\, \overrightarrow{\pi}}} |\overrightarrow{\pi}'|$$

The sublist diameter of Balls in Boxes is 3. A sketch of the reason is visualized in Figure 3.1: The longest simple paths always move balls back and forth. These movements can be dropped in a sublist path: RB | G, R | BG, | BGR, G | BR.

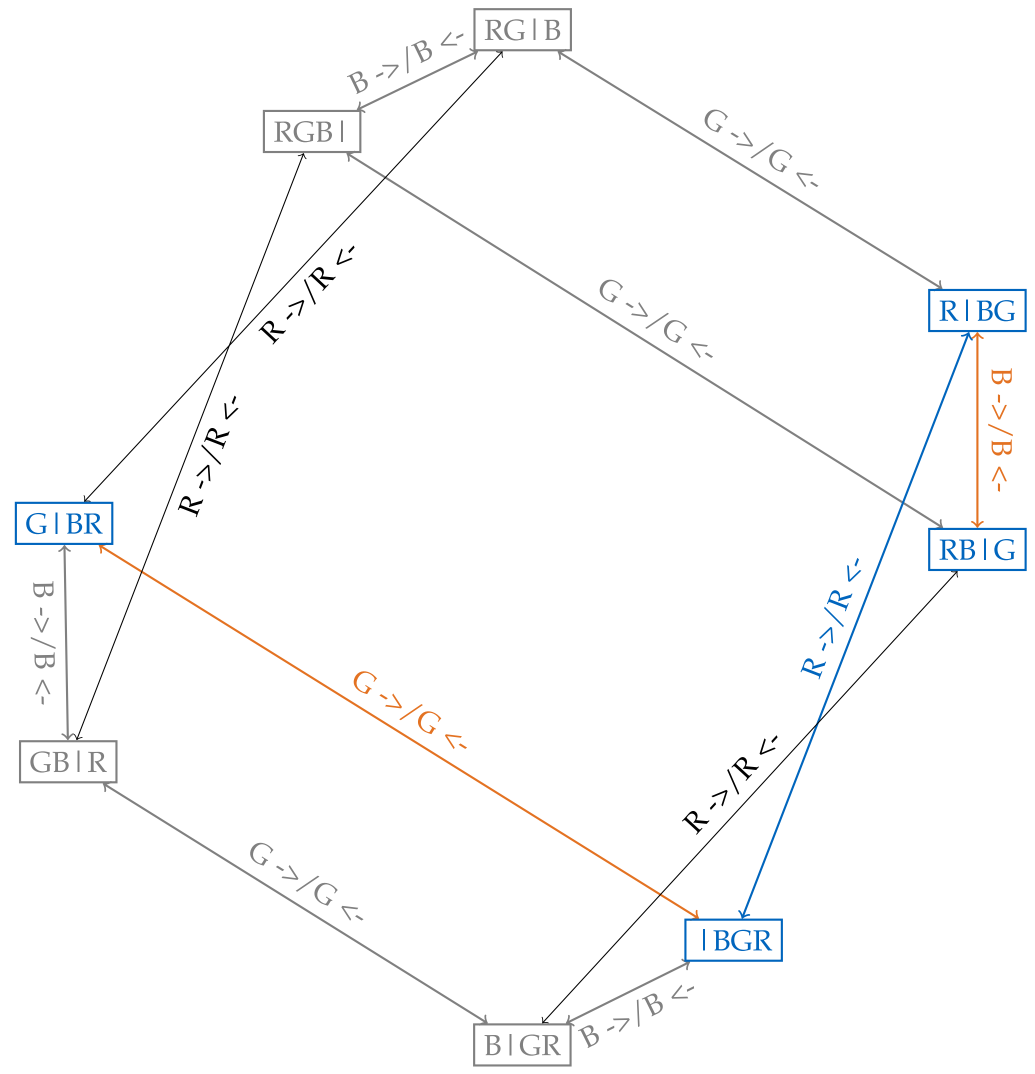

Figure 3.1: Sublist diameter of Balls in Boxes

As for the recurrence diameter, we want to obtain an iterative approach to finding the sublist diameter. We can use the recurrence diameter for that because, looking at Definition 7, we see that if we always just take $\overrightarrow{\pi}'$ to be $\overrightarrow{\pi}$ without its loops, instead of the actual minimal sequence, we get the recurrence diameter.

