Electronic Communications of the EASST
Volume X (2014)

Proceedings of the 13th International Workshop on Graph Transformation and Visual Modeling Techniques (GTVMT 2014)

Solving the *N*-Queens Problem with GROOVE – Towards a Compendium of Best Practices

Eduardo Zambon and Arend Rensink

13 pages

Guest Editors: Frank Hermann, Stefan Sauer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Solving the N-Queens Problem with GROOVE – Towards a Compendium of Best Practices

Eduardo Zambon[1] **and Arend Rensink**[2]

[1]zambon@inf.ufes.br
Department of Computer Science and Electronics (DCEL/CEUNES)
Federal University of Espirito Santo (UFES), Brazil

[2]arend.rensink@utwente.nl
Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: We present a detailed solution to the N-queens puzzle using GROOVE, a graph transformation tool especially designed for state space exploration and analysis. While GROOVE has been freely available for more than a decade and has attracted a reasonable number of users, it is safe to say that only a few of these users fully exploit the tool features. To improve this situation, using the N-queens puzzle as a case study, in this paper we provide an in-depth discussion about problem solving with GROOVE, at the same time highlighting some of the tool's more advanced features. This leads to a list of best-practice guidelines, which we believe to be useful to new and expert users alike.

Keywords: GROOVE, N-Queens Problem, Tool Usage Guidelines

1 Introduction

It is widely understood that the ultimate acceptance of graph transformation as a practically useful and valuable modelling technique is largely dependant on the availability of reliable tool support. Yet it is almost as widely accepted to be a nearly insurmountable challenge to create and especially maintain that tool support in an academic context. The main difficulty may not even be the lack of manpower but the lack of academic credit that flows from this kind of work: how can one package tool maintenance and documentation in such a way that the results are publishable within the scientific community?

This paper represents an attempt to overcome this challenge. We discuss, on the basis of a well-known example, namely the *N-queens problem*[1], the design choices one has to go through in order to arrive at a solution. The contribution of the paper is not so much the solution itself but the actual development process followed to find said solution.

The N-queens problem is a representative of a certain class of questions which can sometimes be solved analytically, by a proof dedicated to the problem itself, which then immediately applies to arbitrary instances of the question (in this case, arbitrary values for N); or they can be

[1] See, e.g., http://en.wikipedia.org/wiki/Eight_queens_puzzle.

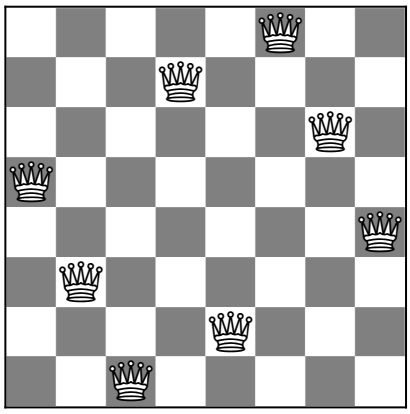


Figure 1: Solution for the 8-queens puzzle.

regarded and treated as search problems, in which the quest is to find a graph (or, more generally, a state) with particular properties. Search problems also include model checking questions, where one typically searches for a state that *violates* some requirements and hence represents erroneous behaviour. These questions can be solved, or attempted to be solved, by general, push-button techniques, but only *once they have been appropriately modelled*. What constitutes an appropriate model is very often a question of trial-and-error: there are always many possible ways to capture any given aspect of a problem, some of which make the solution harder to find automatically whereas others may actually dramatically reduce the search space.

In the case of this paper, we take GROOVE as our general-purpose tool used to solve the problem at hand, and we show how certain tool features can be exploited to significantly improve the capacity for treating larger problem instances. We present these insights in the form of *guidelines* (or *best practices*). Some of these guidelines are valid for any graph transformation modelling tool or even for any modelling attempt at all; some are dedicated to GROOVE and address especially the capacity for symmetry reduction.

More information about GROOVE can be found in, e.g., [Ren03, GMR+12] and the tool user manual; however, this paper is self-contained and understandable without prior experience — even though we heartily recommend trying out the example for oneself.

In the paper, we first recall the problem itself (Section 2) and then develop the solution step-by-step (Section 3). We then show the results for this particular problem (Section 4) and conclude in Section 5.

