

An Improved BDD Method for Intuitionistic Propositional Logic: BDDIntKt System Description

Rajeev Goré and Jimmy Thomson

Logic and Computation Group, Australian National University

Abstract. We previously presented a decision procedure for satisfiability and validity in propositional intuitionistic logic **Int** using Binary Decision Diagrams (BDDs). We now present some further optimisations which greatly improve performance. Primarily we focus on the impact and placement of an explicit mechanism for BDD variable ordering.

1 Introduction

We assume the reader is familiar with the syntax and Kripke semantics of propositional intuitionistic logic (**Int**). We also assume that the reader is familiar with the notions of satisfiability, validity and global logical consequence in **Int**.

For many logics, we can decide the validity of a formula φ_0 by constructing the set of all subsets of some closure $cl(\varphi_0)$, and checking whether these subsets can support a (counter) model that makes φ_0 false. If no such model exists, then we can safely declare φ_0 to be valid using this finite model property (fmp).

We previously showed how to implement this “fmp method” for **Int** without explicitly constructing all exponentially many subsets of the closure by using Binary Decision Diagrams (BDDs) [3]. Here, we present some specific optimisations that significantly improve performance of the BDD method. See: <http://users.cecs.anu.edu.au/~rpg/BDDBiKtProver/optimised-bddintkt.tgz>

1.1 A Terse Overview of the Basic Algorithm

Given φ_0 , our goal is to construct a finite model $\mathcal{M} = (W_f, \preceq_f, \rho)$ by constructing a sequence of frames $(W_0, \preceq_0), (W_1, \preceq_1), \dots, (W_f, \preceq_f)$ such that the final frame gives a model which is “canonical” in two senses: if φ_0 is satisfiable (falsifiable) then some world of W_f satisfies (falsifies) φ_0 . We can then decide whether φ_0 is satisfiable or valid by checking whether such worlds exist.

We first compute $Atoms(\varphi_0) = \{\psi \in subfml(\varphi_0) \mid \psi \in Prop \vee \psi = \psi_1 \rightarrow \psi_2\}$, where *Prop* is the set of all atomic propositions. Any subset of $Atoms(\varphi_0)$ corresponds to a classical valuation on the atoms since each atom is either in the set or not. Thus $\mathcal{W} = 2^{Atoms(\varphi_0)}$ contains all valuations over $Atoms(\varphi_0)$, and contains any valuation corresponding to any world in any model for φ_0 . Under

this view, each $w \in \mathcal{W}$ is both a world and a valuation. Thus the denotation of an atom $a \in \text{Atoms}(\varphi_0)$ is $\llbracket a \rrbracket := \{w \in \mathcal{W} \mid a \in w\}$. The denotations $\llbracket \varphi \wedge \psi \rrbracket$ and $\llbracket \varphi \vee \psi \rrbracket$ of arbitrary conjunctions and disjunctions from $cl(\varphi_0)$ are computed using intersections and unions of the denotations of their subformulae. Using these denotations, we compute $\preceq_{max} \subseteq \mathcal{W} \times \mathcal{W}$, an over approximation of the intuitionistic Kripke binary relation, and $\mathcal{W}_{refl}$, the subset of $\mathcal{W}$ where $\preceq_{max}$ is reflexive.

We then monotonically refine an initial approximation $W_0 \subseteq \mathcal{W}$ towards $W_f \subseteq \mathcal{W}$, using the constructed $\preceq_{max}$ relation to enforce the correct modal interpretation of the elements of $cl(\varphi_0)$ in all the worlds. Since $\preceq_i$ is just the restriction of $\preceq_{max}$ to W_i we do not need to compute it explicitly.

Once W_f has been computed, the final step is to determine which, if any, worlds in W_f satisfy and falsify φ_0 , giving the satisfiability and validity of φ_0 .