Proposition 2 (Bounds of the sublist diameter)**.** *The sublist diameter of a transition system is bound below by its diameter and above by the recurrence diameter.*

$$\mathrm{d}(\delta) \leq \mathrm{sd}(\delta) \leq \mathrm{rd}(\delta)$$

(Abdulaziz, Gretton, & Norrish, 2015, p. 6)

3.3 The encoding

The encoding works as follows: Two action sequences are encoded. Over the variables of the first one we quantify universally, and we quantify existentially over the variables of the second one. These two correspond to $\overrightarrow{\pi}$ and $\overrightarrow{\pi}'$ from Definition 7, respectively. For both of them we encode the length and require that the second one is a sublist of the first. If then, for all possible action sequences of the first length (universally quantified), there exists a corresponding second action sequence with the second given length which is a sublist of the first *and* produces the same final state if executed from the same initial state, we have upper bounded the sublist diameter. From now on we call the first action sequence, which is universally quantified over, the *original* action sequence, and the second action sequence, depending on it, the *sublist* action sequence. We will explore the single parts of the encoding in the following.

First, similar to encoding the recurrence diameter, we need to make sure that between states the variables change according to the action executed. Again, this also ensures that (except for actions with identical effects) at most one action is executed per step. We will later use this part for both the original and the sublist sequence. We introduce an additional empty action because we want original paths of all lengths and not only the maximal length. This can be simulated by executing the empty action in a step.

$$G_1(\delta, k, m) := \bigwedge_{0 \leq i < n} \bigwedge_{\pi \in \delta \cup \{(\emptyset, \emptyset)\}} \pi_{i,m} \implies \mathrm{pre}(\pi)_{i,m} \wedge \mathrm{eff}(\pi)_{i,m} \wedge \bigwedge_{v \in \mathcal{D}(\delta) \setminus \mathcal{D}(\mathrm{eff}(\pi))} v_{i,m} \iff v_{i+1,m}$$

We denote the set of atoms $\{\pi_{i,m} \mid 0 \leq i < k \wedge \pi \in \delta \cup \{(\emptyset, \emptyset)\}\} \cup \{v_{i,m} \mid 0 \leq i \leq k \wedge v \in \mathcal{D}(\delta)\}$ on which $G_1(\delta, k, m)$ is defined by $\mathcal{D}(G_1(\delta, k, m))$. Note that, since the kth step is the last one, no action can be executed in it.

We demand that in every step at least one action is executed.

$$G_2(\delta, k, m) := \bigwedge_{0 \leq i < k} \bigvee_{\pi \in \delta \cup \{(\emptyset, \emptyset)\}} \pi_{i,m}$$

The action sequences have to have the same initial state.

$$G_3(\delta) := \bigwedge_{v \in \mathcal{D}(\delta)} v_{0,1} \iff v_{0,2}$$

And the same final state.

$$G_4(\delta, h, l) := \bigwedge_{v \in \mathcal{D}(\delta)} v_{l,1} \iff v_{h,2}$$

We define correspondences between actions in the original and the sublist sequence. They require that at the respective steps in the sequences the same action is executed. Note that $i \mapsto i'$ is one propositional variable.

$$G_5(\delta, h, l) := \bigwedge_{0 \le i < h} \bigwedge_{i \le i' \le l-h+i} i \mapsto i' \implies \bigvee_{\pi \in \delta \cup \{(\emptyset, \emptyset)\}} \pi_{i',1} \wedge \pi_{i,2}$$

The domain of all correspondence variables is $C = \{i \mapsto i' \,|\, 0 \le i < h \wedge i \le i' \le l - h + i\}$.