2 Problem

The N-queens puzzle requires the placement of N queens on an $N \times N$ chessboard in such a way that no queen can attack any other. In chess, a queen is able to move any number of squares in any of the eight directions, and thus a solution to the puzzle requires that no two queens share the same row, column, or diagonal. In the usual instance of the problem a standard sized chessboard is used, and therefore we have $N = 8$. Figure 1 shows one solution for this instance.

	N	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
N-**queens problem**	**distinct solutions**	1	0	0	2	10	4	40	92	352	724
	unique solutions	1	0	0	1	2	1	6	12	46	92
k-**queens variant**	**minimum** k	1	1	1	3	3	4	4	5	5	5
	unique solutions	1	1	1	2	2	17	1	91	16	1

Table 1: Summary of (known) solution counts for the different problem instances.

The 8-queens puzzle has 92 *distinct* solutions. If we collapse (count as one) solutions that differ only by rotations and reflections of the board, the puzzle has 12 *unique* solutions. Given that there are $\binom{64}{8} = 4,426,165,368$ possible arrangements of eight queens on a 8×8 board, it is unfeasible to use a simple brute force algorithm that generates a configuration and tests if it is a solution.

Finding all (unique) solutions to the 8-queens puzzle is a classic example of a simple but non-trivial problem, which has been used many times as an exercise in algorithm design. In 1972, Dijkstra [DDH72] presented a solution using a recursive depth-first backtracking algorithm. Since then many other programming techniques such as constraint and logic programming have used this problem to illustrate their features.

The number of distinct and unique solutions for $0 < N < 27$ are respectively given in sequences A000170[2] and A002562[3] of the On-Line Encyclopedia of Integer Sequences (OEIS). This information is summarised in Table 1 for N up to 10. A variant of the problem consists of finding the minimum number of k queens needed to occupy or attack all squares of a $N \times N$ board. For the 8-queens puzzle the answer is $k = 5$, with 91 unique configurations. The summary for this problem variant is given in the last two lines of Table 1.

3 Solution

In this section we present our GROOVE solution for the N-queens puzzle, and the results obtained. The rationale as to why the solution was modelled in this way gives rise to a set of design guidelines, presented in Section 4.

An important feature of GROOVE is its ability for isomorphism detection, potentially enabling large state space reductions. For this to be applicable to the problem at hand, the graph representation used should be modelled so as to be isomorphic whenever possible. This means that (in terms of Table 1) we aim to find *unique* and not *distinct* solutions.

3.1 Initial considerations

A GROOVE grammar has at least two main components: a *start graph*, describing an initial configuration; and a *rule set*, describing the actions to be performed. Typically, one starts by designing the start graph, followed by the rules. However, in many cases there is an iterative cycle, where some insights gained during rule creation lead to changes in the start graph.

[2] http://oeis.org/A000170
[3] http://oeis.org/A002562

The design of a grammar is a creative process akin to programming, and as such, some familiarity with the tools employed (programming language, compiler, etc) is expected in order for "effective" solutions to be obtained. Analysing our experiences with novice (student) GROOVE users, it is clear that an important skill that needs to be acquired is the ability to think declaratively and in terms of graph structure. Unfortunately, such ability is hard to teach and also to condense as guidelines; it can usually only be acquired by practice. Nevertheless, as with many intellectual endeavours, once the proper mindset is achieved it becomes a second nature.

Continuing with the programming analogy, the definition of what an "effective" solution means is up to debate. In problems of the kind considered in this paper, effectiveness certainly requires computational efficiency. State space exploration is inherently a complex combinatorial problem, and as such, we consider a grammar "effective" if it produces the desired answer within a reasonable amount of time, and without exhausting the available resources (*e.g.*, memory). Once more, a "reasonable amount of time" is still a subjective metric, dependent on the user view of the problem complexity. We return to this discussion in Section 3.4.

Finally, we would like to point out the interplay between certain modelling choices and the associated difficulty in rule creation. In some cases, resorting to a simpler start graph representation can make the rules more complex, and therefore, harder to write. Additionally, the way a grammar is modelled may have a significant impact on tool performance. When attacking a new problem, we follow the motto by Knuth that "premature optimisation is the root of all evil" [Knu74], and thus we initially strive for ease in grammar design. Only after an initial grammar is crafted, if performance happens to be problematic then some tweaking is done. Further discussion about this point is interspersed throughout the rest of this paper.