In [3], we started with $W_0 = \mathcal{W}$ but noted that we could instead start with $W_0 = \mathcal{W}_{refl}$. Using this latter approach, we can compute W_f as the greatest fixpoint of the sequence $W_0 := \mathcal{W}_{refl}, W_1, \dots, W_f$ where $\overline{X} = \mathcal{W} \setminus X$ and:

$$\begin{aligned} \mathcal{W}_{refl} &= \bigcap_{\phi \rightarrow \psi \in \text{Atoms}(\varphi_0)} \overline{\llbracket \phi \rightarrow \psi \rrbracket \cup \llbracket \overline{\phi} \rrbracket \cup \llbracket \psi \rrbracket} \\ W_i^{\varphi \rightarrow \psi} &= \llbracket \varphi \rightarrow \psi \rrbracket \cup \{x \mid (x, y) \in \preceq_{max} \wedge y \in (W_i \cap \llbracket \varphi \rrbracket \cap \overline{\llbracket \psi \rrbracket})\} \\ W_{i+1} &= W_i \cap \bigcap_{\varphi \rightarrow \psi \in \text{Atoms}(\varphi_0)} W_i^{\varphi \rightarrow \psi} \end{aligned}$$

1.2 Basic Implementation Details

We now give some lower level details on how BDDs are actually used to perform these computations since all our optimisations are at this implementation level.

First, we analyse the initial formula φ_0 and create the set of atoms $\text{Atoms}(\varphi_0)$. During this process, we create two BDD variables a and a' for each atom, and construct the BDDs representing $\llbracket \psi \rrbracket$ and $\llbracket \psi \rrbracket'$ for each subformula ψ of φ_0 according to the definition of $\llbracket \cdot \rrbracket$. For atoms a , the BDD representing $\llbracket a \rrbracket$ is the BDD which is true exactly when the variable for a is true. For non-atomic formulae, the BDD representing $\llbracket \psi \rrbracket$ is a conjunction or disjunction of the BDDs representing the conjuncts or disjuncts as appropriate. The unprimed variables like a give rise to $\mathcal{W}$ while the primed variables like a' give rise to a “photocopy” $\mathcal{W}'$ of $\mathcal{W}$ so that $\preceq_{max}$ is actually a subset of $\mathcal{W} \times \mathcal{W}'$: see [3] for details.

During this initial creation of the BDD variables, we implicitly specify an ordering on the BDD variables according to when they are created, so if the atoms are examined in different orders then the variable ordering can be different.

After the initial analysis, we construct the BDD for $\preceq_{max}$ corresponding to the intuitionistic Kripke relation: see [3] for details. For **Int**, after defining $\mathcal{W}_{refl}$ as above, we can in fact reduce the intuitionistic relation to simply enforcing persistence, so it is a relatively simple conjunction over all atoms:

$$\preceq_{max} = \bigwedge_{a \in \text{Atoms}(\varphi_0)} \llbracket a \rrbracket \Rightarrow \llbracket a \rrbracket' = \bigcap_{a \in \text{Atoms}(\varphi_0)} (\overline{\llbracket a \rrbracket} \times \mathcal{W}') \cup (\mathcal{W} \times \llbracket a \rrbracket').$$

We next compute $\mathcal{W}_{refl}$, the initial over approximation from Section 1.1. Any global assumptions are also conjoined with this initial state. The set $\mathcal{W}_{refl}$ is a conjunction of formulae, and the conjuncts can be arbitrarily complex.

The remainder of the procedure is the fixpoint computation: while the BDD $W_i \neq W_{i-1}$, compute $W_i^{\phi \rightarrow \psi}$ for each atom $\phi \rightarrow \psi$, and conjoin the resulting BDDs with W_i to give W_{i+1} . To compute $W_i^{\phi \rightarrow \psi}$, we take W_i and replace all variables with their next-state (primed) duplicates giving W'_i , and then conjoin with $\llbracket \phi \rrbracket' \wedge \neg \llbracket \psi \rrbracket'$: recall that these are (classical, not intuitionistic) operations on BDDs. Next, we conjoin with the relation BDD $\preceq_{max}$ and existentially quantify out all the primed BDD variables referring to the next state. This leaves us with a BDD representing those worlds which have some potentially good successor satisfying the required conditions. Finally, disjoin with $\llbracket \phi \rightarrow \psi \rrbracket$ to give $W_i^{\phi \rightarrow \psi}$.

Once $W_i = W_{i-1}$, all that remains is to determine whether $W_i \wedge \neg \llbracket \varphi_0 \rrbracket = \perp$.

2 Optimisations

We continue to use pre-processing optimisations such as rewriting formulae to equivalent simpler formulae, and converting $\Gamma \models \psi \rightarrow \phi$ into $\Gamma, \psi \models \phi$ [3]. The following implementation rather than algorithmic changes, also helped.