For every action in the sublist action sequence there must be a corresponding action in the original action sequence.

$$G_6(h, l) := \bigwedge_{0 \le i < h} \bigvee_{i \le i' \le l-h+i} i \mapsto i'$$

But there cannot be more than one corresponding action.

$$G_7(h, l) := \bigwedge_{0 \le i < h} \bigwedge_{i \le i' \le l-h+i} \bigwedge_{i' < i'' \le l-h+i} \neg i \mapsto i' \vee \neg i \mapsto i''$$

Finally, to have the sublist property, the corresponding actions have to be ordered.

$$G_8(h, l) := \bigwedge_{0 \le i < h} \bigwedge_{i \le i' \le l-h+i} i \mapsto i' \implies \bigwedge_{i < i'' < h} \bigvee_{i'' < i''' \le l-h+i''} i'' \mapsto i'''$$

By combining all of this, we can express the sublist diameter. Note that for the sublist diameter a maximum is encoded, i.e., we state it must be lower equal some number. For the recurrence diameter it was a minimum, i.e., we stated it must be greater equal some number.

Theorem 2 (Sublist diameter). *Let $h, l \in \mathbb{N}_0$, $h \le l$ and δ be a factored transition system. Then, if l is the recurrence diameter of δ, the sublist diameter of δ is at most h if and only if the conjunction of the above formulae holds.*

$$\begin{aligned} &\mathrm{sd}(\delta) \le h \equiv l = \mathrm{rd}(\delta) \wedge \\ &\qquad \forall \mathcal{D}(G_1(\delta, l, 1)) : \exists \mathcal{D}(G_1(\delta, h, 2)) : \exists C : (G_1(\delta, l, 1) \wedge G_2(\delta, l, 1)) \implies \\ &(G_1(\delta, h, 2) \wedge G_2(\delta, h, 2) \wedge G_3(\delta) \wedge G_4(\delta, h, l) \wedge G_5(\delta, h, l) \wedge G_6(h, l) \wedge G_7(h, l) \wedge G_8(h, l)) \end{aligned}$$

The encoding states that *if* the variables of the original path form a valid such, *then* a corresponding sublist path must exist. If, on the other hand, the variables for the original path just state nonsense, then of course there also does not have to be a valid sublist path.

Again, we have to convert the propositional part of the encoding into CNF to be able to produce a QDIMACS string. But in this case we omit the CNF version here because it would become too unreadable.

3.4 Implementation

For QBF solving, DepQBF, version 6.03, (Lonsing & Egly, 2017) was used. The implementation follows the same style as for the recurrence diameter but the fusion approach was not used for communication with the solver since here QBF solving was already clearly the bottleneck.

Here you see the code that controls the incremental search for the sublist diameter. Notice how the recurrence diameter has to be computed first.

```
fun SD_algo sas_prob max_d =
let
  val rd = RDsat_algo sas_prob max_d

  val path_to_solver = "/usr/local/bin/depqbf"
  val solver_options = []
  val (sat_fmt, unsat_fmt) = ("SAT\n", "UNSAT\n")

  val strips_prob = SasToStrips.translate sas_prob

  fun sat_for_d d =
  let
    val qbf_proc =
      Unix.execute (path_to_solver, solver_options)
    val (instrm, outstrm) = Unix.streamsOf qbf_proc
    val _ = TextIO.output (outstrm, SDCheckQBF.generate (strips_prob, d, rd))
    val _ = TextIO.closeOut outstrm
    val ret = TextIO.input instrm
    val status = Unix.reap qbf_proc
  in
    if ret = sat_fmt then true
    else if ret = unsat_fmt then false
    else raise Fail ("Something␣went␣wrong␣with␣the␣QBF␣solver:\n" ^ ret)
  end

  fun iterate_d d = if d = rd then rd
                    else if not (sat_for_d d) then iterate_d (d + 1) else d
in
  iterate_d 1
end
```

3.5 Experimental results

The experiments were conducted on the same hardware as in section 2.5. Timeouts were still two hours.