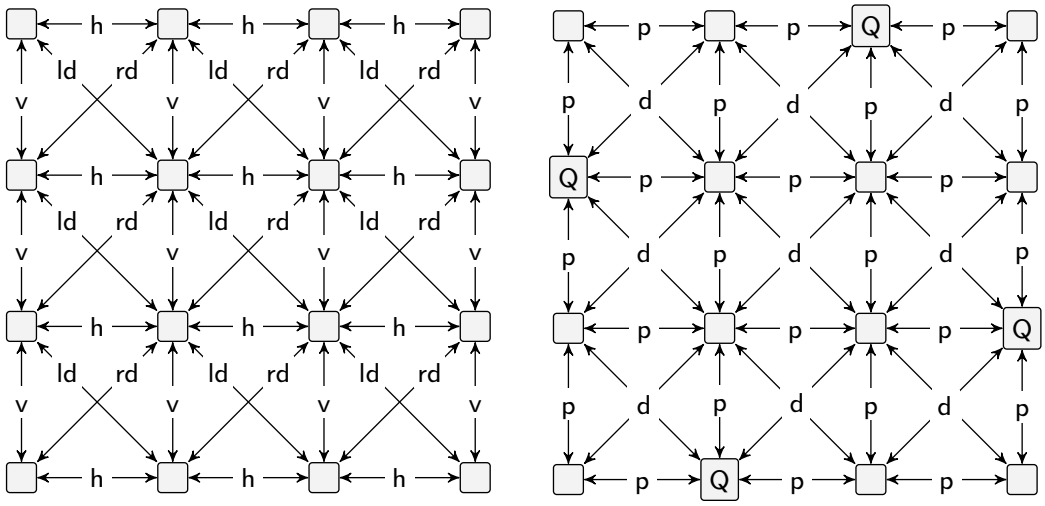
3.1.1 Start graph — representing the board

Representing a chessboard as a graph can be done in several ways. In the course of this section we present the options that were considered until the final representation was found.

We begin by modelling each board square as a graph node. The immediate question that follows is which characteristics of a square should be transferred to its associated node. We recall that a key aspect in abstracting a real-world artefact (the chessboard) into a graph model is preventing unnecessary details from creeping into the model. For instance, while in the real board a square can either be "light" or "dark", this characteristic is irrelevant for solving the puzzle, and therefore square colour is not represented in the graph node.

Each square of the real-world 8×8 chessboard is identified by its column (denoted with letters *a* to *h*) and its row (denoted with numbers *1* to *8*). Such naming convention could in principle be used to distinguish each of the 64 nodes in the graph, but we discard this idea for two reasons:

1. The valid placement of a queen in a node becomes complex to represent in the rules, since the conditions for attacks (no other queen in the same row, column or diagonal) have to be formulated in terms of square attributes. Note that while such conditions can be easily expressed and tested for in a conventional imperative programming language, when working with attributes in a purely graph-based setting this is not so straightforward.
2. Making each node totally distinct from the others completely eliminates graph symmetry under rotations and reflections. This effectively renders the isomorphism checking of GROOVE moot.



(a) Start graph representing a 4×4 board (b) Unique solution (after edge renaming)

Figure 2: Start and final graphs for the 4-queens problem.

In the end, we represent an empty square simply as an unlabelled node, ▢ ; and a square occupied by a queen as a Q-labelled node, Q . This simplistic node model implies that node distinction has to be based on incident edges. Figure 2(a) shows the start graph representation used to model a 4×4 board. Nodes are distinguished by their neighbours, identified by the connecting edges. Edges labelled h and v indicate horizontal and vertical neighbourhood relations, respectively. Diagonals are distinguished by their orientation: NW-SE diagonals are called *left diagonals* and are labelled ld; NE-SW diagonals are called *right diagonals* and are labelled rd. Note that all edges are bidirectional[4] (*i.e.*, undirected) since all relations are symmetric.

