# Chapter 7

# CONCLUSION

We conclude with some general as well as concrete directions for extending the work presented in this thesis.

Our results in Chapter 3 imply exponential lower bounds on the running time of a class of backtracking algorithms for finding a maximum independent set (or, equivalently, a maximum clique or a minimum vertex cover) in a given graph, or approximating it. Analysis of the complexity of the independent set and other related problems under stronger proof systems, such as Cutting Planes [65, 29], bounded-depth Frege systems [2], or an extension of resolution that allows "without loss of generality" reasoning as mentioned in Section 3.11, will broaden our understanding in this respect.

The DPLL upper bounds that we give are based on a rather simple enumeration, with natural search space pruning, of all potential independent sets. As Theorem 3.5 points out, this is the best one can do using any exhaustive backtracking algorithm. Considering more complex techniques may let us close the gap of a factor of nearly $O(\Delta^5)$ in the exponent that currently exists between our lower and upper bounds for general resolution. It appears that we have not taken advantage of the full power of resolution, specifically the reuse of derived clauses.

The work presented in Chapter 4 inspires but leaves open several interesting questions of proof complexity. We showed that there are formulas on which CL is much more efficient than any proper natural refinement of RES. In general, can every short refutation in any such refinement be converted into a short CL proof? Or are these refinements and CL incomparable? We have shown that with arbitrary restarts, a slight variant of CL is as powerful as RES. However, judging when to restart and deciding what branching sequence to use after restarting adds more nondeterminism to the process, making it harder for practical implementations. Can CL with limited restarts also simulate RES efficiently?

We also introduced in that chapter FirstNewCut as a new learning scheme and used it to derive our theoretical results. A characterization of the real-world domains on which it performs better than other schemes is still open. In the process of deriving theoretical results, we gave a formal description of concepts such as implication and conflict graphs, and how they relate to learned clauses and trivial resolution derivations. This framework, we hope, will be useful in answering the complexity questions left open by this work.

In Chapter 5, the form in which we extract and use problem structure is a branching sequence. Although capable of capturing more information than a static variable order and avoiding the overhead of dynamic branching schemes, the exactness and detail branching sequences seem to require for pebbling formulas might pose problems when we move to harder domains where a polynomial size sequence is unlikely to exist. We may still be able to obtain substantial (but not exponential) improvements as long as an incomplete or approximate branching sequence made correct decisions most of the time, especially near the top of the underlying DPLL tree. The performance gains reported for $GT_n$ formulas indicate that even a very simple and partial branching sequence can make a big difference in practice. Along these lines, variable orders in general have been studied in other scenarios, such as for algorithms based on BDDs [see e.g., 11, 61]. Reda et al. [96] have shown how to use BDD variable orders for DPLL algorithms without learning [96]. The ideas here can potentially provide new ways of capturing structural information.

From Chapter 6, the symmetry representation and maintenance techniques developed for SymChaff may be exploited in several other ways. The variable selection heuristic of the DPLL process is the most noticeable example. This framework can perhaps be applied even to local search-based satisfiability tools such as Walksat by McAllester et al. [86] to make better choices and reduce the search space. As for the framework itself, it can be easily extended to handle *k-ring* multi-class symmetries, where the $k$ underlying indices can be rotated cyclically without changing the problem (e.g. as in the `PlanningB` problem, Example 6.3). However, the best-case gain of a factor of $k$ may not offset the overhead involved.

SymChaff is the first cut at implementing our generic framework and can be extended in several directions. Learning strategies for symconflict clauses other than the "decision variable scheme" that it currently uses may lead to better performance, and so may dynamic strategies for selecting the order in which various branches of a multiway branch are traversed, as well as a dynamic equivalent of the static `.ord` file that SymChaff supports. Extending it to handle pseudo-Boolean constraints is a relatively straightforward but promising direction. Creating a PDDL preprocessor for planning problems that uses graph isomorphism tools to tag symmetries in the PDDL description would fully automate the planning-through-satisfiability process in the context of symmetry.

On the theoretical side, how does the technique of SymChaff compare in strength to proof systems such as RES with symmetry? It is unclear whether it is as powerful as the latter or can even efficiently simulate all of RES without symmetry. Answering this in the presence of symmetry may also help resolve an open question from Chapter 4 of whether clause learning (without symmetry) can efficiently simulate all of RES.