It can be seen in Table 3.1 that the use of the sublist diameter as a base case function heavily increases the running times. For those experiments a threshold of 10 was set for use of the sublist diameter as a subroutine, and a threshold of 100 for the recurrence diameter. Recall that, to compute the sublist diameter, it is necessary to first compute the recurrence diameter.

The entries in the table are to be read the following way: The numbers in the first "Bounded" column report the number of instances bound using the recurrence diameter as a base case function. However, for the other columns only those instances were considered which were bound both by the recurrence diameter and by the sublist diameter program

Only for six instances in the logistics domain the sublist diameter-based program was able to compute lower bounds in time. For all other instances, it either did not terminate in time or computed the same bound.

In contrast to the recurrence diameter subroutine, often, the most time was consumed by the QBF solving process and not building the encoding.

Figure 3.2 and Figure 3.3 show scatter plots comparing the running times and the bounds.

	SAT-based recurrence diameter			QBF-based sublist diameter		
Domain	Bounded	Avg. bound	Avg. time	Bounded	Avg. bound	Avg. time
hiking	22	360204777	512	22	360204777	490
logistics	92	17551	63	34	17527	195
nomystery	124	3240	4	105	3240	395
rover	49	32488	260	21	32488	292
tpp	13	402	530	11	402	573
transport	14	488188665	5	8	488188665	5
zeno	43	72995	849	19	72995	82

Table 3.1: Comparison with threshold 100/10

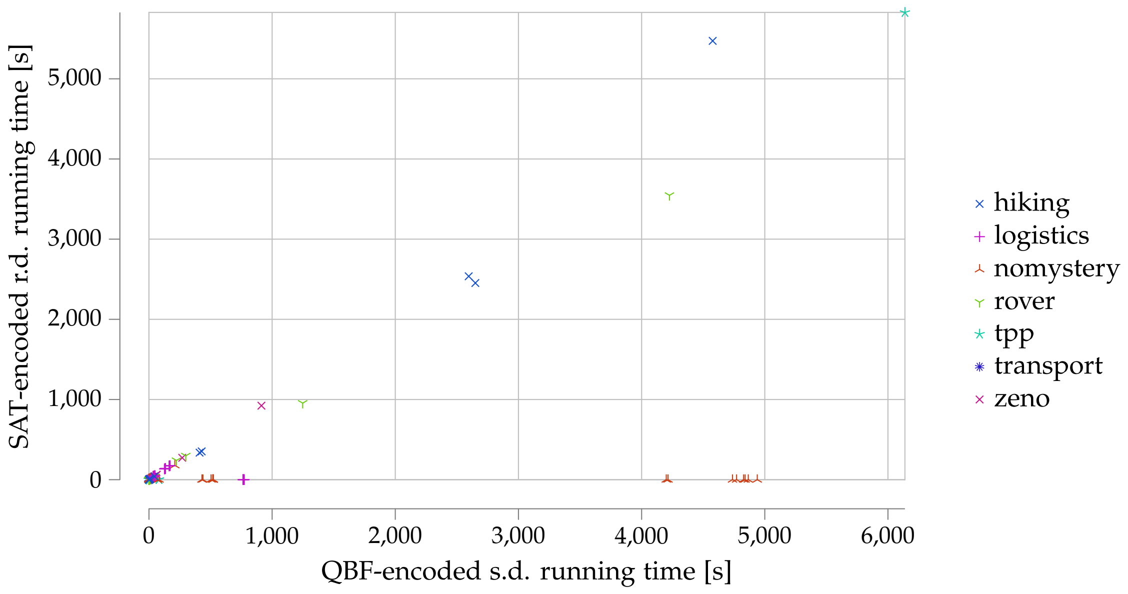

Figure 3.2: Running time comparison with threshold 10 for the sublist diameter and 100 for the recurrence diameter