Reduced ordered BDDs give unique representations of a given Boolean function and allow fast comparisons and efficient storage. The ordering placed on the BDD variables has a significant impact on the time and memory behaviour of BDD operations. Using a good ordering can give significantly smaller representations of the same Boolean function, so finding a good ordering is important.

Initial Order. When analysing the pre-processed formulae to find the atoms and precompute the BDDs for denotations, the order in allocating atoms gives an initial ordering on the BDD variables. After some experimentation, we decided on the following approach: for each non-propositional atom a , construct BDDs for all components in a depth-first left to right manner, creating new BDD variables for encountered atoms not already associated with a BDD variable. Then construct the BDD variable for a itself. Once all non-propositional atoms are handled, finally construct the BDD for the initial formula in a depth first left to right manner, stopping at atoms and creating new variables as appropriate.

Dynamic Reordering. The BDD library [1] we used has the ability to dynamically change the ordering of the BDD variables to try to find a better ordering, and this reordering takes more time when there are more and larger BDDs constructed.

Single Reordering. We investigated a number of dynamic reordering policies, changing when dynamic reordering would be attempted. Initially, we disabled all reordering and kept the ordering initially created when analysing φ_0 . We then

tried a single dynamic reordering after $\mathcal{W}_{refl}$ was constructed, the idea being that the BDDs that had been constructed up to that point would be broadly similar to the ones to follow, so choosing an ordering based on that may be helpful, and doing only a single ordering would incur only minimal overheads.

More Reordering. After examining the performance on the benchmarks, we noted that there was often a bottleneck in computing $\mathcal{W}_{refl}$, before any variable ordering occurred. We therefore tried another policy of reordering more often, after every conjunct in the large conjunctions. This causes many more reorderings and thus leads to more overhead, but the improved variable ordering can result in improved efficiency, using less memory and speeding up further operations.

Automatic Reordering. We also tried a policy where reordering is performed automatic by the BDD package based on usage rather than at our behest. This has less predictable behaviour, but by reordering only when needed, it can reduce overhead while still giving the time and space benefits of more reorderings.

Why Automatic Ordering Wasn't Used Originally. When first implementing the BDD procedure, we reasoned that the large amount of memory available should be provided up-front to reduce subsequent allocation overheads. Consequently, garbage collection and dynamic reordering took a long time since they consider all BDD nodes. Also, the automatic reordering in BuDDy is triggered by an increase in the number of BDD nodes, so if there are many initial BDD nodes, automatic reorderings are far apart and slow. We therefore tried to minimise the number of reorderings. Since then, through experimenting with different options, we have found that a better option is to start with relatively few BDD nodes and attempt to keep that number small, which interacts better with reorderings.

Structure Sharing. Because of their uniqueness, BDDs can often share structure. Suppose we have 3 BDDs a, b and c , each referring to the variables in the sets VA, VB and VC , respectively, with $VA < VB < VC$ in the variable ordering. The BDD for $a \wedge b$ will share the existing BDD for b in its entirety, but cannot share the existing BDD for a since all a 's "exits" to $\top$ must now point to b . Moreover, the BDD for $(a \wedge b) \wedge c$ will share the existing BDD for c in its entirety, but will share nothing with the BDD for $a \wedge b$. Constructing $a \wedge (b \wedge c)$ instead gives an identical BDD, since it represents the same Boolean function, but which contains the intermediate BDD for $b \wedge c$ in its entirety, which in turn contains the BDD for c in its entirety. This results in more sharing and greater efficiency.

Sorted Conjunctions. To try to benefit from this concept, in each large conjunction or intersection in the decision procedure above, we compute the BDD for each of the conjuncts first, then sort those BDDs such that the "lowest" BDDs (in the tree sense) are combined first, giving greater potential for sharing. In the above example, this would select c first since VC is the "lowest" set of variables, then it would conjoin with b , and finally with a , effectively computing $a \wedge (b \wedge c)$.