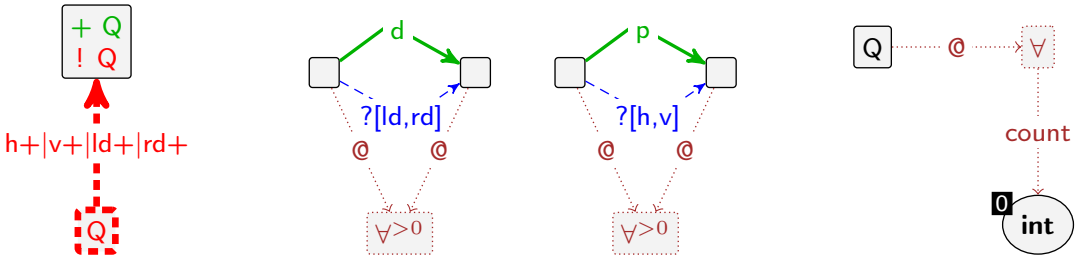
3.1.2 Rules — constructing a solution

It can be immediately seen that the start graph given in Figure 2(a) has only one degree of symmetry, namely over a 180° rotation, due to the distinction between horizontal and vertical edges and between left and right diagonals. However, this representation is convenient as it allows us to solve the problem with a single rule for queen placement. After a solution is found, we "repair" the board with another rule that restores symmetry.

Figure 3 shows the rules of the grammar. The place rule performs a single step towards a valid solution, by placing a queen in a suitable square. The topmost node in the rule tries to find a square that does not already have a queen, as indicated by the embargo[5] condition ! Q inside the node. The candidate square should also not be under attack by another existing queen: this is encoded by the bottom (*embargo*) node, coloured in red. It should not be possible for an existing queen to reach the selected square in a straight line from any direction, either horizontally,

[4] GROOVE works only with directed edges, so to represent a symmetric relation between two nodes we have to use two opposite directed edges. These edges are rendered in the GUI as a single bidirectional (double-arrowed) edge to avoid visual clutter.

[5] *Embargo* is the GROOVE terminology for a *negative application condition (NAC)*.



(a) place rule (b) rename rule (c) count rule

Figure 3: Rules of the N-queens solution.

vertically, or diagonally. This restriction is imposed by means of the regular expression placed in the edge: h+|v+|ld+|rd+. In this expression, h+ matches a path in the host graph formed by one or more h-edges (and similarly for the other labels), and | represents a path choice. If indeed a proper square can be found, a new queen should be placed there, as indicated by the *creator* label + Q inside the reader node.

Rule rename (Figure 3(b)) renames all edges of the board in order to restore the other symmetries. In this way, GROOVE can use isomorphism checking to collapse similar final configurations, thus obtaining only the unique solutions for the puzzle. Symmetry is achieved by renaming left and right diagonals to d-edges, and by transforming horizontal and vertical edges into p-edges (perpendicular).[6] The rule uses *wildcard labels* ?[ld,rd] and ?[h,v], each of which matches either of the labels between square brackets. Additionally, *universal quantification* modifies all edges in a single application, thus avoiding interleaving due to independent rule applications and therefore simplifying the state space (see Section 4 for further discussion).

At this point one may wonder why the renamed board with only d- and p-edges was not already used as a start graph. The reason for avoiding this simpler representation at the beginning is that it would make queen placement much harder to express. For instance, the regular expression in Figure 3(a) cannot simply be replaced by p+|d+, as this new condition allows paths to "bend" over board corners. To illustrate this problem, see Figure 2(b), which shows the single unique solution for the 4-queens puzzle. For example, the queens at the first column and at the last row can reach one another via a p+ path that goes over the lower left corner, but these queens are not attacking each other. This is a case of the point discussed at the last paragraph of Section 3.1, where a compromise between rule and start graph complexity has to be achieved.

Finally, we do not only want to *find* a solution but also *report* it; in particular, we want to count the number of queens that have been successfully placed at the end. This is what rule count in Figure 3(c) achieves: it quantifies over all nodes with a Q-label, and lifts the count to a *rule parameter* (indicated by the black adornment on the top left corner of the **int**-node): for instance, if there are 4 Q-labelled nodes as in Figure 2(b), applying this rule will result in a transition bearing the label count(4).

[6] Edges have their label fixed upon creation so label renaming requires deleting the edge with the old label and creating another edge with the new label. In GROOVE, deletions are represented by the so-called *eraser* elements, drawn in dashed blue.