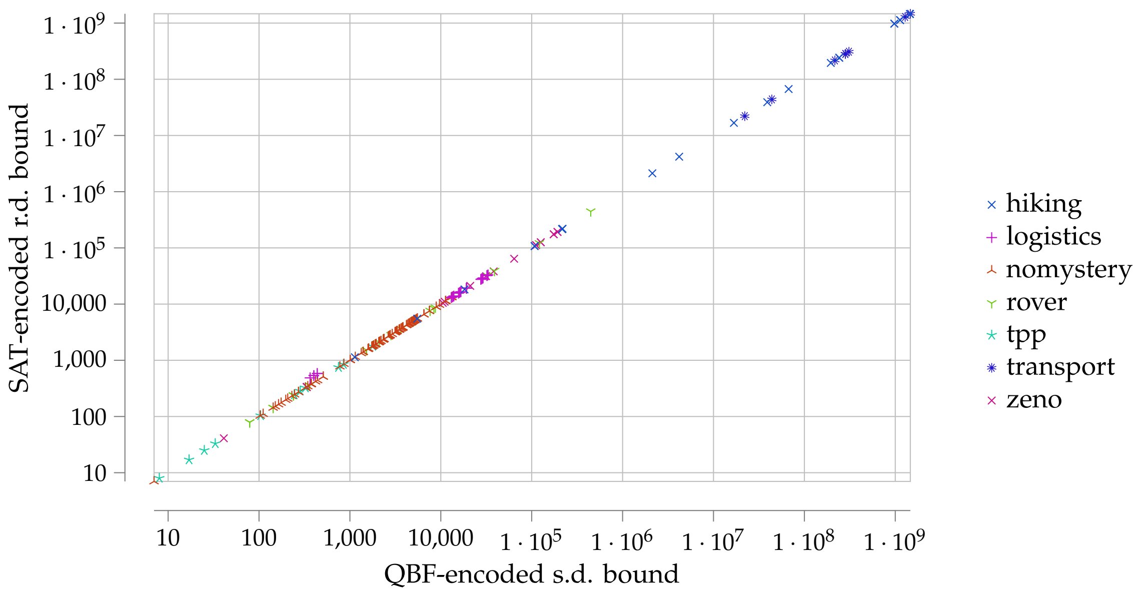

Figure 3.3: Bound comparison with threshold 10 for the sublist diameter and 100 for the recurrence diameter

4 Conclusion

As of today, QBF solving is not yet that advanced and optimized as SAT solving is. This is reflected in the impractical running times of the QBF-based encoding of the sublist diameter.

With the SAT encoding of the recurrence diameter, though, we were able to gain performance in some domains with respect to the SMT encoding. We did not manage to make SAT-solving the bottleneck yet and if fusion for the string building is applied, the encoding process takes the most time. It seems though, that a basis is created on which further optimization might lead to very valuable performance increases in compositional bounding.

4.1 Future work

It would be of interest to formalize aspects of this work in an interactive theorem prover, such as Isabelle (Nipkow, Paulson, & Wenzel, 2002), e.g. proving the encoding of the sublist diameter correct.

Furthermore, it is an open research question whether the encodings we give are the most efficient ones. And it would be very interesting to further optimize the implementation of both the recurrence diameter and the sublist diameter and fine-tune the thresholds to appropriate values. Perhaps the most imminent question is why exactly the SAT-based encoding performs so bad on the elevators domain. Also, it is an option to implement the recurrence and sublist diameter encoding in an imperative language and to compare the performance.