3 Results

We initially used the ILTP v1.1.2 propositional benchmarks [5] with a 600 second timeout. However, with some combinations of

	none-	none+	once-	once+	more-	more+	auto-	auto+	GnaA	Gna	fCube	Imogen
SYJ201	11	50	11	50	50	50	25	49	49	50	50	32
SYJ202	16	16	16	16	16	16	12	12	12	11	8	8
SYJ203	24	26	24	26	50	50	50	50	50	24	50	50
SYJ204	50	50	50	50	50	50	50	50	50	50	50	50
SYJ205	21	23	20	22	50	50	50	50	50	22	50	50
SYJ206	50	50	50	50	50	50	50	50	50	50	10	50
SYJ207	12	50	11	50	50	50	33	50	49	50	50	24
SYJ208	13	13	11	11	9	10	9	9	10	10	37	37
SYJ209	23	25	23	24	50	50	50	50	50	23	50	50
SYJ210	50	50	50	50	50	50	50	50	50	50	50	50
SYJ211	14	14	16	16	42	36	50	44	46	9	50	50
SYJ212	35	35	50	50	50	50	50	50	45	32	50	45
Total	319	402	332	415	517	512	479	514	511	381	505	496

Fig. 1. Number of instances solved by each configuration

optimisations all instances except for pigeonhole formulae were solved in less than 20 seconds each. To get a better idea of how the optimisations perform in the long term, we use an extended (unofficial) set of benchmarks (<http://www.iltp.de/download/SYJ2xx-50/SYJ2xx-50.tar.gz>) based on the ILTP, with 50 instances instead of 20 for each of the SYJ2xx formulae, except for SYJ202 and SYJ208 which go only up to 38. This unofficial set was kindly provided by the developers of the library, Jens Otten and Thomas Rath. We also compared the provers on the same randomly generated formulae as in [3]. All benchmarks were performed on 64bit Ubuntu 10.04 with a Core 2 Duo 3.0 GHz processor and 8GB RAM.

Our previous [3] results were from an implementation written in OCaml. To experiment with the BuDDy BDD library more easily and to try to control the lifetime of BDDs more precisely, we moved to a C++ implementation. In the process, some ‘arbitrary’ orders changed, so we include results for ‘Gna’ (our previous OCaml implementation), and ‘GnaA’ (the OCaml implementation with automatic variable reordering). All of our other results are from our C++ implementation. We tried each variant of dynamic reordering (**none**, reordering **once** after computing initial conditions, reordering **more** during each conjunction, and **automatic** reordering) both with (+) and without (-) sorted conjunctions.

We also include results for Imogen v3.0 [4] and fCube v7.0 [2] on the extended benchmarks to compare against other state of the art theorem provers.

From Figure 1 we observe some trends. Sorting conjunctions with no or limited reordering can give significant performance gains, for example in SYJ201 and SYJ207. Without sorting conjunctions, some of these instances require over 4 million BDD nodes, indicating large BDDs and expensive operations on them. With sorted conjunctions these same instances require fewer than 8000 nodes. This effect is less pronounced when more dynamic variable ordering is applied, since the variable reordering also achieves more compact BDDs. Gna, the OCaml implementation from [3], performs significantly better than none-, the C++ implementation without any reordering or sorting, on SYJ201/207 primarily

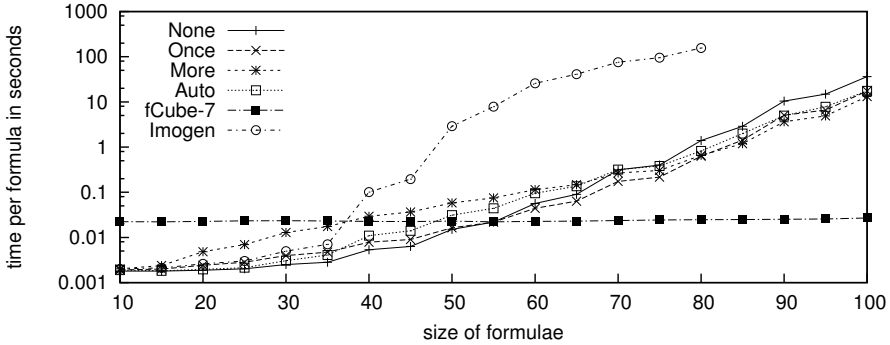


Fig. 2. Average time per random instance

because the order in which the conjunction for W_{refl} is computed is (arbitrarily) reversed, highlighting the importance of computing conjunctions efficiently.