3.1.3 Typing — improving the solution

The start graph and rules we have discussed above are *untyped*: we have used particular labels with a certain intuition, but there was nothing to prevent us from choosing other labels, and nothing to warn us of a typo. This is excellent for rapid prototyping, but at a certain stage it becomes useful to enforce one's own representation choices by introducing a *type graph*, fixing the allowed labels and assigning *node types* to every node. Our sample case is simple enough to remain understandable and maintainable in the absence of typing: for instance, there is only a single node type (representing a square of the board). The actual rule system made available at groove.cs.utwente.nl does use typing.

3.2 Control of rule application

Rule rename should only be applied after all queens are placed because it modifies the board representation expected by rule place. Rule count, furthermore, should come after rename because we want to count unique solutions modulo all possible degrees of symmetry. To ensure such order of rule applications, we use a *control program*, which instructs GROOVE on which rules to try, and in which order.

Listing 1 shows the control program used in our solution. Line 1 in the program tries to place queens in the board as long as possible, as indicated by the keyword **alap**. When rule place can no longer be applied we have found a valid solution for the k queens coverage problem. Among these solutions we also have the answer for the original problem when $k = N$. Note that, by construction, the inapplicability of rule place is a sufficient condition for obtaining a k coverage solution; when GROOVE cannot find a node to apply the rule to, we can conclude that every square of the board is either occupied or under attack.

Listing 1: Control program for rule application.

```
1 alap place; // Place queens as long as possible
2 rename; // Rename the edges to get a symmetric board
3 count; // Count the number of queens placed
```

After all queens are placed, control moves to line 2 of Listing 1, where rule rename is applied. Since this rule operates over the entire board due to quantification, only a single invocation is necessary. Following this renaming, all distinct solutions are collapsed under isomorphism (done automatically by the tool during exploration) and the remaining final states correspond only to unique solutions.

The last step of the control program invokes count and thus marks the final states with the corresponding number of queens. As explained above, the rule does not actually modify the graph, but will result in a transition labelled by the (parameterised) rule.

3.3 Extracting information from the state space

After the state space is fully explored and stored as a labelled transition system (LTS), we can analyse it to obtain the desired information. For our running example, we are interested in the number of final states marked by the count rule. These states also enumerate all unique solutions of a problem instance.

For N up to five, the state space is quite small and can be directly inspected using the LTS view of the Simulator. This approach is quite useful during the grammar design, as it allows the user to interactively experiment with the rules, and to have an immediate feedback after an exploration.

However, for $N > 5$ the state space grows to more than a thousand states, which renders visual inspection unfeasible. For these cases, we use the Prolog extension of GROOVE [GZR+11], which lets us query the state space programmatically.

Listing 2: Prolog program for querying the transition system.

```
1  % Usage: rule_application_target(+RuleName, ?Target)
2  % Finds the resulting state after an application of a rule with a given name.
3  rule_application_target(RuleName, Target) :-
4    state(Source), % Get a source state in the LTS
5    state_transition(Source, Transition), % Get a transition from source state
6    edge_label(Transition, RuleName), % Test whether the transition label is the one we seek
7    transition_target(Transition, Target). % Get the target state from the transition
8
9  % Predicate to print the number of final states with a given N queens count.
10 report(N) :-
11   length(RuleNames, N), append(RuleNames, _, % Select the N first elements of the list of transition labels
12   ['count(1)', 'count(2)', 'count(3)', 'count(4)', 'count(5)', 'count(6)', 'count(7)', 'count(8)', 'count(9)', 'count(10)']),
13   member(RuleName, RuleNames), % Pick one rule name to use
14   findall(State, rule_application_target(RuleName, State), States), % Collect all states where the rule was applied
15   write('␣␣'), write(RuleName), write(':␣'), length(States, L), write(L), write('␣states.'), nl, % Report
16   fail % Fail so we backtrack to 'member' and pick another rule name
17   .
```

Listing 2 shows the Prolog program used for state space analysis. Lines 1–7 show the definition of a predicate for finding a state in the LTS that is the application target of the given rule. Note that the body of rule_application_target is composed entirely of predicates created for the Prolog extension of GROOVE. These provide the user with the power to access the internal GROOVE data structures without having to delve into the tool code.