List of Figures

1.1 States of Balls in Boxes 3
1.2 Transition system of Balls in Boxes 7

2.1 Recurrence diameter of Balls in Boxes 18
2.2 Comparison with threshold 50 19
2.3 Comparison with threshold 100 19
2.4 Comparison with threshold 100 with functional fusion for SAT version 20
2.5 Comparison with threshold 1000 with functional fusion for SAT version 20

3.1 Sublist diameter of Balls in Boxes 22
3.2 Running time comparison with threshold 10 for the sublist diameter and 100 for the recurrence diameter 27
3.3 Bound comparison with threshold 10 for the sublist diameter and 100 for the recurrence diameter 28

List of Tables

2.1 Comparison without fusion with threshold 50 15
2.2 Comparison without fusion with threshold 100 15
2.3 Comparison with threshold 100 16
2.4 Comparison with threshold 1000 16

3.1 Comparison with threshold 100/10 27

Bibliography

Abdulaziz, M., & Berger, D. (2021). Computing plan-length bounds using lengths of longest paths. In *Proceedings of the AAAI conference on artificial intelligence* (Vol. 35, *13*, pp. 11709–11717). Association for the Advancement of Artificial Intelligence. AAAI Press.

Abdulaziz, M., Gretton, C. O., & Norrish, M. (2017). A state-space acyclicity property for exponentially tighter plan length bounds. In *Proceedings of the twenty-seventh international conference on automated planning and scheduling* (pp. 2–10). Icaps 2017. eprint: https://aaai.org/ocs/index.php/ICAPS/ICAPS17/paper/viewFile/15768/15082

Abdulaziz, M., Gretton, C., & Norrish, M. (n.d.). *Compositional upper-bounding of diameters of factored digraphs*. Working Paper.

Abdulaziz, M., Gretton, C., & Norrish, M. (2015, August 19). Verified over-approximation of the diameter of propositionally factored transition systems. In C. Urban & X. Zhang (Eds.), *Interactive theorem proving* (pp. 1–16). doi:10.1007/978-3-319-22102-1_1

Abdulaziz, M., & Kurz, F. (n.d.). *Formally verified SAT-based AI planning*. Working Paper.

Baumgartner, J., Kuehlmann, A., & Abraham, J. (2002, July 27). Property checking via structural analysis. In E. Brinksma & K. G. Larsen (Eds.), *Proceedings of the 14th international conference on computer aided verification* (pp. 151–165). Berlin, Heidelberg: Springer Berlin Heidelberg.

Biere, A., Cimatti, A., Clarke, E., & Zhu, Y. (1999, March 12). Symbolic model checking without BDDs. In W. R. Cleaveland (Ed.), *Tools and algorithms for the construction and analysis of systems* (pp. 193–207). doi:10.1007/3-540-49059-0_14

Biere, A., Fazekas, K., Fleury, M., & Heisinger, M. (2020). CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In T. Balyo, N. Froleyks, M. Heule, M. Iser, M. Järvisalo, & M. Suda (Eds.), *Proc. of SAT Competition 2020 – solver and benchmark descriptions* (Vol. B-2020-1, pp. 51–53). Department of Computer Science Report Series B. University of Helsinki.

Cadoli, M., Giovanardi, A., & Schaerf, M. (1998). An algorithm to evaluate quantified boolean formulae. In *Proceedings of the fifteenth national conference on artificial intelligence*, AAAI Press. eprint: https://www.aaai.org/Papers/AAAI/1998/AAAI98-036.pdf

Coutts, D., Leshchinskiy, R., & Stewart, D. (2007, October 1). Stream fusion: From lists to streams to nothing at all. *ACM SIGPLAN Notices*, *42*, 315–326. doi:10.1145/1291220.1291199

Ghallab, M., Nau, D., & Traverso, P. (2004, May 3). *Automated planning: Theory and practice* (1st ed.). The Morgan Kaufmann Series in Artificial Intelligence. doi:10.1016/B978-1-55860-856-6.X5000-5

Lonsing, F., & Egly, U. (2017, July 11). Depqbf 6.0: A search-based qbf solver beyond traditional QCDCL. In L. de Moura (Ed.), *Automated deduction – CADE 26* (pp. 371–384). Lecture Notes in Computer Science. Cham: Springer International Publishing.

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL: A proof assistant for higher-order logic* (1st ed.). Lecture Notes in Computer Science. doi:10.1007/3-540-45949-9