Classes SYJ202 and SYJ208 are encodings of the pigeonhole principle, and most of the time goes into computing the BDD for the initial formula. Also, this BDD appears to be large and difficult to reduce, so reordering gives no benefit but takes up time. Reducing the effort put into dynamic variable reordering leaves more time for constructing the BDDs, and leads to better performance.

SYJ204 and SYJ210 are trivialised by normalisation and converting top level implications into global assumptions. In the case of SYJ204, once W_{refl} is computed the formula is immediately found to be valid, while for SYJ210, one iteration of the fixpoint is required to discover that $W_f = W_{refl}$.

SYJ205 and SYJ211 benefit from increased dynamic BDD variable reordering, although for SYJ205, the overhead of a single reordering on a large BDD outweighs the benefits on the remaining fixpoint computation. On both classes, the bottleneck is the initial state computation, and adding BDD variable reorderings helps to improve performance. For SYJ211, sorting conjunctions gives considerably worse results. Further inspection showed that the ILTP version of these formulae used a_i and b_i as propositions, while the extended versions simply use p_i for both. The structure is identical, but the names have been changed. We think that this interacts with our relatively naive lexicographical initial BDD variable ordering giving an “unstable” situation where, as the initial condition is computed, the “best” ordering keeps changing. If we rename propositions back to a_i and b_i as with the original problem set, then performance improves significantly. Further work in choosing an initial ordering is likely to help since, ideally, the chosen names for propositions should not have any impact on performance.

SYJ206 is rewritten to $\top$ by the formula normalisation preprocessing step. SYJ212 is not so trivialised, but dynamic variable reordering assists greatly.

On the random (typically “obviously” invalid) formulae we see that initially doing extra work does not pay off, but on larger formulae the benefits outweigh the costs. These formulae exposed a bug in fCube-7.0, again highlighting their utility. The repaired version now performs worse on some classes.

Overall, when dynamic variable ordering helps, it helps a lot. If the number of BDD nodes is small, then constructing the fixpoint formulae and dynamically reordering the BDD variables is very fast, meaning that the number of BDD nodes can probably remain small. Computing reorderings after each conjunct in the large conjunctions performed much better than we expected for this reason, although it is probably excessive. Automatic reordering based on load tends to work well for these reasons, but sometimes it can trigger a reordering at an intermediate point, and this appears to result in optimising for the wrong formulae and allows the number of BDD nodes to increase more.

4 Conclusion, Further Work and Acknowledgements

We have dramatically improved performance by paying closer attention to the BDD variable ordering. Dynamically reordering either using the automatic policy in the BDD package [1] or based on milestones in the decision procedure can keep the BDDs involved small, which translates to faster operations. A combination of the automatic reordering based on runtime usage, along with reorderings at certain milestones, may be more effective, and requires further investigation.

Variable ordering affects the size of intermediate BDDs, and computing large conjunctions by sorting them according to variable order can significantly improve performance. However, combining both dynamic variable reordering and sorting conjunctions can lead to interference and reduced performance.

The primary difference in performance between the improved version of fCube and the better performing versions of our prover is the invalid pigeon-hole class SYJ208. Since fCube is a goal-directed procedure, it can stop as soon as it finds an open branch while the BDD method must continue until it reaches a fixpoint before it can determine invalidity. Performance differs on class SYJ206 as well, but very simple pre-processing techniques trivialise this class.

The initial BDD variable ordering clearly plays a significant role in performance, even with dynamic reordering enabled. This is our current focus.

We thank the anonymous reviewers for their comments and suggestions.

References

- [1] Buddy (2011), <http://sourceforge.net/projects/buddy/>
- [2] Ferrari, M., Fiorentini, C., Fiorino, G.: FCUBE: An efficient prover for intuitionistic propositional logic. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR-17. LNCS, vol. 6397, pp. 294–301. Springer, Heidelberg (2010)
- [3] Goré, R., Thomson, J.: BDD-based automated reasoning for propositional bi-intuitionistic tense logics. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 301–315. Springer, Heidelberg (2012)
- [4] McLaughlin, S., Pfenning, F.: Imogen: Focusing the polarized inverse method for intuitionistic propositional logic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 174–181. Springer, Heidelberg (2008)
- [5] Raths, T., Otten, J., Kreitz, C.: The ILTP problem library for intuitionistic logic, release v1.1. *Journal of Automated Reasoning* (2006)