After exploring the state space, we use the predicate in lines 9–17 to report the result counts. Predicate report selects the N first elements from the rule name list in line 12, and proceeds to collect all states marked with each rule on the list (lines 13–14). For each collected set, its size is printed (line 15). The results of a query for the 6-queens puzzle is shown on the right. For a much more detailed description about the Prolog extension and its usage, see [GZR+11].

```
?− report(6)
  count(1): 0 states.
  count(2): 0 states.
  count(3): 0 states.
  count(4): 17 states.
  count(5): 28 states.
  count(6): 1 states.
```

3.4 Results

After the grammar was finished, we performed tests (state space explorations) with N ranging from 2 up to 10 [7]. When analysing the test results we are interested in two main points:

- **Grammar testing**: the solution counts found for each value of N should match the numbers given in Table 1.
- **Grammar (tool) performance**: the time required for a full space state exploration should be "reasonable", when considering the state space size of each instance.

Table 2 summarises the results obtained. For each value of N the corresponding line shows: the state space size in terms of state and transition counts; the time needed to perform a full state

[7] Picking a value for N amounts to selecting a properly sized start graph from the grammar.

N	State space size		Time	Uni.	Solution count for coverage with k queens									
	States	Transitions	(s)	sols.	1	2	3	4	5	6	7	8	9	10
2	5	7	< 1	0	1	0	–	–	–	–	–	–	–	–
3	14	24	< 1	0	1	1	0	–	–	–	–	–	–	–
4	53	117	< 1	1	0	0	2	1	–	–	–	–	–	–
5	254	717	1	2	0	0	2	4	2	–	–	–	–	–
6	1,429	4,810	2	1	0	0	0	17	28	1	–	–	–	–
7	8,954	35,785	10	6	0	0	0	1	160	69	6	–	–	–
8	62,159	289,581	79	12	0	0	0	0	91	871	307	12	–	–
9	472,424	2,529,678	774	46	0	0	0	0	16	968	4,848	1,335	46	–
10	3,862,735	23,362,184	8,538	92	0	0	0	0	1	107	10,443	30,278	6,199	92

Table 2: Results obtained by varying N with isomorphism checking enabled.

space exploration; the number of unique solutions with N queens found; and the solution count for the k queens coverage problem. We discuss each of these numbers in turn.

State space sizes show the usual explosion, *i.e.*, an exponential growth on the number of states w.r.t. N. This kind of explosion is commonplace within the setting of combinatorial problems. The 3.8 million states computed for $N = 10$ lie around the usual limit the tool can handle. At the time of this writing, the maximum number of states standard GROOVE has ever reached in a published experiment was around 7 million [GMRZ10], for a grammar modelling a network protocol for car platoon construction in highways. Using an experimental version of the tool with distributed exploration, this limit was raised to 35 million states [BKR10] but this required the use of extra tools for parallel execution.

The fourth column of Table 2 lists the running times for state space exploration using a DFS traversal[8]. DFS was preferred over BFS as it usually requires less memory [GMRZ10]. All tests were run on a 64-bit laptop with 6GB of RAM. The running time for the 8-queens problem was slightly over 1 minute. While this might seem high when compared to a dedicated solution written in a common programming language, we consider this time reasonable for a solution obtained by a general purpose tool, *i.e.*, that has no particular heuristics for the problem. When N increases to 9 and 10, the respective running times of 12 minutes and two and half hours can no longer be considered reasonable.

Moving to the columns presenting the solution counts, we see that the unique solutions column of Table 2 matches the second line in Table 1. The same can be said about the results for the k queens coverage problem, where the first non-zero column in Table 2 shows the minimal k for solving the problem and the number of unique solutions for such k. It is interesting to point out that our grammar not only finds all solutions for the minimal k and $k = N$ cases, but also all other solutions for k lying in between these extremes.

4 Guidelines

We are now ready to generalise the content from the previous section into some tool usage guidelines. As discussed in the introduction, it should be born in mind that these apply in the

[8] Querying times for the Prolog program are not shown since they are negligible when compared to exploration time.

N	State space size		Time (s)	Distinct solutions
	States	Transitions		
2	13	12	< 1	0
3	36	43	< 1	0
4	123	220	< 1	2
5	578	1,411	1	10
6	3,331	9,676	3	4
7	20,594	72,427	15	40
8	139,345	585,256	131	92
9	1,030,642	5,098,357	1,241	352

Table 3: Results obtained by varying N with isomorphism checking disabled.

problem class of state space search (which occurs for instance in model checking), where the dreaded space state explosion has to be tamed. For users employing GROOVE in other settings, [GMR+12] offers some more recommendations.

Furthermore, the guidelines below focus mainly on GROOVE characteristics and features; we do not explicitly present generalisations applicable to other tools as this would require an extensive comparison of functionalities. However, users of other similar tools can lean on the similarities to transfer these guidelines to their own tool, when applicable. Comparisons between GROOVE and other existing graph transformation tools are given in [GMR+12, JBW+13].

Minimalism is beautiful. State space exploration entails generating and *storing* all reachable graphs of the grammar, meaning that every rule application yields a new state that has to be kept in memory. While several optimisations are programmed in the tool (*e.g.*, keeping only the deltas between source and target states), the bottom line here is *keep your graphs to a bare minimum*, as smaller and simpler graphs usually help performance. Of course, as discussed in Section 3.1, this recommendation needs not to be taken to extremes. In the end, the user is presented with an engineering problem, where a proper balance between graph simplicity, performance, and ease of grammar design has to be found. An exception should be made here concerning *remark* nodes and edges, which can be used to document rules: these are quite helpful and cause no overhead in the exploration.

Symmetry is your friend. The main feature that distinguishes GROOVE from other model checkers is its representation of states as graphs. This representation in turn allows the tool to perform symmetry reduction of similar states, by collapsing isomorphic graphs into a single canonical representative.

Much research has been done to make isomorphism checks fast. GROOVE uses *certificates* (a sort of graph hash) to speed up comparisons, which works very well in practice [Ren06a]. To illustrate the impact symmetry reduction has on our running example, we ran exploration tests for N up to 9 with isomorphism checking disabled, with Table 3 showing the results. Taking for instance the 8-queens problem, the state space doubles in size when isomorphism checks are not used, with the running time for the exploration also doubling accordingly.

Of course, for isomorphism checks to be effective, state graphs need to contain a certain degree of symmetry. When this symmetry is not present in the real world artefact being modelled, the checks can be disabled to avoid unnecessary overhead. However, in general the user should

strive for host graph representations that have symmetry, as was done for this case study. A suggestion for achieving this is sticking to standard graphs (with nodes and edges only) when possible. An example was discussed in Section 3.1: attributes make nodes distinct and therefore break symmetry.

Be greedy when designing your rules. Here the recommendation is simple: *do as much as possible in one rule* to avoid producing unnecessary intermediates states. This was illustrated in Figure 3(b) with rule rename; we want to relabel all edges of the board, so we do it in one step using universal quantification. Had this relabelling been done in a stepwise fashion, it would generate possibly many additional states storing the mixed board representation; a wasted effort since all these states end up in the same final configuration.

GROOVE allows nested quantification of rule conditions, enabling one to write very expressive transformation rules [RK09]. In [Ren06b], an analysis of a gossiping network protocol showed that the grammar state space is reduced by 2 orders of magnitude when quantification is used.

Keep your rules under control. The interleaving of rule applications is one of the big culprits of state space explosion, as every sequence of applicable rules have to be considered. Furthermore, rule matching is an NP-complete problem, and therefore, computationally expensive. Luckily, rule applications usually follow some predetermined sequence that can be used. The simplest form of rule ordering in GROOVE is done with priorities: a lower priority rule is only scheduled for matching if all other rules with higher priority are not applicable. A more powerful and flexible form of scheduling uses control programs, such as the one shown in Listing 1. Except for the most simpler cases, the use of control programs is usually preferred over priorities, since programs allow more complex rule sequencing. In the end, the take away message is: *use some form of rule scheduling whenever possible*.

Make sure to get your results out. The simplest form of state space analysis is by visual inspection but this can only be done for very small transition systems (LTSs). For larger LTSs, the usual method is model checking, and GROOVE supports both variants: CTL and LTL formulæ. However, model checking can be cumbersome as it requires the entire property of interest to be written in a single logical formula. A much more flexible method is the use of Prolog queries, such as the one given in Listing 2. All in all, here the message is: *there are several methods for state space analysis, make sure to pick the one most suitable to your needs*.

5 Conclusion

We usually advertise GROOVE as a flexible modelling tool, specially suited for rapid prototyping [GMR+12]. Indeed, at least for us, GROOVE is flexible and easy to use, but our own tool experience is obviously not a good representative of the "average" user. The goal of this paper was to bridge the gap (or at least shrink it) between novice and more advanced GROOVE users by pointing to information that we, as tool developers, consider important but that up to now was not properly available and documented.

As a side note, we are aware that this text contains a high number of imprecise terms such as "usually", "generally", "in some cases", etc. We hope for some understanding from the readers

in this matter, as this language use is an unfortunate side-effect of writing about best practices and guidelines. For every rule there is an exception, and making clear-cut statements about certain topics would make them dubious at best, and plain wrong at worst.

The N-queens puzzle is a good representative for the class of combinatorial problems that can be tackled using GROOVE, and therefore the information presented here is geared towards this kind of problem. Future work w.r.t. this form of documentation points to the development of similar tool usage guidelines for other problem classes, such as model transformation. A more extensive line of future work points towards a tool usability study, where an empirical evaluation with novice GROOVE users could help identify the strengths and weaknesses of the tool.

Related tools. In this paper we have concentrated on the capabilities and features of GROOVE; as stated in the beginning of Section 4, our guidelines have been formulated especially with this in mind. Of the rich set of other graph transformation-based tools, we believe that HENSHIN [ABJ+10] comes closest in matching the particular capabilities of GROOVE. For a very comprehensive overview of other modelling graph-based tools see [JBW+13].

Availability. The experiments presented in this paper were performed with GROOVE version 4.9.2, available at http://groove.cs.utwente.nl. The grammar for solving the N queens puzzle can also be downloaded at the same address.

Bibliography

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems (MODELS), Part I*. LNCS 6394, pp. 121–135. Springer, 2010.

[BKR10] S. C. C. Blom, G. Kant, A. Rensink. Distributed Graph-Based State Space Generation. In *International Workshop on Graph-Based Tools (GraBaTs)*. Electronic Communications of the EASST 32. European Association of Software Science and Technology (EASST), 2010.

[DDH72] O. J. Dahl, E. W. Dijkstra, C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., London, UK, 1972.

[GMR+12] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon, M. Zimakova. Modelling and Analysis Using GROOVE. *International Journal on Software Tools for Technology Transfer (STTT)* 14(1):15–40, 2012.

[GMRZ10] A. Ghamarian, M. de Mol, A. Rensink, E. Zambon. Solving the Topology Analysis Case Study with GROOVE. In *Transformation Tool Contest (TTC)*. 2010. http://is.ieis.tue.nl/staff/pvgorp/events/TTC2010/submissions/final/groove.pdf.

[GZR+11] I. Galvão, E. Zambon, A. Rensink, L. Wevers, M. Aksit. Knowledge-based Graph Exploration Analysis. In *International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. LNCS 7233, pp. 121–136. Springer, 2011.

[JBW+13] E. Jakumeit, S. Buchwald, D. Wagelaar, L. Dan, Hegedüs, M. Hermannsdörfer, T. Horn, E. Kalnina, C. Krause, K. Lano, M. Lepper, A. Rensink, L. M. Rose, S. Wätzoldt, S. Mazanek. A Survey and Comparison of Transformation Tools Based on the Transformation Tool Contest. *Science of Computer Programming*, pp. 1–59, November 2013.

[Knu74] D. E. Knuth. Structured Programming with Goto Statements. *ACM Comput. Surv.* 6(4):261–301, 1974.

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In *International Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. LNCS 3062, pp. 479–485. Springer, 2003.

[Ren06a] A. Rensink. Isomorphism Checking in GROOVE. In *International Workshop on Graph-Based Tools (GraBaTs)*. Electronic Communications of the EASST 1. European Association of Software Science and Technology (EASST), 2006.

[Ren06b] A. Rensink. Nested Quantification in Graph Transformation Rules. In *International Conference on Graph Transformations (ICGT)*. LNCS 4178, pp. 1–13. Springer, 2006.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the Geraniums: On Nested Graph Transformation Rules. In *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT)*. Electronic Communications of the EASST 18. European Association of Software Science and Technology (EASST), 2009.